Assessing the Performance of Docker in Docker Containers for Microservice-based Architectures

Felipe Bedinotto Fava, Luiz Felipe Laviola Leite, Luís Fernando Alves da Silva,
Pedro Ramires da Silva Amalfi Costa, Angelo Gaspar Diniz Nogueira, Amanda Fagundes Gobus Lopes,
Claudio Schepke, Diego Luis Kreutz, Rodrigo Brandão Mansilha
Graduate Software Engineering (PPGES) – Laboratory of Advances Studies in Computation (LEA),
Federal University of Pampa (UNIPAMPA), Alegrete, Brazil.
Email: {felipefava,luizlaviola,luisfads,pedroamalfi,angelonogueira,amandalopes}.aluno@unipampa.edu.br,
{claudioschepke,diegokreutz,rodrigomansilha}@unipampa.edu.br

Abstract—We provide a comprehensive and updated assessment of Docker versus Docker in Docker (DinD), evaluating its impact on CPU, memory, disk, and network. Using different workloads, we evaluate DinD's performance across distinct hardware platforms and GNU/Linux distributions on cloud Infrastructure as a Service (IaaS) platforms like Google Compute Engine (GCE) and traditional server-based environments. We developed an automated tools suite to achieve our goal. We execute four well-known benchmarks on Docker and its nested-container variant. Our findings indicate that nested-containers require up to 7 seconds for startup, while the Docker standard containers require less than 0.5 seconds for Debian and Alpine operating systems. Our results suggest that Docker containers based on Debian consistently outperform their Alpine counterparts, showing lower CPU latency. A key distinction among these Docker images lies in the varying number of installed libraries (e.g., stretching from 13 to 119) across different Linux distributions for the same system (e.g., MySQL). Furthermore, the number of events and CPU latency indicates that the influence of DinD over Docker proves that it is insignificant for both operating systems. In terms of memory, running containers of Debian-based images consume 20% more size of memory than those based on Alpine. No significant differences are between nested-containers and Dockers for disk and network IO. It is worth emphasizing that some of the disparities, such as a bigger memory footprint, appear to be a direct result of the software stack in use, including different kernel versions, libraries, and other essential packages.

Index Terms—Docker in Docker, DinD, Docker Containers, Nested-container, Performance Evaluation, Benchmarks, Cloud Computing

I. Introduction

Containers represent a paradigm of isolation without compromising speed and efficiency. Remarkably lightweight, a single x86 server can seamlessly host hundreds of containers, with memory often becoming the primary constraint. Notably, containers have fast startup times, typically taking under 1 to 2 seconds to initiate, ensuring swift deployment.

Several factors to the resurgence of containers are attributed, but from a technical standpoint, two pivotal reasons stand out. Firstly, enhancements in namespace support within the Linux kernel, now prevalent in popular distributions, have played a fundamental role. Secondly, a specific container implementation has revolutionized the landscape. Docker not only introduced an appealing packaging format, but also provided indispensable tools and fostered a diverse ecosystem, making it a cornerstone of this container renaissance.

A Docker container image stands like a streamlined, self-contained software package, encapsulating everything essential for running an application: code, runtime, system tools, libraries, and configurations. When deployed, these container images transform into active containers, a transformation facilitated seamlessly on Docker Engine for Docker containers. Compatible with Linux and Windows applications, containerized software guarantees consistent operation regardless of the underlying infrastructure. By isolating software from its surroundings, containers ensure uniform functionality across different environments, such as development and staging.

Docker containers offer advantages such as standardization, lightweight efficiency, and security. While standardization enables portability across diverse environments, lightweight efficiency eliminates the need for a separate OS per application, enhancing server efficiency and reducing server and licensing expenses. Finally, Docker provides robust default isolation capabilities, ensuring enhanced security within containers for applications.

Docker has emerged as a pivotal technology in development and production environments, particularly for implementing and deploying microservice-based architectures [1]–[3]. However, its is underscored of prevalence by extensive adoption across diverse domains and applications. This versatile platform has become essential in modern software engineering, facilitating seamless deployment and microservices management. Its impact resonates across a spectrum of industries, demonstrating its adaptability and effectiveness in meeting the complex demands of contemporary software development.

There are two prevailing models for container-based implementation within the microservices framework: the master-slave model and the nested-container model [4]. The master-slave model comprises a master container orches-

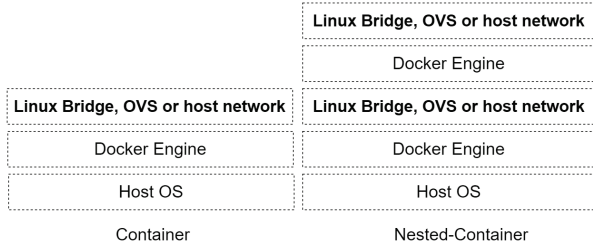|	
	Linux Bridge, OVS or host network
	Docker Engine
Linux Bridge, OVS or host network	**Linux Bridge, OVS or host network**
Docker Engine	Docker Engine
Host OS	Host OS
Container	Nested-Container

Fig. 1. Two layers of Docker daemon (adapted from [4], Figure 2)

trating others as slaves, where the application processes operate. The master is responsible for monitoring the subordinate containers and facilitating their communication. In contrast, the nested-container approach involves hierarchically creating subordinate containers (children) within a main container (parent). The parent container can be agnostic and merely exist, while the children run the application processes, confined within the boundaries set by the parent. The DinD approach simplifies the management in microservices architectures, reduces the number of steps for reproducing experiments, and isolates the development and testing environments, enhancing security and isolation, implementing of CI/CD pipelines, and resource management in multi-tenancy scenarios where multiple teams or users require isolated environments on shared infrastructure [5], [6]. DinD facilitates Inter-Process Communication (IPC), ensures fate sharing, and enables shared memory, disk, and network resources. However, this approach may introduce additional due to its two layers of Docker daemon, as illustrated in Figure 1.

In [4], the authors note five key points. First, CPU-intensive tasks in containers not have significant performance difference compared to bare-metal setups. Second, regular containers are the fastest, followed by nested-containers and virtual machines during instance creation. Creating nested-containers involves more overhead but is still faster than virtual machines. Third, creating multiple parents with a single child takes longer due to parent container initialization bottlenecks. Lastly, varying network configurations of nested containers (host network, Linux bridge, and Open vSwitch) show no significant performance impact once the physical network becomes the bottleneck. In short, the authors provide a good overview of the main challenges in DinD environments back to 2015. However, the landscape of Docker technology has evolved substantially from 2015 to 2024. Furthermore, in [4], the authors did not explore workloads encompassing memory, disk, and network IO. Their evaluation focused solely on CPU-intensive tasks with SysBench and a client-server tool for network IO.

We provide an updated (2024) and more comprehensive assessment of Docker versus DinD, evaluating its impact on CPU, memory, disk, and network. Additionally and beyond

past works such as [4], we assess DinD performance across different hardware platforms and GNU/Linux distributions using established benchmarks like SysBench[1], Stress[2], IOzone[3], and iPerf[4] [7]–[9]. Furthermore, we explore the impact of DinD on cloud Infrastructure as a Service (IaaS) platforms like Google Compute Engine (GCE) and traditional local server environments.

Our contributions is three-fold. **Automated Scripts:** We provide a collection of automated scripts for building container images and running the benchmarks in Docker and DinD across multiple GNU/Linux distributions. These scripts also provide automated statistics. It makes replication of our experiments easy, allows fast inclusion and testing of new GNU/Linux distributions, and automatic execution. **Evaluation and Analysis:** We meticulously evaluate the impact of Docker and DinD using cutting-edge GNU/Linux distributions and the latest Docker Engine versions. Our analysis covers CPU processing, memory size, and I/O (network and disk) latency and throughput. This analysis is invaluable for microservice architectures and other scenarios running aplications in DinD mode. **Comprehensive Data Repository:** We offer a repository housing a wealth of data from our experiments. This extensive dataset enables further analyses beyond the scope of our paper, empowering researchers and practitioners to delve deeper into the nuances of container performance.

Our GitHub repository [10] serves as a comprehensive resource, housing all automation scripts essential for reproducibility. Within this repository, you'll find an extensive collection of data comprising: (a) a detailed record of all benchmark execution outcomes; (b) in-depth technical details about each hosting system involved in the study; (c) comprehensive lists of all packages and libraries installed in both Docker images (two per benchmark) and the hosting systems; (d) a curated collection of useful tools designed for testing and executing all benchmarks. We tried to ensure transparency and facilitate the replication of our work.

The remainder of the paper does organize as follows. Section II describes the methodology of the experiments. We characterize the benchmarks, metrics, and execution environments. Section III shows the experimental results for CPU, Memory, Disk, and Network, considering four distinct benchmarks on five different clouds' infrastructure. Section IV presents the conclusion and future works.

II. METHODOLOGY

We present an overview of our methodology in Figure 2. To evaluate the performance of four crucial computer components (CPU, memory, disk, and network) across two container environments (Docker and Docker in Docker) and two GNU/Linux distributions (Debian and Alpine), we conducted four benchmarks in five distinct hardware

[1]https://github.com/akopytov/sysbench
[2]https://github.com/resurrecting-open-source-projects/stress
[3]https://www.iozone.org/
[4]https://iperf.fr

138

setups. Each benchmark run 20 times, resulting in a total of 1600 trials. Table I summarizes the key settings. For a detailed description and configurations of the experiments, we have made the information accessible on GitHub [10].

Our execution environment comprised three nodes from Google Cloud Platform (GCP): C3, N2, and E2, and two local nodes, A9 (Intel i7) and W1 (AMD Ryzen). GCP nodes C3 (type c3-standard-4), N2 (type n2-standard-2), and E2 (type e2-medium) feature 2, 1, and 1 cores. Local nodes A9 and W1 are equipped with 4 and 8 physical cores.

We selected benchmarks widely recognized in both practitioner and researcher communities to assess hardware aspects [7]–[9], [11]–[14]. These benchmarks evaluate processing, memory, and I/O performance, encompassing both network and disk operations. In the following, we elaborate on specific benchmark tools chosen for our evaluation.

SysBench: A versatile benchmarking tool widely utilized on Unix-like systems, SysBench evaluates system performance in areas like CPU, memory, threads, disk I/O, and databases. This tool aids system administrators and developers in testing and analyzing system and application performance across diverse scenarios.

Stress: An essential tool designed to apply controlled tension on a system, Stress can simulate varying degrees of CPU, memory, I/O (synchronization), or disk stress. The tool is commonly used for kernel programmers, enabling them to assess the system's scalability and scrutinize perceived performance features. Additionally, it aids systems programmers in unveiling specific classes of bugs that often surface only under heavy loads, offering critical insights into system performance under stress conditions.

IOzone: IOzone is a versatile benchmark utility that generates and evaluates a diversity of range of file operations. It has been ported to numerous machines and is compatible with multiple operating systems. IOzone proves invaluable for conducting comprehensive file system analyses on various computer platforms. The benchmark rigorously assesses file I/O performance across a spectrum of operations, including read, write, reread, rewrite, read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read, and aio_write.

iPerf: This open-source tool is indispensable for measuring network performance in IP-based computer networks. iPerf excels in conducting essential assessments, including bandwidth testing, latency, packet loss, and more. iPerf designates one device as the server, which actively listens for connections, operating on a client-server model. Another device initialize the connections, working as the client, to perform thorough performance tests.

III. Experimental Results

We discuss the main results regarding CPU, memory, disk, and network components in its respective subsections. We present a series of bar plots obtained from multiple benchmarks for each hardware element. Each plot contains 20 bars, organized in 5 groups, one per hardware setup.

Each group has 4 bars from combining two OS (Alpine filled in blue and Debian filled in orange) and two container models (DinD filled with darker colors and Docker filled with lighter colors).

A. CPU

In Figure 3, we present the SysBench CPU usage results. We take into account both the number of events and latency. By observing the CPU technology and operation frequency (see Table I), we find that the number of CPU events correlates directly with CPU capacity, as shown in Figure 3a. In particular, our hosting system W1 demonstrates the highest performance, whereas N2 exhibits the lowest. Figure 3b illustrates an almost inverse relationship, i.e., the best CPU displays the lowest latency.

Our findings indicate that Docker containers based on Debian consistently outperform their Alpine counterparts, showing lower CPU latency. Recent studies have also highlighted performance disparities between Docker images built on different Linux distributions for various systems such as databases (e.g., Cassandra, Mongo, MySQL) and web servers (e.g., Nginx) [15]. A key distinction among these Docker images lies in the varying number of installed libraries (e.g., stretching from 13 to 119) across different Linux distributions for the same system (e.g., MySQL). Furthermore, the number of events and CPU latency shown in Figure 3 indicate that the influence of DinD over Docker proves that it is insignificant for both operating systems.

In Figure 4, we present the data on the number of events and percentiles of SysBench results for the thread subsystem test. Notably, Docker instances exhibit a higher count of events with a lower latency. The higher latency of DinD remains consistent across diverse hosting systems. It's important to note that this latency varies among the machines. For instance, in scenarios like W1, the DinD latency can surpass Docker's latency by more than 10%.

When examining the execution behavior, as depicted in Figure 5, a noticeable bootstrap latency becomes evident in DinD instances. For instance, while the Stress benchmark initiates its execution immediately in Alpine Docker, it can take more than 20 seconds in an N2 hosting system when using DinD. Our additional results [10] indicate that this latency pattern persists across different benchmarks and execution environments, as well observed consistently in Debian instances. Notably, the DinD bootstrap latency can vary significantly from one machine to another. While it might require more time in an N2 system, it exhibits faster performance in a W1 hosting system.

To further investigate the bootstrap latency shown in Figure 5, we analyze the startup and shutdown times of DinD containers in Figure 6. While Docker containers initiate in under 500ms, DinD averages up to 6 seconds to start. That means that DinD can be more than 12 times slower than Docker containers in becoming operational for running applications. While Docker containers terminate in less than 100ms, DinD may take over 1.5 seconds to
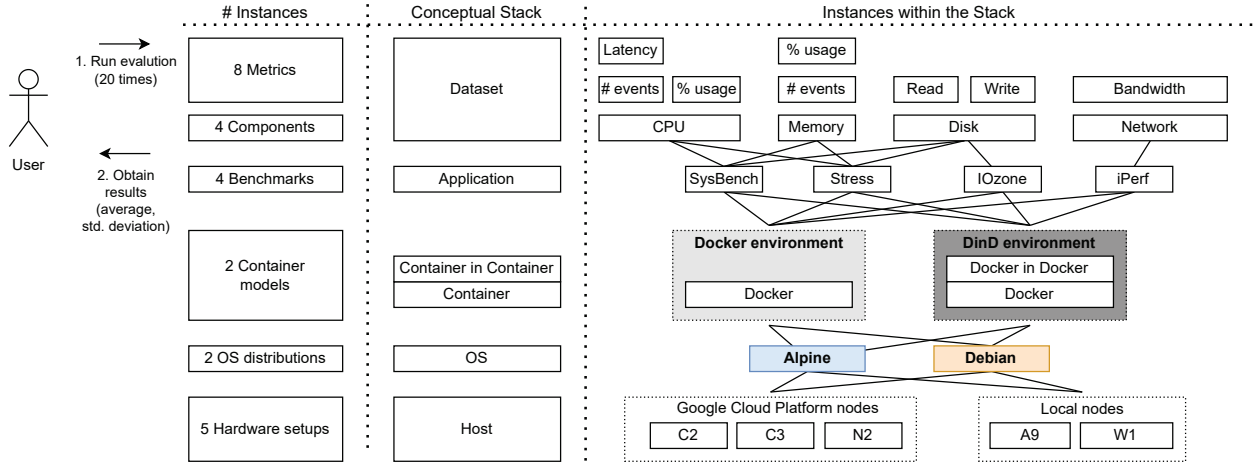
Fig. 2. Methodology Overview

TABLE I
Benchmark, container, and hosting systems settings

| Setting	Value(s)
SysBench	Employ default batteries of tests, including threads, memory, CPU, and FILEIO.
Stress	One test with a timeout of 100s for each CPU, HDD, I/O, and VM parameter.
IOzone	Utilization of 1 GB of data for reading and writing using 4 KB blocks.
iPerf	Use of the standard client-server (local) parameters to measure throughput and latency.
Container	Docker and Docker in Docker (DinD)
Container OS	Alpine 3.18 and Debian 12.1
C2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 3.10GHz, 32GB NVMe disk, 16GiB DIMM RAM
C3 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon Platinum 8481C, 32GB NVMe disk, 16GiB DIMM RAM
N2 (GCP)	Debian 12, Docker Engine 24.0.7, Intel Xeon 2.80GHz, 32GB PersistentDisk, 8GiB DIMM RAM
A9 (Local)	Debian 11, Docker Engine 24.0.7, Intel i7-9700 3.00GHz, 1TB WDC WD10EZEX-08W, 16GiB DIMM DDR4
W1 (Local)	Ubuntu 22.04.3 LTS, Docker Engine 24.0.5, AMD Ryzen 7 5800X, 512GB NVMe disk, 64GiB DIMM DDR4

(a) Events

(b) Latency
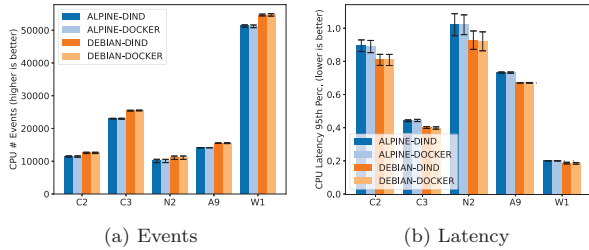
Fig. 3. SysBench CPU statistics

(a) DinD

(b) Docker
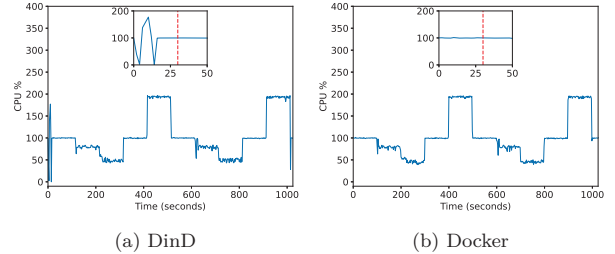
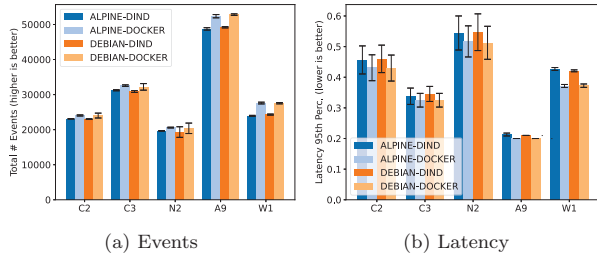Fig. 5. Stress CPU behavior for Alpine on N2

(a) Events

(b) Latency

Fig. 4. SysBench threads subsystem

stop completely. It is also worth noting that there is no significant difference in startup and shutdown times between Alpine- and Debian-based DinD containers. Interestingly, the A9 system exhibits a slightly slower startup and shutdown time despite its Intel i7-9700 CPU clocked at 3GHz. We attribute the high standard deviation to the inherent non-deterministic nature of container shutdown times. However, it's important to note that the software stack in usage could potentially influence this variance. Unlike the other hosting systems which operate on Debian 12 or Ubuntu 22.04, the A9 system runs on Debian 11, as
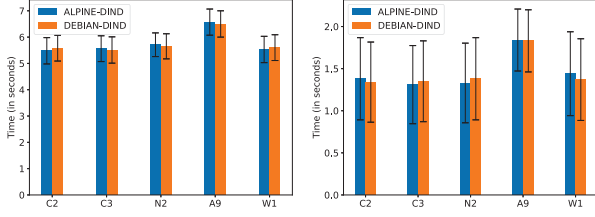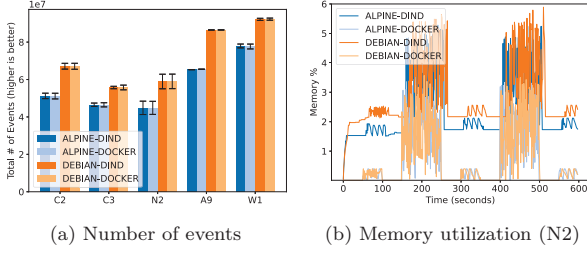
(a) DinD start   (b) DinD stop

Fig. 6. Required time to start and stop DinD containers
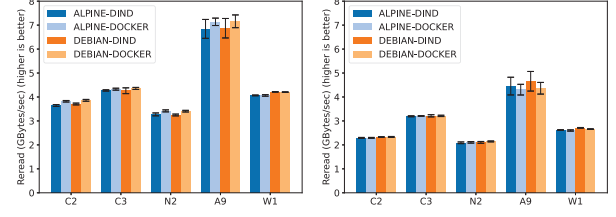
(a) Reread   (b) Rewrite

Fig. 8. Disk operations by IOzone

(a) Number of events   (b) Memory utilization (N2)

Fig. 7. SysBench (a) and Stress (b) memory consumption

indicated in Table I. This distinction in operating systems may contribute to the observed disparities in performance.

B. Memory

In Figure 7a, we present the memory event counts for SysBench and a memory utilization sample for Stress. Notably, it becomes evident that the total number of memory events remains consistent across DinD and Docker setups employing the same Linux distribution. We attribute this consistency to SysBench's uniform data collection methodology in both environments. However, there is a notable disparity in memory event counts when comparing Debian and Alpine distributions. In particular, the Debian-based Docker image incurs a significant overhead, surpassing 20% of memory size across all executing environments. This discrepancy underscores the impact of the underlying Linux distribution on memory event generation.

Additionally, we can make at least three observations in Figure 7b. Firstly, it is evident that the overall DinD's memory footprint is significantly bigger than Docker across both Linux distributions. This measurement encompasses the benchmark's inherent memory consumption as the additional memory overhead introduced by DinD. It is worth emphasizing that these gather memory utilization samples were by using the live data stream provided by `docker stats` for a specific running container.

Secondly, a visible difference emerges between Debian-based DinD (depicted by the yellow line) and Alpine-based DinD (represented by the blue line). Debian DinD consistently exhibits significantly higher memory consumption than its Alpine counterpart (in the plot, it is easy to view

considering the DinD pair). Once again, this variance underscores that Docker images based on Debian can demand 20% more memory than those built on Alpine.

Furthermore, Figure 7b reveals a cyclic pattern in the benchmark's behavior. This cyclical occurrence is usual across all programs when capturing memory utilization statistics using `docker stats`.

We attribute the disparities illustrated here to differences in Docker images rooted in distinct Linux distributions [15]. Depending on the specific distribution-specific libraries used to support and run the benchmarks, the memory footprint can vary significantly, leading to the observed discrepancies in memory consumption.

C. Disk

We show the IOzone benchmark results in GB/s in Figure 8. We present our findings for run options reread and rewrite. We observed minimal differences among the four approaches, for both DinD and Docker using Alpine and Debian, across our hosting systems. Generally, reread operations exhibit a slight advantage in Docker containers compared to DinD. In specific environments such as A9 and W1, distinctions between Docker containers and DinD may fluctuate for short-term disk-intensive benchmarks or applications. We anticipate a more comprehensive exploration of these differences through prolonged and robust I/O benchmarks as the evaluation of workloads.

While C3, C2, and N2 use Google PersistentDisk with specifications such as revision 1, SPC-4 compliance, and a Solid State Device rotation rate, W1 achieves superior performance using SSD disks (model: SM2P32A8-512GC1, Firmware Version: VC0S032V, NVMe Version: 1.4). A9 should also achieve higher performance using its high-speed SSD disks (model: SKHynix_HFS256GD9TNI-L2B0B, firmware version: 11710C10, NVMe Version: 1.3). However, the docker stack in A9 throttles down the disk I/O throughput from top 62 MB/s to between 10 and 15 MB/s.

D. Network

In Figure 9, we present the network performance execution results for iPerf with measurements expressed in Gbits/s. It is evident from the data that there are negligible performance variations between DinD and Docker in nodes C2, C3, and N2, all hosted on the Google Cloud Platform.
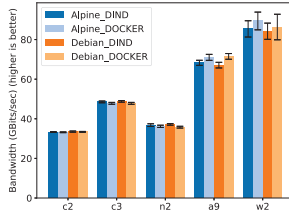
Fig. 9. Network bandwidth by iPerf

However, in local hosting systems, A9 and W1, Docker containers exhibit up to 7% higher performance in relation to DinD. One of the key factors contributing to this discrepancy is the use of different versions of Linux distributions and operating systems (i.e., kernel versions). While all GCP nodes employ Debian 12, A9 employs Debian 11, and W1 operates on Ubuntu 22.04. Consequently, variations in the installed versions of packages and libraries contribute to the observed performance difference.

IV. CONCLUSION

In this paper, we assess the performance of nested-Docker containers (DinD) within a context of microservice-based architectures. To systematically evaluate the capabilities of DinD in a range of hosting environments, we developed automated scripts designed for executing well-established CPU, memory, disk, and network I/O benchmarks, namely Sysbench, Stress, IOZone, and iPerf. We deployed these benchmarks within Docker images built on Debian and Alpine Linux distributions. Finally, we collected the benchmark statistics from three Google Computing Platform nodes and two local nodes.

Our primary findings reveal non-negligible differences in memory consumption between Docker and DinD and among distinct Linux distributions. Additionally, DinD containers have a startup time of up to 7 seconds. These factors, namely memory consumption and startup time, emerge as potential constraints in microservice architectures operating under time or resource constraints. For instance, a 7-second startup time may be deemed unacceptable for low-latency and short-lived microservices.

To facilitate transparency and reproducibility, all our scripts and extensive collected data are available in our GitHub repository [10]. This repository ensures the reproducibility of our experiments, and also used as a valuable resource for future extended research initiatives.

In our future work, we intend (a) to delve into the specific impact of DinD in application-centric scenarios and workloads; (b) to incorporate a broader spectrum of Linux distributions in our Docker images; (c) to explore other cloud infrastructures, investigating the potential variability in disk I/O based on the underlying cloud abstraction file system; and (d) to investigate the implications of different Docker Engine versions on performance outcomes.

REFERENCES

[1] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann, "Microservices in industry: Insights into technologies, characteristics, and software quality," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2019, pp. 187–195.

[2] F. Tapia, M. Ángel Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From Monolithic Systems to Microservices: A Comparative Study of Performance," *Applied Sciences*, vol. 10, no. 17, 2020.

[3] I. Karabey Aksakalli, T. Çelik, A. B. Can, and B. Tekinerdoğan, "Deployment and communication patterns in microservice architectures: A systematic literature review," *Journal of Systems and Software*, vol. 180, p. 111014, 2021.

[4] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 ieee 14th international symposium on network computing and applications*. IEEE, 2015, pp. 27–34.

[5] S. Kushwah, "Docker inside docker," 2023, https://medium.com/@shivam77kushwah/docker-inside-docker-e0483c51cc2c.

[6] J. Petazzoni, "Using docker-in-docker for your ci or testing environment? think twice." 2023, https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/.

[7] C. O. Diaz, J. E. Pecero, P. Bouvry, G. Sotelo, M. Villamizar, and H. Castro, "Performance evaluation of an iaas opportunistic cloud computing," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2014, pp. 546–547.

[8] N. G. Bachiega, P. S. L. de Souza, S. M. Bruschi, and S. d. R. S. de Souza, "Performance evaluation of container's shared volumes," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2020, pp. 114–123.

[9] A. Pokharana and R. Gupta, "Using sysbench, analyze the performance of various guest virtual machines on a virtual box hypervisor," in *2023 2nd International Conference for Innovation in Technology (INOCON)*, 2023, pp. 1–5.

[10] F. B. Fava, L. F. L. Leite, L. F. A. da Silva, P. R. da Silva Amalfi Costa, A. G. D. Nogueira, A. F. G. Lopes, C. Schepke, D. L. Kreutz, and R. B. Mansilha, "Assessing the Performance of Docker in Docker Containers for Microservice-based Architectures Benchmark," 2024, https://github.com/CNuvem23/PDP2024.

[11] UpCloud, "Evaluating cloud server performance with sysbench," 2018, https://upcloud.com/blog/evaluating-cloud-server-performance-with-sysbench.

[12] C. Qian, "Testing i/o performance with sysbench," 2019, alibaba Cloud ECS. https://www.alibabacloud.com/blog/testing-io-performance-with-sysbench_594709.

[13] Dell Technologies, "How to test available network bandwidth using 'iperf'," 2021, alibaba Cloud ECS. https://www.dell.com/support/kbdoc/en-us/000139427/how-to-test-available-network-bandwidth-using-iperf.

[14] M. G. Xavier, K. J. Matteussi, F. Lorenzo, and C. A. De Rose, "Understanding performance interference in multi-tenant cloud databases and web applications," in *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 2016, pp. 2847–2852.

[15] M. H. Ibrahim, M. Sayagh, and A. E. Hassan, "Too many images on dockerhub! how different are images for the same system?" *Empirical Software Engineering*, vol. 25, pp. 4250–4281, 2020.