International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 08 Issue: 05 | May - 2024 SJIF Rating: 8.448 ISSN: 2582-3930

Container Mapping and its Impact on Performance in Containerized Cloud Environments

Nandini N S
Assistant Professor
Computer Science and Engineering
B.G.S Institute of Technology
Adichunchanagiri University

Suchithra H M
20CSE081
Computer Science and Engineering
B.G.S Institute of Technology
Adichunchanagiri University

Abstract

Containerization has become a popular approach for deploying and managing applications in cloud environments due to its lightweight nature and scalability. However, optimizing the mapping of containers to underlying cloud resources is critical for maximizing performance and resource utilization. This paper investigates the impact of container mapping strategies on the performance of containerized applications in cloud environments.We begin by examining various container mapping techniques, including static mapping, dynamic mapping, and hybrid approaches. Static mapping assigns containers to resources based on predefined rules or constraints, while dynamic mapping adjusts container placement based on real-time resource availability and workload characteristics. Hybrid approaches combine elements of both static and dynamic mapping to achieve a balance between predictability and adaptability.Next, we evaluate the performance implications of different container mapping strategies through experimentation and analysis. We measure key performance metrics such as response time, throughput, resource utilization, and scalability under varying workload conditions. Our findings highlight the trade-offs between different mapping strategies in terms of performance, resource efficiency, and overhead.

Index Terms—Cloud computing, containers, microservices

I. Introduction

"In recent years, containerization has emerged as a dominant paradigm for deploying and managing applications in cloud environments. Containers offer lightweight, portable, and efficient packaging of software components, enabling rapid deployment and scalability across diverse computing environments. The popularity of containers, exemplified by platforms like Docker and Kubernetes, has revolutionized the way applications are developed, deployed, and managed in cloud-native architectures.

However, while containerization offers numerous benefits, including improved resource utilization, faster deployment, and simplified management, effectively harnessing these advantages requires careful consideration of container mapping strategies. Container mapping involves the allocation of containers to underlying cloud resources, such as virtual machines (VMs) or physical servers, to ensure optimal performance, resource utilization, and scalability.

The selection of appropriate container mapping strategies is crucial for maximizing the benefits of containerization while minimizing overhead and resource contention. Static mapping assigns containers to resources based on predefined rules or constraints, providing predictability and stability but may lead to suboptimal resource utilization. In contrast, dynamic mapping adjusts container placement based on real-time resource availability and workload characteristics, offering greater flexibility and efficiency but introducing overhead and complexity. Understanding the impact of container mapping on performance, resource utilization, and scalability is essential for cloud operators and application developers to make informed decisions. This paper aims to investigate the implications of different container mapping strategies on the performance of containerized applications in cloud environments. By examining various mapping techniques, evaluating their performance under diverse workload conditions, and considering relevant factors influencing mapping decisions, we seek to provide insights and guidelines for optimizing containerized deployments in cloud environments. Containerization has become a dominant approach for deploying and managing applications in cloud environments due to its lightweight, portable, and efficient nature. Platforms like Docker and Kubernetes have played a significant role in popularizing containers, reshaping the development and deployment landscape.

Despite the benefits of containerization, effective utilization requires thoughtful consideration of container mapping strategies. Container mapping involves allocating containers to underlying cloud resources to ensure optimal performance, resource utilization, and scalability.

Two primary mapping strategies exist: static mapping, which assigns containers based on predefined rules, and dynamic mapping, which adjusts placement based on real-time resource availability and workload characteristics. Each strategy presents trade-offs in predictability, stability, flexibility, and efficiency.

Understanding the impact of container mapping on performance is crucial for cloud operators and application developers. This paper aims to investigate different mapping techniques' implications on containerized application performance in cloud environments. By evaluating performance under various workloads and considering relevant factors, the study aims to offer insights and guidelines for optimizing containerized deployments in the cloud.

II. Challenges and Opportunities

Navigating container mapping in cloud environments presents both challenges and opportunities. On one hand, the complexity of resource allocation poses a significant challenge, especially in dynamic environments with fluctuating workloads. Balancing optimal resource utilization while avoiding suboptimal mappings is another hurdle, particularly with the introduction of overhead and latency issues in dynamic mapping strategies. Scalability concerns also emerge, particularly in multi-tenant environments where isolation and fairness are critical. However, amidst these challenges lie opportunities for advancement. Algorithmic improvements, integration with orchestration platforms, and the automation of mapping processes offer promising avenues for enhancing efficiency. Hybrid mapping strategies that combine static and dynamic approaches could provide a balance between predictability and adaptability. Moreover, leveraging performance monitoring and analysis tools enables continuous evaluation and refinement, paving the way for more efficient and performant containerized deployments in cloud environments.

III. Related Work

Previous research has extensively explored various aspects of containerization and its implications for cloud environments. Studies have investigated container orchestration techniques, such as Kubernetes and Docker Swarm, focusing on their effectiveness in managing containerized applications at scale [1]. Additionally, research has examined the performance overhead and resource utilization of different containerization platforms, providing insights into their comparative advantages and limitations [2]. Furthermore, the impact of container mapping on performance has been a subject of interest. Several studies have explored static and dynamic container mapping strategies, assessing their effectiveness in optimizing resource allocation and improving application performance [3]. Additionally, research has investigated the integration of machine learning algorithms into container mapping processes, aiming to enhance predictive accuracy and adaptability [4]. Moreover, the role of containerization in enabling microservices architectures and DevOps practices has been widely studied. Research has examined the benefits of microservices-based application design in terms of modularity, scalability, and fault isolation [5].

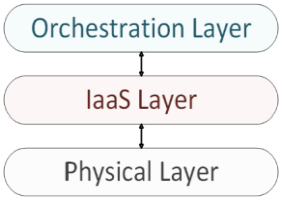
While existing literature provides valuable insights into various aspects of containerization and its impact on cloud environments, there remains a need for further research to address emerging challenges and opportunities. This includes exploring novel container mapping strategies, optimizing container orchestration techniques, and investigating the integration of containers with emerging technologies such as serverless computing and edge computing.

IV. A Layered Reference Architecture

A layered reference architecture provides a structured framework for designing complex systems, offering a systematic approach to organizing components and functionalities. At the topmost layer is the Presentation Layer, which interfaces with users or external systems, delivering information through various user interfaces like web browsers, mobile applications, or APIs. Beneath it lies the Application Layer, where the system's core logic and functionality reside, managing business rules, data processing, and inter-component interactions. The Service Layer encapsulates this logic into reusable services, accessible through APIs or web services, facilitating seamless integration and communication.

The Integration Layer facilitates data exchange and communication between different system components or external systems, managing data transformation and routing. Below, the Data Layer oversees persistent data storage and management, ensuring data consistency, integrity, and security. Supporting these layers is the Infrastructure Layer, comprising hardware and software components that provide the computing resources and runtime environment. The Security Layer safeguards the system against unauthorized access and data breaches, employing authentication, encryption, and access controls.

Lastly, the Management Layer offers tools and capabilities for monitoring, managing, and maintaining system health and performance. By delineating responsibilities and interactions across these layers, a layered reference architecture promotes modularity, scalability, and maintainability, facilitating the design, implementation, and evolution of complex systems.



V. CONTAINER TECHNOLOGIES

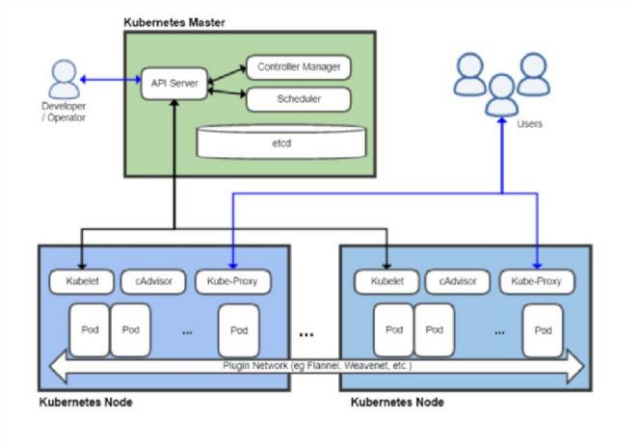
delve into key technologies utilized for deploying, managing, and orchestrating containers within clustered environments.

A. Docker:

At the forefront of containerization technology is Docker, a platform celebrated for its ability to encapsulate operating system processes into isolated environments with dedicated namespaces like. For developers, Docker offers a unified software environment that includes an application alongside its necessary library dependencies and configuration file.

Major cloud providers like Google Container Engine, Amazon Elastic Container Service (ECS), and Microsoft's Azure Container Service have contributed to the growing popularity of containerization through their products. Containerization is an extension of OS-level virtualization methods such as LXC (Linux Containers), and it is based on advances in operating system virtualization. Several separate Linux Virtual Environments (VE) can be created and managed on a single host by LXC by utilizing the cgroups capabilities of the Linux kernel. At the moment, Docker is the most used container technology. Its Docker Engine is the core program that hosts containers; it was first developed from LXC.

Deploying complex applications extends beyond simply activating individual containers. Embracing the microservices paradigm necessitates robust tools for automating the lifecycle management of potentially large collections of containers, a practice known as container orchestration. Leading this field is Kubernetes, an open-source platform explicitly designed to automate the deployment, scaling, and operational aspects of application containers across clusters of hosts. Kubernetes provides a container-centric infrastructure that simplifies the management of containerized applications. Serving as a comprehensive orchestration system for Docker containers, Kubernetes efficiently handles various workloads to ensure users' specified objectives are achieved. With Kubernetes, containers can be seamlessly created, terminated, restarted, and scaled up as required. Moreover, Kubernetes facilitates container deployment across diverse machines while establishing a cohesive communication network between them. A Kubernetes cluster comprises two primary types of nodes: worker nodes, responsible for executing containerized workloads, and control plane nodes.which manage the cluster's overall operation and orchestration.



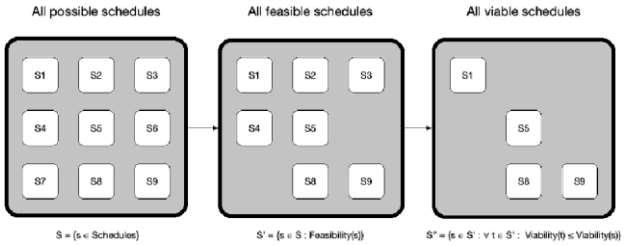
coordinating applications that are containerized. The main software components found in each kind of node. The Kubernetes Master, which is in the center of the cluster, is in charge of basic components that handle user requests and start worker node container activation. These essential parts are the key-value store, etcd, the Controller Manager (kube-controller-manager), the Scheduler (kube-scheduler), and the API Server (kube-apiserver). The main job of the Scheduler is to assign a cohesive group of connected containers, or Pods, to certain Kubernetes nodes. A pod is a collection of one or more containers that share network and storage resources, allowing for effective resource usage and communication within the cluster.

VI. KUBERNETES SCHEDULER

A key component of the Kubernetes design is the scheduler, which is in charge of keeping an eye on the object store and the API server to identify pods that users or controllers have produced but haven't yet allocated to a worker node. When the scheduler comes across these unassigned pods that don't have a designated nodeName, it assumes responsibility for allocating them to an appropriate node and then modifies the pod's nodeName parameter. After this assignment, the object store notifies the kubelet component that is deployed on the chosen worker node that the new pod is about to be executed. The kubelet starts the execution procedure by launching the pod on the selected node after getting this notification.

However, container scheduling presents an optimization challenge for the scheduler. Efficiently balancing resource utilization, workload distribution, and other factors requires sophisticated algorithms to ensure optimal deployment and performance across the Kubernetes cluster.

It must determine the most suitable node for executing the pod. This entails considering various factors such as resource availability, workload characteristics, and scheduling



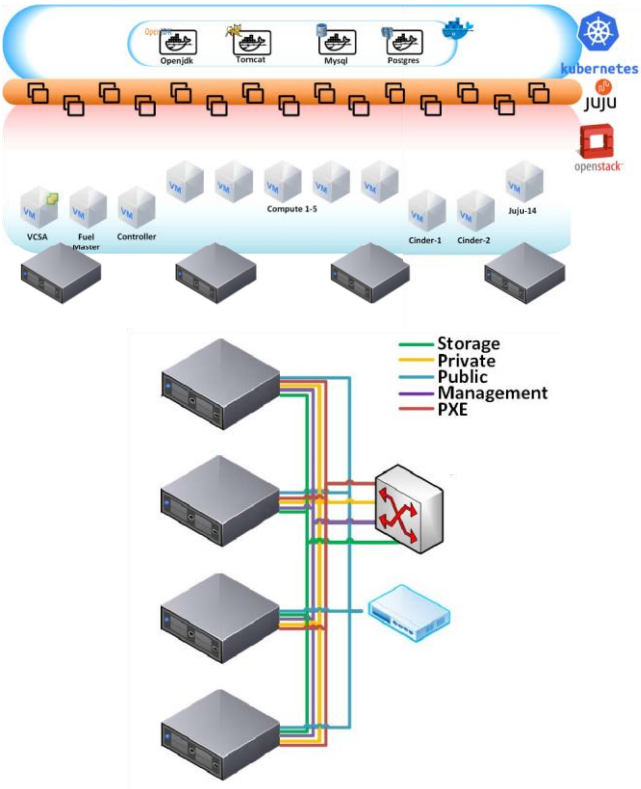
constraints. The scheduler's objective is to achieve efficient resource utilization, load balancing, and fault tolerance while meeting application requirements and constraints.

Decision regarding container placement constitutes a placement strategy. In this context, "partial" denotes the possibility of pods remaining unassigned, while "non-injective" suggests that multiple pods may be assigned to the same node, contingent on the scheduling algorithm utilized.

VII. Deployment of a Containerized Application

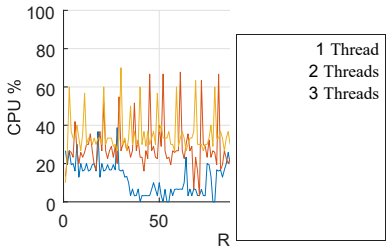
Deploying a containerized application involves a structured process to ensure seamless execution within containerized environments. Initially, the application is containerized by crafting a Docker image, encapsulating its dependencies, configurations, and startup commands within a Dockerfile. Subsequently, the Docker image is stored in a container image repository, such as Docker Hub or Google Container Registry, for centralized management and version control. To orchestrate deployment at scale and manage lifecycle operations, container orchestration platforms like Kubernetes or Docker Swarm are employed. These platforms automate tasks such as scheduling, scaling, and monitoring containers across a cluster of nodes. Deployment configurations, specified in YAML manifest files, detail parameters like Docker image references, replica counts, networking settings, and resource constraints. Upon applying these configurations to the orchestration platform, it orchestrates the deployment process by pulling the Docker image, scheduling pods, and distributing them across cluster nodes. Service discovery and load balancing mechanisms ensure accessibility To ensure effective distribution of incoming requests, monitoring, and logging tools are employed to track application health and performance, facilitating troubleshooting and optimization efforts. Moreover, integration with automation tools enables continuous deployment pipelines, automating deployments upon code changes. Through these measures, organizations can deploy containerized applications efficiently, harnessing the advantages of containerization and orchestration for scalability, reliability, and agility in deployment practices.

The hardware infrastructure comprises two Dell PowerEdge R630 servers, each equipped with dual Intel Xeon CPU E5-2640-v3 processors operating at 2.6 GHz, 64 GB of RAM, and two SAS internal hard disks. internal hard disks.



The Infrastructure-as-a-Service (IaaS) layer is set up on OpenStack in the testbed configuration, guaranteeing effective deployment and maintenance. The networking infrastructure in the OpenStack system is comprised of five discrete virtual networks, as follows: PXE Network, Public Network, Private Network, Storage Network, and Management Network. In addition, the testbed's Juju tool [18] facilitates the deployment, scaling, and lifecycle management of Kubernetes on top of OpenStack, guaranteeing the smooth orchestration of containerized apps in the Cloud.

Two Master Nodes and seven Worker Nodes in the Kubernetes cluster architecture were set up to satisfy the deployment specifications. We tested the containerized application's load after installing the M3 application inside of Kubernetes. For this, the application environment underwent basic load testing using Apache JMeter [19]. To represent the "number of users that JMeter will attempt to simulate," three different testing scenarios were developed, each with a gradual increase in the number of threads from one to three. All test scenarios involved sending HTTP requests to the application endpoints, hence a thorough analysis of performance under various load scenarios was possible.



According to the results, CPU usage in the virtual environment is not very sensitive to changes in the number of people logged in at once. The effectiveness of containerization, which efficiently divides the workload among several physical servers, is primarily responsible for this resiliency. Furthermore, the application exhibits automatic scalability within the suggested architecture as the workload grows. The Kubernetes Horizontal Pod Autoscaler [20], which dynamically modifies the number of containers based on their CPU consumption, makes this dynamic scaling possible. As a result, the tests emphasize how well the application is managed within the three-tier architecture, providing the necessary resources in an optimal way.

VIII. Performance Benchmarking

In this section, our aim is to explore the performance implications of various container allocations within a layered Cloud system. To achieve this goal, we have established a simplified experimental environment comprising three Kubernetes nodes deployed as virtual machines (VMs) distributed across two physically separate OpenStack compute nodes. Figure 7 offers a visual depiction of this experimental setup.

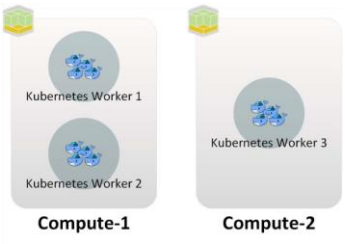


Fig. 7. Benchmarking environment

In our experimental setup, we conducted two distinct benchmarks across various scenarios to generate controlled and replicable workloads. The first benchmark focused on network performance and employed a client-server application structure. The second benchmark evaluated execution time, data transfer rate, and memory performance using the "dd" command-line utility.

A. Network Benchmarking:

In the second benchmark, we evaluated execution time, data transfer rate, and memory performance by incrementally increasing the number of concurrently active containers within a single Kubernetes worker. This setup enabled us to assess the impact of container allocation on overall performance metrics.

We measured network performance during the network benchmarking phase using iPerf3 [21]. iPerf3 is a tool that facilitates TCP throughput measurement between two endpoints by actively measuring the maximum bandwidth that may be achieved on IP networks. It consists of client and server parts. By default, the client connects to the server using TCP and initiates a data stream, sending data. At one-second intervals during the test, the sender and the recipient report the average bandwidth and the quantity of data sent and received. Every test was set up to run for ten seconds.

The average network bandwidth for the sender and receiver in the first case, when they were both operating within containers on the same virtual machine, was 12.9 Gb/s.In contrast, in the second and third scenarios, where Kubernetes worker nodes were deployed as VMs with 1 Gbps virtual NICs, the average network bandwidth for both sender and receiver decreased to approximately 0.8 Gb/s, with an average data transfer of about 1 GB. These results were attributed to the limitations imposed by the virtual NIC bandwidth and, in the third scenario, by the actual capacity of the server's physical NICs. Additionally, the third scenario exhibited more retransmissions compared to the second scenario.

The tests further expanded by incrementally increasing the number of clients from 1 to 5 for each scenario, yielding consistent results demonstrating the superior network bandwidth performance of the first scenario.

B. CPU-intensive and I/O-intensive workloads

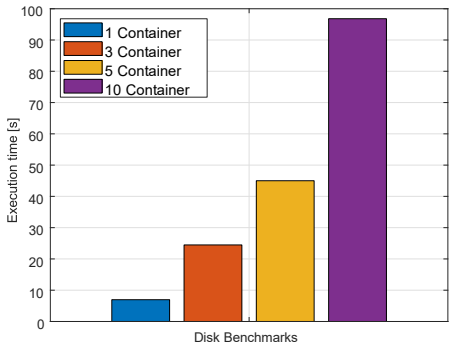
In the second benchmark campaign, we assess the performance of multiple containers coexisting on the same physical host, focusing on execution time, data transfer rate, and memory usage. This evaluation consists of two distinct benchmarks:

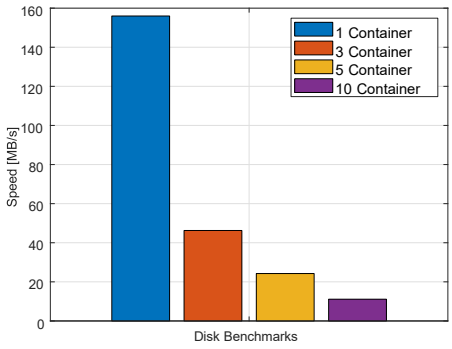
i) I/O-intensive workload: This benchmark entails executing the "dd" command, which retrieves a gigabyte of zeros from the Linux kernel and streams them into a file on the file system. We measure the average execution time, data transfer rate during write operations to the file system, and memory consumption.

ii) CPU-intensive workload: In this benchmark, we analyze the performance of CPU-intensive tasks executed by the containers.

For the "dd" command benchmark, we systematically initiate, gradually scaling up to 10 containers. Through this

iterative process, we aim to observe how the performance metrics evolve in response to the growing container workload.



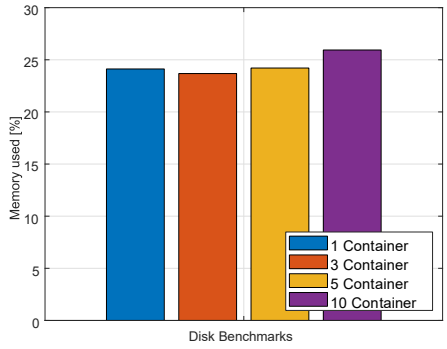


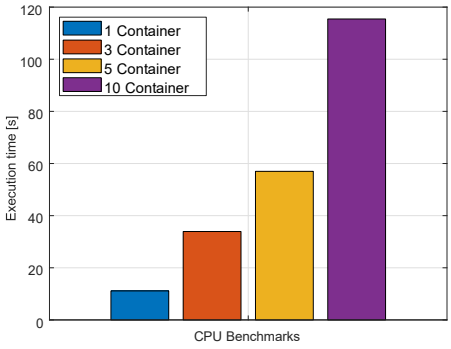
IX. Conclusion

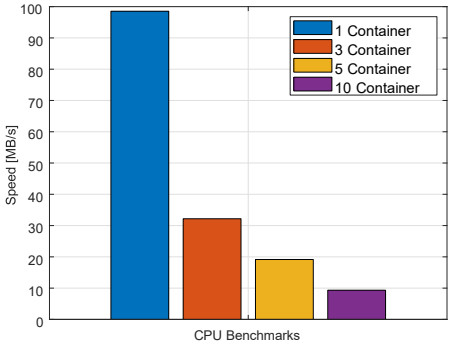
The decision to migrate is driven by various factors. In contemporary times, this transition, known as cloudification, is predominantly undertaken due to several reasons.

In this paper, we delve into the process of decomposing the original monolithic application into smaller components, which can be instantiated separately as containers. We explore both the advantages and risks associated with combining containers with virtual machines (VMs) in a virtualized private infrastructure. While containerization presents clear benefits in terms of flexibility, we also investigate the potential advantages in scalability and efficiency of resource utilization. Our study aims to provide insights into the integration of containers and VMs and their respective impacts on application deployment and management.

In this paper, we explore the process of breaking down the original monolithic application into smaller components, which can be instantiated separately as containers. We thoroughly examine the advantages and risks associated with combining containers with virtual machines (VMs) in a virtualized private infrastructure







In this paper, we explore the process of breaking down the original monolithic application into smaller components, which can be instantiated separately as containers. We thoroughly examine the advantages and risks associated with combining containers with virtual machines (VMs) in a virtualized private infrastructure. While containerization provides clear benefits in terms of advantages in scalability and efficiency of resource utilization. Our findings demonstrate that while a well-designed combination of VMs and containers offers maximum flexibility, it is crucial to consider how these two virtualization layers interact, as a simplistic approach may lead to suboptimal outcomes.

ACKNOWLEDGEMENT

We acknowledge the partial support provided by Cisco Systems through the Sponsored Research Agreement titled "Research Project for Industry 4.0".

REFERENCES

[1] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing—the business perspective," *Decision support systems*, vol. 51, no. 1, pp. 176–189, 2011.

[2] R. LeFebvre, "Why openstack and kubernetes are better together," https://superuser.openstack.org/articles/openstack kubernetes better together/, last accessed: 5 Feb 2020.

[3] "Openstack homepage," https://www.openstack.org/, last accessed: 5 Feb 2020.

[4] J. Lewis and M. Fowler, "Microservices – a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, last accessed: 5 Feb 2020.

[5] A. Simioni and T. Vardanega, "In pursuit of architectural agility: Experimenting with microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018, pp. 113–120.

[6] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, and B.-J. Kim, "Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud," *Advanced Science and Technology Letters*, vol. 66, no. 12, pp. 105–111, 2014.

[7] E. F. Boza, C. L. Abad, S. P. Narayanan, B. Balasubramanian, and M. Jang, "A case for performance-aware deployment of containers," in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, ser. WOC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 25–30. [Online]. Available: https://doi.org/10.1145/3366615.3368355

[8] Z. Usmani and S. Singh, "A survey of virtual machine placement techniques in a cloud data center," *Procedia Computer Science*, vol. 78, pp. 491 – 498, 2016, 1st International Conference on Information Security and Privacy 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050916000958

[9] G. B. Fioccola, P. Donadio, R. Canonico, and G. Ventre, "Dynamic routing and virtual machine consolidation in green clouds," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2016, pp. 590–595.

[10] P. D. Bharathi, P. Prakash, and M. V. K. Kiran, "Virtual machine placement strategies in cloud computing," in *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, April 2017, pp. 1–7.

[11] R. Zhang, A.-m. Zhong, B. Dong, F. Tian, and R. Li, *Container-VMPM Architecture: A Novel Architecture for Docker Container Placement*. Springer International Publishing, 06 2018, pp. 128–140.

[12] A. Khan, "Key characteristics of a container orchestration platform to enable a modern application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, September 2017.

[13] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: a state-of-the-art review," *IEEE Transactions on Cloud Computing*, 2017.

[14] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[15] "Docker homepage," https://www.docker.com/, last accessed: 5 Feb 2020.

[16] "Kubernetes homepage," http://kubernetes.io, last accessed: 5 Feb 2020.

[17] "Fuel mirantis documentation homepage," https://docs.mirantis.com/fuel-docs/mitaka/userdocs/fuel-userguide.htmll, last accessed: 5 Feb 2020.

[18] "Juju documentation homepage," https://jaas.ai/docs/getting-startedwith-juju, last accessed: 5 Feb 2020.

[19] "Apache jmeter homepage," https://jmeter.apache.org/, last accessed: 5 Feb 2020.

[20] B. Hofmann and S. Pearce, "Auto scaling kubernetes clusters on openstack," https://www.openstack.org/summit/berlin-2018/summitschedule/events/22884/auto-scaling-kubernetes-clusters-on-openstack, last accessed: 5 Feb 2020.

[21] "iPerf - The ultimate speed test tool for TCP, UDP and SCTP," https://iperf.fr/, last accessed: 5 Feb 2020.