The Journal of Systems and Software 210 (2024) 111965

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A qualitative and quantitative analysis of container engines ☆

Luciano Baresi, Giovanni Quattrocchi *, Nicholas Rasi

Dipartimento di Elettronica, Informazione e Bioingegneria, Politecnico di MIlano, Milan, Italy

ARTICLE INFO

Keywords:
Containerization
Container engines
Performance
Cloud computing
Internet of things
High-performance computing

ABSTRACT

Containerization is a virtualization technique that allows one to create and run executables consistently on any infrastructure. Compared to virtual machines, containers are lighter since they do not bundle a (guest) operating system but they share its kernel, and they only include the files, libraries, and dependencies that are required to properly execute a process. In the past few years, multiple container engines (i.e., tools for configuring, executing, and managing containers) have been developed ranging from some that are "general purpose", and mostly employed for Cloud executions, to others that are built for specific contexts, namely Internet of Things and High-Performance Computing. Given the importance of this technology for many practitioners and researchers, this paper analyzes six state-of-the-art container engines and compares them through a comprehensive study of their characteristics and performance. The results are organized around 10 findings that aim to help the readers understand the differences among the technologies and help them choose the best approach for their needs.

Editor's note: Open Science material was validated by the Journal of Systems and Software Open Science Board.

1. Introduction

Containerization (Merkel, 2014) is a widely adopted virtualization technology that allows one to create lightweight executables that can run consistently on any infrastructure. They exploit a shared operating system (OS) and package executables along with their required dependencies (e.g., code, configuration files, and libraries) in a standard format. They are lighter, faster to boot, and scale better than virtual machines (VM) because they share a host OS kernel and only virtualize userspaces (Soltesz et al., 2007; Baresi and Quattrocchi, 2020). Originally conceived for cloud-native applications (Pahl, 2015), they are now the de-facto compute unit in many contexts—for example, Internet of Things (IoT) (Celesti et al., 2016) and High-Performance Computing (HPC) (Higgins et al., 2015) solutions. Containers mainly exploit the features of the Linux kernel, but there are also a few implementations for other operating systems (Microsoft, 2022).

A *container engine* supplies a set of tools to create and manage containers. It provides means to create *container images*, that is, container blueprints that embed all the necessary files needed at runtime: a *container* (or *container instance*) is then a standard process that instantiates a container image. Multiple containers can be created from a single image. Images are usually defined through dedicated languages (e.g., Dockerfile (Docker, 2022c)) and built using *image builders*. The *container runtime* allows one to start and manage containers from images, configure the kernel, and initialize the process that the container wraps. Container engines are usually connected to a public (or private) *registry*, where users can upload or download container images, and can also provide *orchestration tools* to schedule, connect, and oversee the execution of multiple containers running on a cluster of physical or virtual machines. In the last few years, numerous container engines have been presented and adopted by practitioners. Each approach is dedicated to an execution environment and provides different features, security guarantees, and performance.

This paper aims to shed some light on these technologies and help the readers understand the technical differences among the approaches and choose the best technology for their needs. Given the diverse engines that exist today, we selected three reference domains: cloud computing, IoT (Internet of Things) solutions, and HPC (High Performance Computing) systems, to categorize the different engines, understand the particular requirements, and emphasize the specificity of studied engines. In the domain of cloud computing, agility and operational efficiency are key; container technologies facilitate these by accelerating release cycles and reducing operational complexities, particularly in the context of highly modularized applications (Pahl et al., 2019). HPC systems are characterized by specialized hardware, the need for complex library dependencies like MPI, and stringent

☆ Editor: Uwe Zdun.

* Corresponding author.
E-mail addresses: luciano.baresi@polimi.it (L. Baresi), giovanni.quattrocchi@polimi.it (G. Quattrocchi), nicholas.rasi@polimi.it (N. Rasi).

https://doi.org/10.1016/j.jss.2024.111965
Received 2 March 2023; Received in revised form 1 October 2023; Accepted 3 January 2024
Available online 5 January 2024

security protocols in multi-tenant settings; container technologies serve to ease these complexities by simplifying dependency management, facilitating GPU utilization, and enhancing security (Sampedro et al., 2018). In IoT environments, a variety of hardware architectures and limited resources like memory and storage make deployment challenging (Balaji et al., 2019). Containers offer a solution by enabling easier management of devices, optimizing resource use, and addressing privacy concerns (Botez et al., 2020).

For each domain, we retrieved the two most famous container engines, according to GitHub. We then analyzed Docker (Docker, 2022b), which played a key role to made containers become popular, and Podman (Podman, 2022) as cloud-native solutions, Charliecloud (Charliecloud, 2022) and Singularity (Sylabs, 2022a) as HPC-specific engines, and we only found balenaEngine (Balena, 2022b) as container engine for IoT systems. We decided to also add Sarus (Sarus, 2022b), as third HPC-specific engine since it is a promising solution and it is the only one that is fully developed in academia.

The proposed analysis is both qualitative, since we confront offered features, and quantitative, since we compared their performance on a comprehensive set of benchmarks. Obtained results are summarized in 10 findings that highlight the characteristics, similarities, and differences of considered engines.

The rest of this paper is organized as follows. Section 2 describes the selection process and the container engines we selected. Section 3 presents the feature-wise comparison of the technologies. Section 4 shows our empirical evaluation , while Section 5 discusses the results. Section 6 illustrates some related works and Section 7 concludes the manuscript.

2. Identified engines

Containers are not only used in the cloud anymore; other computing contexts exploit them and appreciate their characteristics. Docker was the first engine that gained traction among practitioners, and other solutions have been developed over the last years to address particular requirements and provide more appropriate engines. The characteristics of cloud-based engines, which are often common to all solutions, along with the specific needs that come from HPC and IoT applications, explain why we decided to identify three groups: cloud-specific, or general-purpose solutions, and HPC- and IoT-centric engines. We then explain how we selected the six engines by considering their popularity on GitHub, and we sketch their main characteristics.

2.1. Contexts

Containers are used to wrap applications in lightweight, isolated, and self-contained executables that are easily portable among different execution environments. They allow developers to create software in a local machine without worrying about compatibility issues that may arise when migrating to production environments.

Containers have gained a lot of traction in **cloud** environments because they increase the speed of release cycles (i.e., more agility) and reduce the number of operations required (i.e., lower costs) (Hilton et al., 2017). Moreover, oftentimes modern cloud-based applications are developed by following the microservices architecture (Dragoni et al., 2017) where, the application is structured in a set of loosely-coupled, "small" services that can be developed, tested, and deployed independently. In such a setup, each service can be packaged as a lightweight container to isolate their execution and management without adding any significant overhead. Containers can also be used to package off-the-shelf monolithic applications to add portability and isolation.

Novel use cases for containers have emerged recently. Bentaleb et al. (2022) have noticed that in addition to cloud computing, HPC (High-Performance Computing) and IoT (and, similarly, Edge and Fog infrastructures) could benefit from the adoption of containers, but they pose some additional requirements.

HPC systems consist of nodes that are equipped with specialized hardware, networking, and storage solutions designed for highly parallel computations. These systems often have a large number of CPUs and GPUs, and the computation on these devices is typically managed by a workload manager. In this context, managing configurations can be difficult due to the complexity of the platform, the need to integrate with the workload manager, the requirement for specialized libraries (e.g., MPI[1]), and the differences between development environments (i.e., standard computers and the HPC center that hosts the execution).

Containers can help simplify the management of dependencies and libraries, and increase the portability of applications by abstracting away differences between execution environments. This enables the integration of HPC applications into continuous integration pipelines (Sampedro et al., 2018), which have been shown to improve software quality and speed up application evolution (Elazhary et al., 2022).

Furthermore, containers can help isolate resources among different processes without introducing a significant overhead. This is particularly relevant in highly distributed and concurrent setups like HPC infrastructures, where resource contention can significantly reduce overall system performance (Tuncer et al., 2019). With containers, each process is isolated and can be configured to use a set of resources exclusively (which will prevent resource contention by design) or to share them with other containers by defining the priority of each container in the event of resource saturation. The combination of reliable dependency management and deterministic resource allocation mechanisms provided by containers enables more deterministic and reproducible executions, which can help both practitioners and researchers design better HPC-based services and understand their performance.

In addition to the challenges posed by the complexity of HPC systems and the need to manage dependencies and configurations, HPC centers are also multi-tenant and multi-user systems, which makes security a critical concern. In order to protect against security threats such as malware and unauthorized access, it is important for processes to be executed without privileged permission (rootless executions). This helps to prevent processes from gaining access to sensitive resources or from making unauthorized changes to the system.

IoT nodes are characterized by a wide variety of different configurations, which can make it challenging to deploy and manage applications across this heterogeneous environment. In addition, IoT devices often have limited resources such as memory, storage, and processing power, which can make it necessary to use lightweight and highly optimized solutions (Balaji et al., 2019).

Containers can help address these challenges by increasing the portability of applications across devices with different configurations, and by enabling the optimization of resource usage. Containers can also help isolate concurrent processes running on the same device and manage available resources, which is important to prevent resource saturation and to carefully allocate resources to each process. This is particularly important in IoT environments where the number of devices may be large and dynamic, and where the demand for resources may vary over time. By using containers, it is possible to easily increase or decrease the number of instances of an application running on IoT devices, providing a flexible and scalable solution (Botez et al., 2020).

In IoT environments, where devices often operate on constrained energy sources such as batteries, energy and computational efficiency are paramount for sustaining longer device uptime and reducing operational costs. To address these challenges, container engines tailored for IoT should prioritize low computational overhead and provide means for fine-grained resource allocation to optimize the use of limited energy and processing capabilities.

[1] https://www.mpi-forum.org

Finally, the exchange of data between IoT nodes can often involve sensitive or privacy-critical information, and it is important for container engines to prioritize security in order to protect this data and prevent the compromise of devices. To prevent data from being intercepted and accessed by unauthorized parties, it is important to encrypt data as it is transmitted between IoT devices. In addition, container engines should implement measures to ensure that data is not tampered with or modified by unauthorized parties.

As described herein, cloud computing, HPC, and IoT systems pose different challenges and there is no one-size-fits-all solution that can easily accommodate all of them. As we will discussed in the rest of the paper, the state-of-the-art offers container engines that are "general-purpose" and mainly used in cloud setups and others that are specialized to a single context to better address its main issues.

2.2. Selection process

To select the most important container engines in each context (Cloud, IoT, and HPC), we used a semi-automated pipeline[2]: (i) GitHub queries, (ii) automated filtering, and (iii) manual labeling.

We started retrieving a set of GitHub repositories that may provide a container engine dedicated to a given context by issuing three dedicated queries (*container* + <*context*>) to GitHub, where <*context*> was "", to mean "standard" cloud solutions, and *hpc* and *iot* for the other two cases. The search was restricted to either README files or repository descriptions, and we set the number of GitHub stars of an acceptable repository to be equal to or greater than 100. We sorted the results by stars and we analyzed the first 1000 results for each query. At the end of the first step, we obtained 1000, 90, and 385 repositories for cloud-, HPC-, and IoT-focused repositories, respectively.

As additional refinement, we automatically filtered out some of the results not related to container engines. In particular, we removed all the repositories whose descriptions (lower-cased and without punctuation and symbols) did not contain any of the following keywords: *cloud*, *container*, *iot*, *internet of things*, *hpc*, and *high performance computing*. After this step, we remained with 104, 21, and 49 repositories, respectively.

As final step, we manually scanned the results to properly identify container engines. We used four labels to characterize each repository: `ENGINE` (a container engine), `RUNTIME` (a low-level container engine component), `TOOL` (a tool related to containers but not a container engine or runtime), and `N/A` (a non-relevant element).

Container engines and container runtimes serve distinct roles within the ecosystem of containerization. We refer to "container engines" to identify comprehensive solutions that manage the complete lifecycle of a containerized application. This includes the definition, building, and storage of images, version management, registry interactions, and execution. Container engines may also provide functionality for orchestration and networking. On the other hand, container runtimes are specialized components responsible solely for the execution phase. They handle tasks like spinning up, running, and tearing down container instances. Container runtimes are utilized by container engines to perform these specific functions but do not encompass the broader range of capabilities that container engines offer. In summary, while container runtimes focus exclusively on the runtime execution of containers, container engines provide a holistic solution that covers multiple aspects of container management, of which the runtime is just one part.

Among the 104 repositories retrieved for cloud container engines, we tagged 4 with label `ENGINE`, 3 with label `RUNTIME`, and, 46 and 51 with labels `TOOL` and `N/A`, respectively. The four identified container engines are *Docker*[3] (62.7k stars), *Podman* (13.2k stars), *RKT* (8.8k stars) and *Pouch* by Alibaba (4.4k stars); while the three discovered container runtimes are *containerd* (10.6k discovered), *run-c* (9.0k stars), and *cri-o* (3.9k stars). In the context of HPC, we obtained 3 engines, no runtimes, 6 tools, and 12 repositories marked as `N/A`. The three identified container engines are *Charliecloud* (238 stars), *Singularity* by Sylabs (214 stars), and *Apptainer* (196 stars). The last two engines share the same core functionality being originated from the same project (i.e., Singularity has been recently renamed Apptainer). Finally, for IoT we discovered one single engine, *balenaEngine* (586 stars), no runtime, 6 tools, and 42 repositories marked as `N/A`.

We selected the two container engines (if available) with the highest amount of stars from each group, and selected five engines: Docker, Podman, Charliecloud, Singularity, and balenaEngine. Because of its characteristics, we also added Sarus[4] (78 stars), an emerging solution completely developed in academia, and optimized for HPC.

For each selected engine, Table 1 reports the year of the first release (FRY), the number of releases ($\#R$) until now, the size of the project in lines of code (LOC), and the amount of GitHub stars retrieved by our queries ($\star$), to estimate the adoption rate and its reputation.

Table 1
Container engines.

Name	FRY	#R	LOC	⋆
Docker	2013	123	2.331.244	62.7k
Podman	2017	96	971.647	13.2k
Charliecloud	2015	25	8.071	238
Singularity	2015	78	62.819	214
Sarus	2018	10	16.890	78
balenaEngine	2015	127	1.051.024	586

3. Qualitative analysis

All six container engines are open source and their development is active. Most of the implementations are written in Go, but Charliecloud and Sarus: the former is written using a mix of shell scripts and C code, the latter is written in C++. Even if we have identified three groups, each solution, with the exception of balenaEngine that requires a dedicated OS, can be used in any context due to the natural portability of containers (at least on Linux-based systems).

Docker (2022b) was the first popular engine, and it is still probably the most famous one. It provides a complete production platform for developing, distributing, securing, and orchestrating container-based solutions. It is easy to use, well established among developers, and its general-purpose nature fits almost any software project. Docker exploits *containerd* (Containerd, 2022) for managing and running containers, which in turn uses *run-c* (OCI, 2022c) as, low-level container runtime for container creation. Gantikow et al. (2020) is younger compared to Docker. It is based on library *libpod*, from the same developers, that is used for managing the entire container lifecycle. By default, it uses

[2] The results of each step can be found at: https://github.com/deib-polimi/container-engines-selection.

[3] Each engine has its own repository. The core functionality of Docker is now included in the Moby project: https://github.com/moby/moby. Podman is available from: https://github.com/containers/podman. RKT is available from: https://github.com/rkt/rkt, but it was discontinued in 2020 as reported at https://github.com/rkt/rkt/issues/4024. Pouch is available from: https://github.com/alibaba/pouch, containerd from: https://github.com/containerd/containerd, run-c from: https://github.com/opencontainers/runc, cri-o from: https://github.com/cri-o/cri-o, Charliecloud from: https://github.com/hpc/charliecloud, Singularity from: https://github.com/sylabs/singularity, Apptainer from: https://github.com/apptainer/apptainer, balenaEngine from: https://github.com/balena-os/balena-engine, and Sarus from: https://github.com/eth-cscs/sarus.

[4] https://github.com/eth-cscs/sarus

Table 2
Qualitative comparison: ✓ available, ~ partially available, and × not available.

Feature	Doc	Pod	Cha	Sin	Sar	bal
Compatibility						
OCI compliance	✓	✓	~	✓	✓	✓
Support for multiple host operating systems	✓	✓	✓	✓	✓	×
Image management						
Built-in image builder	✓	✓	✓	✓	×	✓
Images from Dockerfiles/Containerfiles	✓	✓	✓	✓	×	✓
Images from SDFs	×	×	×	✓	×	×
Images from git repositories	✓	✓	×	×	×	✓
Images from archives	✓	✓	✓	×	×	✓
Public/private registries	✓	✓	✓	✓	✓	✓
Support for Docker/OCI images	✓	✓	~	✓	✓	✓
Support for SIF	×	×	×	✓	×	×
Optimizations						
Layered images	✓	✓	✓	×	✓	✓
Differential image updates	×	×	×	×	×	✓
On the fly image extraction	×	×	×	×	×	
Fine-grained cache management during image pull	×	×	×	×	×	✓
Image compression	×	×	×	✓	✓	×
Containers executions						
Detached executions	✓	✓	×	✓	×	✓
Dynamic Resource Management	✓	×	×	✓	×	×
Built-in orchestration capabilities	✓	×	×	×	×	✓
Support for workload managers	×	×	✓	✓	✓	×
Support for GPUs	✓	×	~	✓	✓	✓
Explicit support for MPI libraries	×	×	~	✓	✓	×
Security						
Signed container images	✓	✓	×	✓	×	✓
Rootless executions	✓	✓	✓	✓	×	×
Daemonless execution model	×	✓	✓	✓	✓	×
VPN support	×	×	×	×	×	✓

the *run-c* container runtime for instantiating containers but other OCI-compliant runtimes can be also employed (e.g., *katacontainers*[5] or *crun*[6]) .

Priedhorsky and Randles (2017) focuses on the management of containers on HPC frameworks by simplifying the packaging and transmission of HPC applications, and is the lightest solution among the analyzed ones (i.e., only a few thousands of LOC). Godlove (2019) is a container engine optimized for the computation of HPC workloads. It can also be used in cloud deployments and supports Docker images and registries. From version 3.1.0, Singularity is fully OCI compliant, and supports both OCI specifications: image-spec and runtime-spec. Benedicic et al. (2019) is the most recent project, with the lowest number of releases, and has been designed to run containers in the HPC context. It leverages *run-c* as container runtime, and provides an extensible runtime to support current and future custom hardware while achieving native performance.

balenaEngine (Balena, 2022b) is based on the Moby Project (Project, 2022), a framework provided by Docker to assemble specialized container systems, and offers a complete platform for deploying IoT applications using containers.

Table 2 summarizes our analysis and organizes discovered features around five main dimensions: (i) *compatibility*, the degree of compliance to standards and the availability on different operating systems, (ii) *image management*, the ways container engines generate, use, and share container images, (iii) *optimizations*, the techniques employed to reduce resource usage, (iv) *container executions*, the features available at runtime, such as resource management and the support to specific libraries and tools, and (v) *security*, the way container engines guarantee secure image builds and executions. Note that each technology is abbreviated with the first three letters of their name (e.g., *Doc* for Docker).

[5] https://katacontainers.io
[6] https://github.com/containers/crun

3.1. Compatibility

The Open Container Initiative (OCI) is an initiative, led by the Linux Foundation, that focuses on establishing universally accepted standards for containers (Initiative, 2024). It currently comprises two main specifications: the Runtime Specification (*runtime-spec*) (OCI, 2022b) and the Image Specification (*image-spec*) (OCI, 2022a). These specifications provide a standard framework that ensures that any OCI-compliant container runtime can unbundle and execute the contents of an OCI-compliant image in an isolated environment.

We inspected the available documentation of the analyzed container engines and all container engines fully support the OCI standard, with the only exception of Charliecloud that only provides it as an experimental feature[7]: the support is only partial and available with a set of additional commands. OCI assumes that containers be always-on services with a complex lifecycle, while Charliecloud focuses on scientific applications that are usually executed once and are terminated as soon as the required calculations are completed. Similarly Singularity provides both OCI-compliant commands (e.g., *singularity oci exec*) and additional ones (e.g., *singularity exec*).

In terms of compatibility, balenaEngine is the only solution that only works on top of a dedicated operating system, namely balenaOS (Balena, 2022a), that is customized and optimized for the execution of balenaEngine and containers. This Linux-based operating system, which is compatible with a wide range of computer architectures including armv5, armv6, armv7, aarch64, i386, and x86_64, is also responsible for running a containerized daemon (device supervisor) that automatically updates running containers if an update is available.

All other engines can run on full-fledged, standard operating systems.

> **Finding #1** All analyzed container engines support OCI and standard host operating systems with a few exceptions. Docker, Podman, Sarus, and balenaEngine are fully OCI compliant. Singularity is also compatible with OCI but with a dedicated set of commands, and Charliecloud offers it as an experimental feature. balenaEngine can only run on top of its dedicated oerating system.

3.2. Image management

The process of defining container images is often scripted for automation and reproducibility. There are three types of scripts used for this purpose: Dockerfiles, Containerfiles, and Singularity Definition Files (SDFs). Dockerfiles, originally proposed in the Docker ecosystem, are the most widely used scripts for defining container images. They contain a set of instructions that automate the process of creating a container image, and specify everything from the base operating system to the application executable and its dependencies. Containerfiles have the same syntax and semantics as Dockerfiles but are used to signify vendor-neutrality or the intention to operate in a broader ecosystem beyond Docker. Singularity Definition Files (SDFs) offer a different syntax and are unique to Singularity. They enable specific optimizations for high-performance computing environments. An SDF is composed of two parts, the *header*, which defines the configuration of the execution environment (e.g., kernel features, Linux distribution), and *sections*, which execute commands during the build process. SDF allows users to incorporate HPC-specific configurations, such as tailored MPI configurations, GPU environment setups, custom filesystem mappings, and specialized run scripts. While Dockerfiles/Containerfiles can perform some of these tasks, they may require additional scripting or manual interventions after image creation.

[7] https://hpc.github.io/charliecloud/command-usage.html#ch-run-oci

Docker can build images from a Git repository, from a compressed archive, or a Dockerfile. Docker allows one to publish images onto private or public *registries* that act as shared repositories, where images can be versioned through tags. This way, different versions of the same image (e.g., latest, old or beta version) can be associated with the same project. Docker also provides a public, official registry, called *Docker Hub* (Docker, 2022a), that offers a rich set of base images that can be used as a starting point for building new containerized applications. *Docker Registry* allows us to create private registries.

Podman images can be built from a Dockerfile or Containerfile. At its core, Podman uses Buildah (Buildah, 2022) to build container images. Since it replicates all the commands we can find in a Dockerfile, Podman allows the user to build images with or without Dockerfiles while not requiring any root privileges. Podman supports both OCI images, and allows one to pull images from a local directory that stores all the image files, a Docker registry, or an OCI archive, that is, an image that complies with OCI.

For image creation, Charliecloud uses external builders – including Docker and Buildah –, or an internal one, called *ch-image*. The image can be built with or without privileges, depending on the used builder. The created image is wrapped with a Charliecloud interface and converted into a compressed archive that can be easily moved on the HPC cluster. The archive is then unpacked and executed without root privileges.

Singularity allows one to build images using SDF or Dockerfiles and to pull them from external resources such as the Singularity official public registry, called *Singularity Cloud Library* (Sylabs, 2022b) (SIC), and Docker Hub. The container image can be produced in two different formats: a writable sandbox for interactive development, or in the Singularity Image Format (SIF), a compressed read-only format that can be easily moved, shared, and distributed.

Sarus does not provide means to build images and rely on external tools. It supports already-built images that are OCI compliant, and that can be downloaded from public and private registries.

balenaEngine uses Docker behind-the-scene for both image building and usage providing analogous capabilities.

Finding #2. All analyzed engines but Sarus provide means to build images from Dockerfiles/Containerfiles. Singularity can also use a proprietary format for both image definition and building that allows for a simplified configuration of HPC systems.. Non-HPC engines support images created from git repositories and archives, while all the engines allow users to share and download images using private or public registries. Charliecloud supports images loaded from archives.

3.3. Optimizations

All engines provide key optimizations. All but Singularity exploit layered images, an approach pioneered by Docker. A layered image consists of several read-only layers of data, each of them corresponding to a single instruction (e.g., a command in a Containerfile). The layers are stacked and contain only the changes from the previous one. Layered images are particularly useful because they allow reusing any layer as starting point for a different image. Being read-only, the layers shared across different images are only stored once on the disk, and if needed, in memory.

When a new container instance is created from an image, a writable layer, called the container layer, is also created on top of the image layers. This layer hosts all changes made to the running container, for example, it stores newly written files, modifications to existing files, and deleted files to allow for customizing the container. Changes made to the container layer do not affect image layers. This way, different images can share common files and components without the need of storing them multiple times.

balenaEngine optimizes the usage of memory, network and I/O bandwidth, thereby contributing to energy efficiency, which is particularly crucial for resource-constrained IoT devices.

When an image is updated, the devices are notified and only the differences are downloaded, and in case of faults (e.g., out of batteries) container images cannot be corrupted. When working with container images to be generated from archives, balenaEngine builds the image while uncompressing the files without the need for occupying the disk with temporary files. Moreover, during the image pulling process, balenaEngine iterates over each file of every layer and writes it to disk while optimizing for minimal page cache usage. Each file is immediately synced to disk, and the kernel is informed to release its associated cache pages. This approach serves to prevent page cache thrashing and ensures that existing containers continue to run undisturbed even in low-memory situations. Although this method may slightly slow down the image pulling process, it offers a trade-off that is particularly advantageous for IoT use cases by maintaining system performance and application stability.

Sarus does not provide any built-in mechanism for image creation and publication but optimizes existing images (e.g., Docker or OCI ones) by reducing their size through SquashFS (Kernel.org, 2022), an optimization that is also employed by Singularity.

Finding #3. Layered images are a widespread technique to save disk space by allowing images to share common parts. To improve energy and computational efficiency, balenaEngine provides some key optimizations to reduce network transfers (i.e., differential image updates), disk usage (i.e., on-the-fly image extraction), and memory (i.e., fine-grained cache management during image pull). Sarus and Singularity compress images with SquashFS to reduce disk usage.

3.4. Containers executions

When it comes to container executions, one of the main features is the support for "detached" executions, that is, the ability to run long-lasting interactive processes (e.g., a database management system or a web server). While applications executed in the cloud or IoT infrastructures may benefit from this feature, HPC applications are usually non-interactive and terminate as soon as the calculations are completed. For this reason, Charliecloud and Sarus do not support detached executions by design.

While all container engines provide the means to allocate resources (e.g., cores) to a container at startup time, only Docker and Singularity allow for dynamic resource management, that is, the reconfiguration of allocated resources at runtime.

balenaEngine provides some important orchestration capabilities to deploy and update containers on a fleet of registered IoT devices, while Docker provides two dedicated tools: Docker Compose and Docker Swarm.

Docker Compose allows us to configure and run multi-container applications. Users write *docker-compose* files to define all container images used by the application, and the instances we need of each image. Typical use cases are "traditional" three-tier applications or microservices where a single system is composed of multiple separated components (i.e., multiple executables). By default, Docker Compose also sets up a network for the application, and each container is reachable by the others. *Docker Swarm* lets users manage containers deployed across multiple machines. It uses the standard Docker API and networking for merging a pool of Docker hosts into a virtual, single host. It also provides means to scale applications or single containers that can be replicated onto the cluster. Instead, Podman relies on external tools such as Kubernetes. HPC engines do not provide ad-hoc orchestration capabilities.

GPU computations are supported by all the engines, but Podman, assuming that the host operating system has the proper driver installed. Charliecloud provides this feature but requires non-trivial effort[8] (e.g., recompilation) to configure some important GPU-related libraries (e.g., NVIDIA ones). Singularity exploits GPU frameworks such as NVIDIA CUDA[9] and AMD ROCm[10]. In Sarus GPU devices are supported through a dedicated library, namely, the NVIDIA Container Runtime (NVIDIA, 2022) (that is OCI-compatible).

While one can install MPI libraries in any container engine, HPC-dedicated container engines offer explicit support for configuration and execution of MPI-based applications. All such HPC-focused engines include support for MPI libraries, which are essential for complex distributed computations, as well as workload managers for overseeing distributed tasks. Singularity supports both OpenMPI[11] and MPICH[12], and offers two distinct execution models: *hybrid* and *bind*. The *hybrid* model allows for the utilization of both the host's and the container's MPI implementations, if available. The *bind* model relies exclusively on the host's MPI. Charliecloud focuses on OpenMPI, and Sarus is geared towards MPICH; both provide a dedicated set of commands and configuration files. In terms of HPC workload managers, Singularity is compatible with Torque (Computing, 2022), Slurm (SchedMD, 2022), and SGE (SGE, 2022), while both Charliecloud and Sarus integrate well with Slurm.

Finding #4. Detached executions are not supported by Charliecloud and Sarus because of their focus on scientific applications. Docker and Singularity provide means to dynamically allocate resources, while only Docker and balenaEngines have built-in orchestration capabilities. GPU computations are supported by all the engines except Podman. HPC-engines provide ad-hoc support for workload managers and MPI-based computations.

3.5. Security

Docker is a client–server application. The server is the Docker daemon (*dockerd*) that processes the requests sent by clients through a command-line interface (CLI) or a RESTful API and manages Docker images and instances. The daemon is executed with root privileges, and thus only trusted users should be allowed to control the daemon. A *rootless* mode was available as an experimental feature starting from version 19.03 and it has been fully supported since version 20.10. The rootless mode executes the Docker daemon and containers inside an isolated user namespace (a feature that Docker itself uses also to isolate containers with one another). Thus, both the daemon and the containers run without root privileges.

By default, Docker starts containers with a restricted set of Linux kernel capabilities (Man7, 2022)(Hallyn and Morgan, 2008) to allow us to implement a fine-grained access control system. The best practice is to remove all the capabilities except those explicitly required by the application.

Unlike Docker, Podman is daemonless and so it has no background service and the container runtime is executed only when requested. This way, Podman offers less attack surface, because it is not always in execution. It does not require root privileges by leveraging user namespaces. Both Docker and Podman require an initial configuration by a system administrator to enable rootless execution. Podman and Docker allows to sign images to only trust selected image providers and mitigate man-in-the-middle attacks. Docker uses a proprietary signing system, while Podman uses GNU Privacy Guard (GPG) before pushing the image to a remote registry. In this configuration, all the nodes running the container engine must be properly configured to retrieve the signatures from a remote server.

Similarly to Podman, Charliecloud only needs Linux user namespaces to run containers without the need of any privileged operations. It is daemonless and requires only minimal configuration changes on the computing center. It is not designed to be an isolation layer, so containers have full access to host resources. Charliecloud commands only require privileged access when used in conjunction with external builders, such as Docker. This approach still avoids most security risks while maintaining access to the performance and functionality already offered (Priedhorsky and Randles, 2017) since image building is usually executed on the user machine, while only the non-privileged execution is run on in the HPC center preserving its security.

Singularity is daemonless and requires containers to have the same permissions as the users that started them, while the access to files within the container runtime is managed by the standard POSIX permissions. Containers are started with flag $PR\_NO\_NEW\_PRIVS$[13] that prevents applications to gain additional privileges. On multi-user, shared systems, such as HPC centers, Singularity provides an optional *FakeRoot*[14] to allow an unprivileged user to run a container as root. This way, the user has almost the same administrative rights as root but only within the container. In addition, Singularity provides several strategies to ensure safety in these types of environments. Singularity containers can be signed and verified using PGP keys and thus provide a trusted method to share containers. Singularity supports the encryption/decryption of containers at runtime to create a secure and confidential container environment.

Sarus is also daemonless. Root permissions are required to install and work with Sarus. For this reason, Sarus checks a list of conditions to ensure that critical files and directories opened during privileged execution meet them.

balenaEngine provides multiple security layers. Updates are sent to devices in a reliable and verifiable way to protect devices against attacks. An API key is created for each device, balenaOS then controls who can access it, the actions that are permitted, and the available communication channels. balenaEngine uses OpenVPN to control the state of its devices. The VPN disallows device-to-device traffic and prohibits outbound traffic to the Internet. balenaEngine maintains a repository of secure base images. Any resource added to these base images is verified by a GPG key or a checksum.

Note that these security measures may be intrinsically linked to the isolation capabilities of the underlying operating system. Features such as Linux user namespaces, as utilized by Podman and Docker's rootless mode, or Charliecloud's minimalistic design, serve to enhance security by effectively isolating the container environment from the host system. For example, both Docker and Podman use Linux user namespaces to map the UIDs (user ids) and GIDs (groud ids) of a container to different UIDs and GIDs on the host. A user might be an "admin" inside a container but map to a non-privileged user on the host system. This way, even if there is a security vulnerability within the container, the impact is limited because the container does not have real "root" access to the host system. Similarly, the daemonless architectures in Podman, Charliecloud, and Singularity offer an extra layer of protection by reducing the attack surface, as there is no persistently running service that could serve as a target for unauthorized accesses.

[8] https://hpc.github.io/charliecloud/command-usage.html#notes
[9] https://developer.nvidia.com/cuda-toolkit
[10] https://rocmdocs.amd.com/en/latest/
[11] https://www.open-mpi.org
[12] https://www.mpich.org

[13] $PR\_NO\_NEW\_PRIVS$ is a flag of the Linux kernel that can be set for a process to restrict its ability to gain additional privileges. Once this flag is set for a process, neither the process nor any of its children can elevate their privileges.
[14] https://wiki.debian.org/FakeRoot

Finding #5. All container engines except for Charliecloud and Sarus support container image signing to increase reliability. Rootless executions are supported by Podman, Charliecloud, Singularity, and only recently by Docker. Docker and balenaEngine share the same daemon-based architecture, while the others are daemonless. balenaEngine is the only engine to provide VPN support.

4. Quantitative evaluation

To evaluate the performance of the six container engines we carried out an extensive empirical evaluation. We conducted a series of tests organized into five distinct categories. The first four categories aim to explore *key performance metrics*, namely startup and shutdown time, memory footprint, image size, and overhead. The fifth category is specialized on *HPC performance metrics* in utilizing MPI-based applications.

4.1. Experiment setup

To run the tests we used the engines with their default configurations and exploited two different execution environments as detailed in the following.

Experiments on key performance metrics. To run the first four types of tests we used a single-user bare-metal server equipped with an AMD CPU Ryzen 5 2600 @ 3.40 GHz (6 Cores/12 Threads), with 32 GB RAM DDR4 @ 3200MHz, and Ubuntu 19.10. We exploited this configuration to avoid the performance variability introduced by hardware shared among multiple, concurrent jobs, and the overhead introduced by a virtualization system. The tests on startup/shutdown times, memory footprint, and image size used containers created from three popular images that include a ready-to-use Linux distribution: *Ubuntu 18.04*, *CentOS 8*, and *Alpine 3*. We selected these images because they do not contain any application-level dependency that could introduce noise and bias in understanding the impact of the execution times of each engine. These images are Linux distributions that share the host machine's kernel; they are not different OSs as if they were bundled in VMs. They only include distribution-specific utilities and libraries, without the overhead of a separate kernel or fully-fledged OS.

For the tests related to overhead we containerized[15] the Phoronix Test Suite (as explained in Section 4.5) and we used *Ubuntu 18.04* as base image since it is the most popular Linux distribution, and a widely used solution for containers.

Experiments on HPC performance.

To run HPC-dedicated tests, we deployed a cluster of high-performance virtual machines on Microsoft Azure. We used machines of type HB60rs equipped with 60 cores and 240 GB of RAM. These machines also feature a network interface for RDMA (Remote Direct Memory Access) connectivity that allows processes to communicate over an InfiniBand network (100 Gb/s Mellanox EDR with single root input/output virtualization). For the experiments, we used up to 8 instances of HR60rs VMs.

We initially used Azure Cyclecloud, a tool provided by Microsoft, for the configuration of HPC clusters. Since the tool was quite unstable and slow, we created a set of Ansible playbooks[16] to automate the management operations. We set up the cluster with the Slurm workload manager and a distributed file system (NFS[17]). We used the Azure CentOS VM image since it was designed for HPC applications and embeds all the necessary drivers to work with InfiniBand and MPI. We used OpenMPI 4, configured in *hybrid* mode so that both the library installed in the host node and the one in the container can be used (see Section 3).

We built the container images with Docker to use the same solution across the different engines. We created a Dockerfile[18], starting from the one provided by Azure[19] that includes support for InfiniBand and OpenMPI. In addition, we updated the drivers to match the one installed on the host VM and we included the benchmarks we wanted to execute. We used two well-known test suites for MPI: the OSU Micro-Benchmarks[20] by the Ohio State University, and mpiBench[21] by the Lawrence Livermore National Laboratory. While we successfully configured both Singularity and Charliecloud, we could not make Sarus properly execute the benchmarks by using the envisaged OpenMPI tools. However, the team behind Sarus executed some of the benchmarks we used on its own, and the results are publicly available (Sarus, 2022a).

4.2. Startup and shutdown times

Since a new container instance should become available quickly, to fulfill user requests properly, and it should also die quickly, to avoid wasting resources, we measured startup and shutdown times. We measured them by means of the Linux command *netcat*, which allows one to send data packets between a (netcat) server and a client. We installed the netcat server on the host machine and configured each container to send a packet to the host and then terminate. The startup time was measured as the difference between the time the data packet is received by the server and the instant we started the container. The shutdown time is computed as the difference between the instant the container terminated and the one the data packet was received. We also tested how the startup/shutdown times were affected when multiple containers are running concurrently on the same machine.

Table 3 shows the average (μ) startup and shutdown times along with the 95% confidence interval for the mean (in parenthesis) and the standard deviation (σ) of each experiment executed with the three container images. Given that the sample size is 100 for each experiment, we computed the confidence interval using a Z-distribution that is suited for large datasets. To evaluate the performance differences between engines, we examined the overlap between their respective confidence intervals. If the intervals are disjoint, the engine with the lower mean is deemed to outperform the other. On the other hand, overlapping intervals indicate that the performance of the two engines is statistically indistinguishable, and therefore, comparable.

Note that *SinS* abbreviates Singularity with SIF images. To assess the startup and shutdown times of single containers, we repeated the measurements 100 times, waiting 1 s before each execution and termination.

Most of the container engines exhibit a constant behavior: startup and shutdown times remained stable during the experiment. Podman and Singularity (using Docker images) show the highest variance in startup times with multiple spikes during the experiments, especially when using the Ubuntu and CentOS images. For example, Fig. 1(a) shows the startup times of the different engines running CentOS images over the 100 repetitions.

The fastest engine in starting containers is Charliecloud, followed by Singularity with SIF images and Sarus. Docker, Podman, and balenaEngine obtained higher and quite similar startup times, while Singularity (using Docker images) is the slowest implementation. This trend is quite consistent across the three images we used. Note that even if the difference between the fastest and the slowest is some 1.7 s, the

[15] The source code of the containerized test suites can be found at: https://github.com/deib-polimi/containers-test-suites.

[16] https://github.com/deib-polimi/containers-SlurmCluster

[17] http://nfs.sourceforge.net

[18] https://github.com/deib-polimi/containers-DockerHPC

[19] https://github.com/Azure/batch-shipyard/tree/master/recipes/mpiBench-Infiniband-OpenMPI

[20] http://mvapich.cse.ohio-state.edu/benchmarks/

[21] https://github.com/LLNL/mpiBench

Table 3
Startup and shutdown times mean as μ (± 95% confidence interval) and standard deviation (σ) in ms with single containers.

		Ubuntu		*CentOS*		*Alpine*	
		Start	Stop	Start	Stop	Start	Stop
Doc	μ	871 (± 4)	.08 (± .00)	888 (± 4)	.08 (± .00)	875 (± 4)	.08 (± .00)
	σ	29	.01	34	.01	31	.01
Pod	μ	744 (± 24)	.08 (± .00)	1412 (± 46)	.07 (± .00)	559 (± 4)	.08 (± .00)
	σ	178	.03	337	.02	33	.02
Sin	μ	1755 (± 11)	23 (± .80)	1836 (± 21)	23 (± .80)	1740 (± 10)	23 (± .80)
	σ	81	5	156	5	74	5
SinS	μ	125 (± 1)	23 (± .83)	141 (± 1)	23 (± .80)	108 (± 1)	23 (± .77)
	σ	8	5	10	5	7	5
Cha	μ	5 (± .09)	.08 (± .00)	6 (± .09)	.08 (± .00)	5 (± .11)	.08 (± .00)
	σ	.68	.01	.68	.01	.81	.01
Sar	μ	126 (± 1)	23 (± .74)	154 (± 1)	23 (± .71)	109 (± 1)	23 (± .76)
	σ	9	5	11	5	7	5
bal	μ	1007 (± 4)	.08 (± .00)	1012 (± 4)	.08 (± .00)	1009 (± 4)	.08 (± .00)
	σ	30	.01	29	.01	29	.01

Table 4
Startup and shutdown times mean as μ (± 95% confidence interval) and standard deviation (σ) in ms with multiple containers.

		Ubuntu		*CentOS*		*Alpine*	
		Start	Stop	Start	Stop	Start	Stop
Doc	μ	621 (± 8)	402 (± 6)	624 (± 10)	395 (± 6)	610 (± 8)	10434 (± 6)
	σ	45	33	53	31	45	31
Pod	μ	1645 (± 206)	648 (± 104)	4988 (± 371)	1115 (± 127)	474 (± 41)	10329 (± 6)
	σ	1055	533	1894	651	213	35
Sin	μ	2102 (± 90)	78 (± 2)	1912 (± 14)	82 (± 2)	1919 (± 14)	78 (± 2)
	σ	463	10	74	11	73	10
SinS	μ	304 (± 1)	78 (± 1)	306 (± 2)	82 (± 2)	302 (± 1)	77 (± 1)
	σ	7	9	11	11	9	9
bal	μ	748 (± 11)	456 (± 9)	753 (± 13)	455 (± 8)	747 (± 15)	10501 (± 7)
	σ	60	45	70	41	76	37

fastest is some 290 times faster than the slowest and 20 times faster than the second fastest engines. Charliecloud uses a flattened and unpacked image that is ready for execution when the start command is issued and this may be the reason why it is the fastest engine. Singularity SIF and Sarus leverage SquashFS and this may help reduce the startup time compared to the others.

Shutdown times are small, negligible, and jagged for all engines: for example, Fig. 1(b) shows the shutdown times of the different engines running Ubuntu images over the 100 repetitions. Except for Singularity (with both Docker and SIF images) and Sarus, which had shutdown times of around 23 ms, all other engines achieved shutdown times under 1 ms.

We can observe that, for most engines, the startup and shutdown times do not change when using different images.

We also measured the startup and shutdown times when multiple containers are executed concurrently on the same machine. We started with a container and created a new one as soon as the previous one was up and running, incrementally up to 100 concurrent instances. We do not report Charliecloud and Sarus because they do not provide the means to run containers in detached mode (i.e., in parallel in the background).

Table 4 shows the average startup and shutdown times along with the mean 95% confidence intervals and standard deviations. The times obtained by Docker, Singularity (with and without SIF), and balenaEngine are almost constant while the number of running containers increases. On the other hand, Podman shows a noticeable delay during the experiment with the Ubuntu and CentOS images. For example, Fig. 2(a) shows the execution of the experiment with CentOS images and we can observe that the startup time increases at around the execution of the 12th container and again at around the launch of the 22nd. Podman also shows some spikes when running the Alpine image.

Podman exhibits a strong variability during the shutdown phase, especially with Ubuntu and CentOS images (Fig. 2(b) refers to the experiment with CentOS images). Other implementations present a constant behavior and Singularity (using both Docker and SIF images) is significantly the fastest engine in stopping containers. Shutdown times obtained by the other technologies are particularly high (around 10 s) when using a small image such as Alpine while they are small when using Ubuntu or CentOS. The variance is negligible for all the engines but Podman.

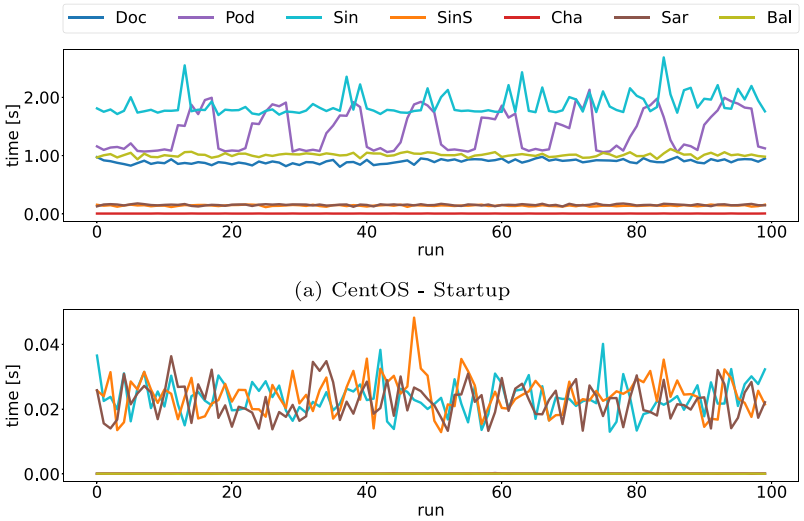
> **Finding #6.** HPC-optimized container engines (Charliecloud, Singularity, and Sarus) are the fastest in terms of startup times. Charliecloud obtained results that are at least two orders of magnitude better than the other implementations. Singularity is fast but only when used with SIF images. Concurrent images do not affect the startup times except for Podman that showed a significant variance. Shutdown times are usually negligible and almost constant but concurrent executions may slow down the operation.

4.3. Memory footprint

We measured the memory footprint to access the memory consumed by an engine to instantiate and execute images. The containerized executable should consume the same amount of memory as the non-containerized version. The memory should only be freed when the container is stopped and deleted to make room for other containers. We obtained it by means of the Linux command *free -m*, which provides information about the total amount of physical and swap memory, and with an engine-specific command if available. We also measured the evolution of memory allocation during container creation using command *nmon*, which supplies a benchmark tool to collect performance data regarding memory and other resources.[22]

Fig. 3(a) shows the evolution of the allocated memory during the creation of 100 container instances from image Ubuntu. As in the

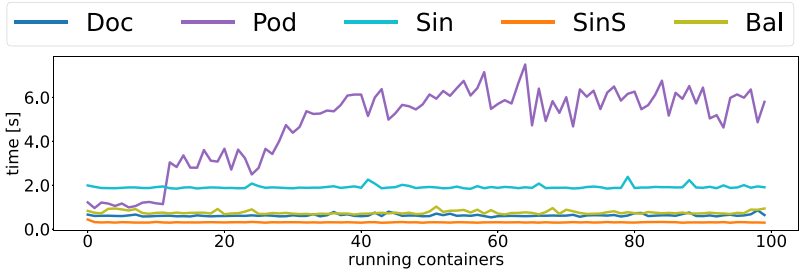
[22] https://www.ibm.com/docs/en/aix/7.2?topic=n-nmon-command.



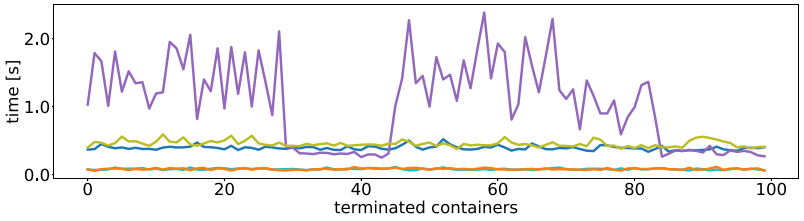
(a) CentOS - Startup

(b) Ubuntu - Shutdown

Fig. 1. Example startup and shutdown times in milliseconds.



(a) CentOS - Startup



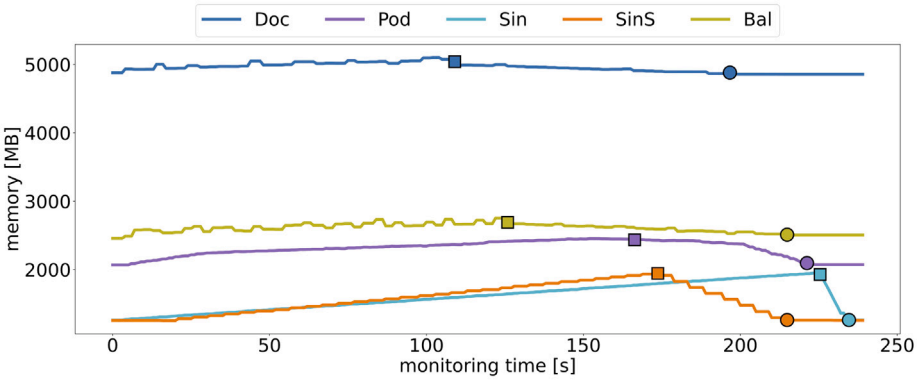
(b) CentOS - Shutdown

Fig. 2. Example concurrent startup and shutdown times in milliseconds.

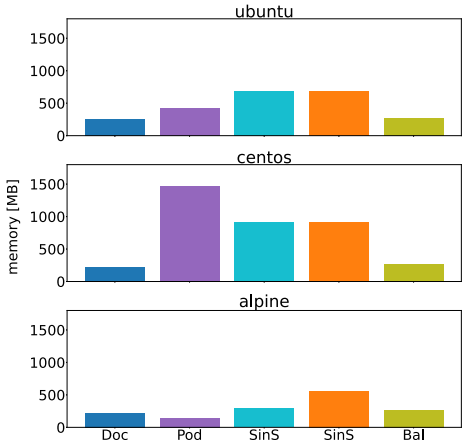
previous experiment, one instance was added as soon the creation of the previous one was concluded, up to 100 containers. When 100 instances were created, we started terminating them one after the other with the same rationale used for their creation. Given that Charliecloud and Sarus do not allow one to run containers in the background, we did not consider them for this experiment.

In the creation phase, the memory grows linearly. We observed some spikes with Docker and balenaEngine, while Singularity and Podman are more stable. During the termination phase, Singularity (without SIF) appeared to deallocate the memory quite fastly with an almost vertical descent; the other engines, instead, do it more gradually. In this experiment, Docker is the fastest to complete the creation phase and the overall process. Fig. 3(b) shows the delta of the memory allocated (from the beginning of the experiment to the creation of the 100th container) by the different engines for the three images. Results appear to depend on the considered image. Singularity allocates more memory than the others with Ubuntu and Alpine, while Podman with CentOS. We did not notice any noticeable memory leakage during the experiments.

Table 5 shows the average memory footprints we measured–using command *free*–during the above experiment. The values are reported in megabytes along with the respective 95% confidence interval. A few interesting trends can be observed from the table. For instance, compared to other container engines, Docker's memory usage appears to be on the higher side for Alpine but stays comparable for Ubuntu and CentOS. When running CentOS, Podman shows a significantly lower memory footprint compared to all the other engines (15±1.0). Singularity with or without SIF obtained comparable performance across the three images and the best of all values in Ubuntu. Balena exhibits overall the highest memory usage, but still remains similar to the others.



(a) Evolution (Ubuntu image). Squares and circles highlight the end of container creation and termination, respectively.



(b) Comparison

Fig. 3. Memory allocation during container creation.

Table 5
Memory footprints in megabytes (mean and 95% confidence interval).

	Ubuntu	*CentOS*	*Alpine*
Doc	24 (± 0.0)	24 (± 0.0)	30 (± 0.0)
Pod	23 (± 0.5)	15 (± 1.0)	28 (± 0.0)
Sin	21 (± 0.0)	27 (± 0.0)	28 (± 0.5)
SinS	21 (± 0.0)	27 (± 0.0)	27 (± 0.0)
bal	27 (± 0.0)	27 (± 0.0)	29 (± 0.0)

Finding #7. In terms of memory footprint, Docker and balenaEngine exhibit occasional spikes during container creation, while Singularity and Podman maintain more stable profiles. During container termination, Singularity deallocates memory most efficiently. Podman shows the lowest memory usage when running CentOS, and Singularity performs best with Ubuntu. Overall, no significant memory leaks were observed across the tested engines.

4.4. Image sizes

We focused on the image size of containerized applications since it should be proportional to the packaged application along with its dependencies. The ability to reuse data (e.g., libraries) contained in other images and image compression techniques can help save disk space. We measured it in three ways: (i) through the data reported in the registry that the engines use, (ii) with the Linux command *du*, which allows a user to gain disk usage information, to obtain the size of the image on the disk, and (iii) with an engine-specific command if available. For Singularity, we run the tests both with Docker images and SIF images.

Table 6 shows the sizes of used images retrieved from a public registry (compressed size, calculated by summing up the size of each image layer) and from the disk (compressed and uncompressed sizes). Note that all the engines, except Singularity that leverages Singularity Hub, use Docker Hub as default public registry. We observed that the size on disk can often exceed the one reported on the registry. Registry data may not consider all the layers of the image and report the compressed size.

Docker, Podman, and balenaEngine allow one to save the image as an archive, and the image size does not change (significantly) when exported. Singularity exploits SIF and SquashFS to compress images, which results in an important reduction of the image size on disk. Charliecloud and Sarus require the smallest images. Charliecloud provides means to store images as compressed archives (i.e., tar files) while Sarus uses SquashFS to optimize image sizes.

Table 6
Image sizes. Data retrieved from [a] *DockerHub*, [b] *Singularity Library*, [c] *docker images*, [d] *podman images*, [e] *sarus images*, [f] *balena images*, [g] folder size, [h] SIF size, and [i] tar file size.

	Ubuntu		CentOS		Alpine	
	Registry	*Disk*	*Registry*	*Disk*	*Registry*	*Disk*
Doc	25 MB[a]	64 MB[c] 64 MB[i]	70 MB[a]	237 MB[c] 234 MB[i]	3 MB	6 MB[c] 6 MB[i]
Pod	25 MB[a]	67 MB[d] 64 MB[i]	70 MB[a]	245 MB[d] 234 MB[i]	3 MB[a]	6 MB[d] 6 MB[i]
Sin	53 MB[b]	109 MB[g] 53 MB[h]	80 MB[b]	293 MB[g] 81 MB[h]	3 MB[b]	6 MB[g] 3 MB[h]
Cha	25 MB[a]	70 MB[g] 25 MB[i]	70 MB[a]	252 MB[g] 69 MB[i]	3 MB[a]	6 MB[g] 3 MB[i]
Sar	25 MB[a]	25 MB[e]	70 MB[a]	67 MB[e]	3 MB[a]	3 MB[e]
bal	25 MB[a]	64 MB[f] 64 MB[i]	70 MB[a]	237 MB[f] 234 MB[i]	3 MB	6 MB[f] 6 MB[i]

Table 7
Resource-level test suite.

	OSBench					Py	7-Zip	SSL	RamSpeed			I/O		sockperf		
	F	P	T	P	MA				Add	Avg	Copy	RE	WR	TP	PPL	LL
Doc	−41%	−7%	−4%	−4%	0%	−29%	−2%	0%	0%	0%	0%	−7%	0%	25%	5%	−7%
Pod	−4%	−5%	−7%	0%	−5%	−29%	−3%	0%	1%	1%	0%	−9%	0%	22%	8%	7%
Sin	1%	−5%	3%	−5%	0%	0%	1%	0%	0%	0%	0%	−2%	1%	6%	1%	5%
SinS	2%	−1%	5%	−3%	3%	0%	0%	0%	0%	0%	0%	−4%	−3%	4%	2%	2%
Cha	4%	9%	15%	5%	1%	0%	−1%	0%	0%	0%	1%	−17%	−3%	2%	0%	3%
Sar	4%	−1%	4%	0%	1%	0%	1%	0%	0%	0%	0%	−2%	−4%	2%	−5%	7%
bal	−12%	−6%	−2%	−1%	0%	−29%	−2%	0%	0%	0%	0%	−10%	−4%	23%	3%	−5%

Finding #8. Docker, Podman, and balenaEngine do not provide any significant optimization to save disk space. SIF helps Singularity reduce significantly the size on disk (around three times smaller than Docker ones in the best case). Charliecloud and Sarus are the best engines in terms of image size (around 50% smaller images than Singularity in the best case).

4.5. Overhead

Container engines introduce a computational overhead to run images. Ideally, the engine should provide performance comparable to the one obtained on bare-metal machines. We conducted our measurements by utilizing the Phoronix Test Suite (PTS) (Suite, 2022). This suite facilitates the running of open-source benchmarks from *OpenBenchmarking.org* (OpenBenchmarking, 2022) and allows one to assemble these benchmarks into customized test suites. We defined two types of test suites: a *resource-level* suite and an *app-level* suite. The resource-level suite conducts low-level assessments focusing on CPU, memory, disk, and network usage. It encompasses the following benchmarks[23]: *OSBench*, *PyBench*, *Threaded I/O Tester*, *RAMspeed SMP*, *7-Zip Compression*, *OpenSSL*, and *Sockperf*. Conversely, the app-level suite carries out application-level evaluations, incorporating benchmarks for Apache HTTP Server, nginx Server, BlogBench, SQLite, Redis, and Apache Cassandra.

Tables 7 and 8 show the overheads measured with the two test suites: resource- and application-level, respectively. Each value in the tables refers to the percentage difference between container-based and bare-metal executions: a negative value means that containers perform worse than bare metal. Each test was repeated 5 times and we present the average values.

[23] Detailed information about each benchmark can be accessed at: https://openbenchmarking.org/tests.

The *resource-level* test suite shows (i) the amount of files created (F), processes (P), threads (T), and programs (P) started, and memory allocated (MA) through OSBench; (ii) the time taken to complete the execution of the Python programs in PyBench (Py); (iii) the time needed to archive/unarchive the files contained in 7-Zip; (iv) the speed ($sign/s$) in generating cryptographic signatures with OpenSSL (SSL); (v) the performance of memory operations *add* and *copy*, and the average (Avg) with RamSpeed; (vi) the bytes read (RE) and written (WR) on disk during Threaded I/O Tester; and (vii) the throughput (TP), ping-pong latency (PPL), and load latency (LL) measured with sockperf.

A first surprising result is the presence of positive percentage values, which indicate a performance improvement in the containerized environment compared to bare-metal configurations. This may initially seem counterintuitive, given the general notion that virtualization layers introduce overhead. However, our study is not the first to report such unexpected outcomes. Previous studies (Morabito et al., 2015) have demonstrated that under specific scenarios, various virtualization techniques can outperform bare-metal performance. For example, Giallorenzo et al. (2021) report improvements of up to 2.5% in RAM speed and a 2x boost in I/O workloads. These gains may be attributed to optimized configurations or to a more efficient usage of system resources in containerized environments. While the underlying reasons for these performance enhancements warrant a more in-depth, system-level analysis, our findings do signal the potential for optimized performance in containerized setups.

As for the values obtained with OSBench, most container engines perform similar to bare-metal, except for file generation with Docker (−41%), and thread management with Charliecloud (+15%). Singularity, Charliecloud, and Sarus obtained results equal to bare-metal ones with PyBench, while the other engines performed significantly worse (−29%). We measured no overheads with 7-Zip, OpenSSL, and RamSpeed for all the engines. The reading performance obtained with test I/O appears to be worse than bare-metal for Charliecloud (−17%),

Table 8
App-level test suite.

	Apache	nginx	BlogBench		SQLite		Redis		Cassandra
			Read	Write	1T	32T	Get	Set	
Doc	15%	−16%	−14%	−6%	−1%	0%	0%	0%	−2%
Pod	19%	22%	−1%	−1%	0%	0%	5%	0%	1%
Sin	0%	0%	−1%	0%	0%	−1%	3%	4%	0%
SinS	0%	0%	0%	0%	0%	1%	3%	−1%	7%

balenaEngine (−10%), Podman (−9%) and Docker (−7%). The majority of Sockperf results obtained with containers are better than bare metal with a significant gain in throughput with Docker, Podman, and balenaEngine. Most of the reported data are very close to the ones of bare metal indicating that containers add a negligible overhead to computations. Being these tests executed on the cloud, both small positive and negative variations may be introduced by disturbances (e.g., resource contention) at the underlying hardware infrastructure.

The results with the *app-level* test suite refer to Apache Web server, nginx, BlogBench, which measures the performance in reading and writing blog posts, SQLite, configured with both 1 and 32 threads, Redis, and Cassandra. Note that these applications are mostly used in cloud-based deployments since they rely on always-on processes, which are not well suited for HPC enviroments, and they are heavily stateful and do not fit the typical "ephemeral" and stateless IoT computations. For these reasons, we do not report data for Charliecloud, Sarus and balenaEngine. We report the results for Singularity since it is sometimes used as a general purpose container engine (e.g., for running web applications[24]).

Table 8 shows that container engines demonstrated performance similar to or better than bare-metal with all applications, except for Docker with nginx (−16%) and with BlogBench-Read (−14%). These tests confirm the results reported for *resource-level* experiments: in general containers do not introduce a significant overhead to computation even when tested with well-known middleware software. Singularity and Podman provide performance that are very close to bare-metal, while we observed small performance degradation with Docker.

Finding #9. Overall, the performance metrics observed for container engines closely align with those measured in bare-metal environments, and even exceed them in certain instances. Docker reported a significant overhead in a few experiments (OSBench file generation, nginx, and blog post read), while all the non-HPC engines showed around 30% worse performance compared to bare-metal in Python-related tests. We observed that HPC-engines provide a smaller overhead compared to other engines, with performance that are almost identical to bare-metal. The maximum difference in performance compared to bare-matel is 5% for Sarus and Singularity, 17% for Charliecloud, 29% for Podman and balenaEngine, and 41% for Docker.

4.6. HPC evaluation

The last set of experiments is dedicated to evaluating the performance of HPC-specific container engines. In particular, the goal of the tests is to analyze the overhead introduced during the execution of computationally intensive and highly parallel applications.

We run three types of experiments: *latency*, *all-to-all*, and *scatter/gather*. The first two are part of the OSU benchmark, while the third is included in mpiBench. Test *latency* measures the min, max, and average latency of a ping-pong communication between two processes. The messages are sent repeatedly with different payload sizes to measure one-way latency. Test *all-to-all* measures the min, max, and average latency of the $MPI\_Alltoall$ operation in which each node sends (receives) a message to (from) all the other nodes, among *all* involved processes. Finally, test *scatter/gather* measures the latency of sending messages (between 0 and 64 kB) through the MPI primitives *scatter* and *gather*. To automate the execution of these benchmarks with Slurm, we developed a set of scripts[25]. The container images were fetched from local disks to avoid any performance degradation due to network overhead.

4.6.1. Results

In the following, we describe the results of the experiments we carried out. Fig. 4 shows some relevant charts that picture different tests run by varying either the cluster or the message size.

Test *latency* was designed to investigate the effect of message size on operation time during the execution of MPI applications with no containers (nocont), Singularity, and Charliecloud. This test was run in two configurations: 2 nodes with 1 core each, and a single node with 2 cores. The message sizes ranged from 1 byte to 4MB, and the operation time was measured in µs.

The results for the two configurations were similar, and the results for the first configuration are shown in Fig. 4(a). The chart clearly shows that the three lines (nocont, Singularity, and Charliecloud) are almost overlapped, indicating that the overhead introduced by the container engines in this test was negligible. This suggests that the container engines had a minimal impact on the performance of the MPI applications in ping-pong communications, even if the message size increased.

Test *all-to-all* was run with 30 cores per node (leaving half of the computational capacity to process incoming messages), three cluster configurations (2, 4 and 7 nodes), and a variable message size (from 0 to 4MB). All the charts show the average latency in µs. Charliecloud and Singularity perform very similarly in all the experiments. When using 7 nodes, the latency is up to 2 times greater than with containers with a message size between 0 and 512 bytes. For larger sizes, the overhead introduced is almost negligible. Similar results are obtained with 2 and 4 nodes with a non-negligible overhead when operating with small message sizes. Although the difference is significant in relative terms, the absolute operation times when dealing with small messages are very small (less than 1 ms).

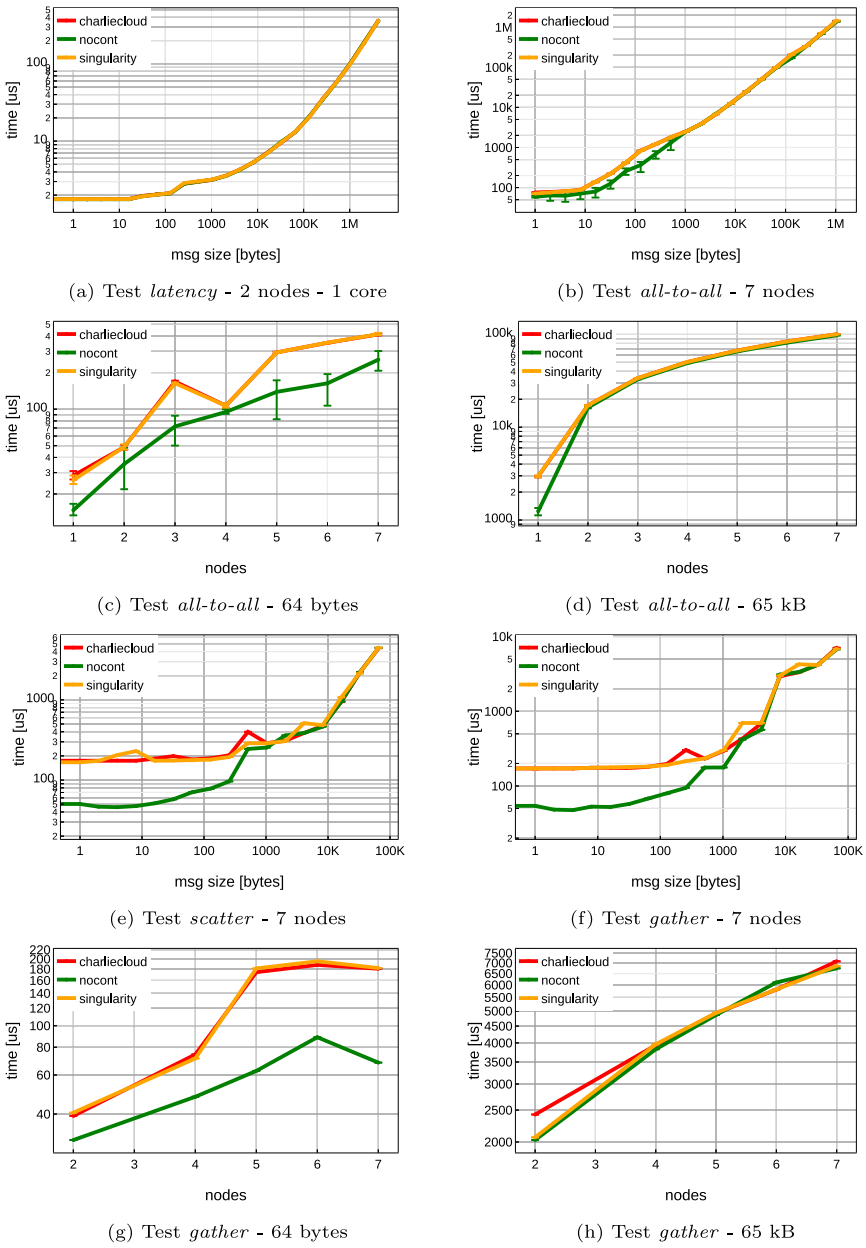
To better visualize the results, we also run test *all-to-all* with a fixed message size (either 8, 16 bytes or 6 kB) and a variable number of nodes. The overhead does not appear to be strongly correlated with the number of nodes in the system. In the experiment with 8 byte messages the latency introduced by container engines ranges from 1.2 to 2 times the one with no containers (*nocont*); similar values were obtained with 64 bytes (Fig. 4(c)). With 65 kb messages, Figure 4(d), the overhead is negligible with more than two nodes.

These experiments suggest that container engines introduce non-negligible overheads (but yet very small in absolute terms) with small messages, while it becomes negligible when the size increases (and if more than one node is employed).

Tests *scatter/gather* were run with 60 cores per node, three cluster configurations (2, 4 and 7 nodes), and a variable message size (from 0 KB to 100 KB). All the charts show the average latency in µs. Figs. 4(e)

[24] https://sylabs.io/2018/09/nodejs-on-singularity/

[25] https://github.com/deib-polimi/containers-MPIBenchmarksBatch



(a) Test *latency* - 2 nodes - 1 core

(b) Test *all-to-all* - 7 nodes

(c) Test *all-to-all* - 64 bytes

(d) Test *all-to-all* - 65 kB

(e) Test *scatter* - 7 nodes

(f) Test *gather* - 7 nodes

(g) Test *gather* - 64 bytes

(h) Test *gather* - 65 kB

Fig. 4. MPI experiments.

and 4(f) show two runs with 7 nodes for tests *scatter* and *gather*, respectively. Similar results were obtained with 2 and 4 nodes.

The obtained results confirm the ones of test *all-to-all*. Charliecloud and Singularity appear to have similar performance and they only introduce overhead with small messages. For example, the two engines added a latency that ranges between 1.3 to 5 times the execution with no container when the message size is between 0 and 256 bytes and we execute *scatter* operations; the overhead becomes negligible with bigger messages. The results obtained with *gather* operations are similar but with an overall smaller maximum overhead.

This behavior is even clearer when fixing message sizes (either 8, 64 bytes or 65 kB) and varying the number of nodes. Figs. 4(g) and 4(h) show test *gather* with a message size of 64 bytes and 65 kB, respectively. As with test *all-to-all*, the overhead is almost constant and relatively significant with the first two configurations, while with messages of 65 kB, the overhead becomes negligible.

Finding #10. Container engines dedicated to HPC obtained almost identical performance compared to executions without containers in tests that involved simple MPI communications. When tested with more complex operations, the overhead is still negligible unless the messages exchanged are small. In this case, the overhead is significant relatively to the executions times which are, in turn, very small (less than 1 ms) in absolute terms.

5. Discussion

The purpose of the second part of our study (Section 4) was to investigate the performance of the six container engines. Through our empirical evaluation, we were able to compare the results obtained by these engines across a wide-range of tests and examine how context-specific solutions may or may not outperform general purpose ones in certain metrics

In most of the tests, we discovered that HPC-specific solutions outperformed all the other ones. For example, Charliecloud, Singularity

with SIF images, and Sarus obtained the fastest startup times. On the contrary, Podman, balenaEngine and Docker showed noticeable delays. Singularity was by far the fastest in stopping multiple running containers. Shutdown times were generally negligible and constant, except for Podman, which showed a significant variance.

We also delved into the memory usage of the engines, recognizing its critical significance in both HPC and IoT domains. In HPC contexts, where multiple concurrent processes often run, it is essential that the memory footprint of the container engine be minimized to avert resource contention. Similarly, in IoT contexts, where resources are inherently limited, efficient memory usage is critical to allow applications to fully harness available resources. Our experiments highlighted that Singularity stands out for its quick memory deallocation, making it favorable for HPC scenarios requiring optimal resource utilization. Podman, with its lower memory footprint, is well-suited for resource-constrained environments like IoT and also offers an advantage in the cloud. Docker, despite its somewhat higher memory usage, offers stable performance, aligning well with cloud infrastructures where resources are typically more plentiful. Although memory is constrained in IoT environments, balenaEngine appeared to lack optimization in this aspect and exhibited per- formance on a par with Docker.

The size of container images can be a an important factor to consider, particularly in IoT where storage may be limited. Our experiments revealed that Charliecloud and Sarus obtained the smallest image sizes among the six container engines. Docker, Podman, and balenaEngine did not offer significant optimization for saving disk space. Singularity uses SIF to compress images, resulting in a significant reduction in disk occupation in some cases.

One key factor we considered was the overhead introduced by each container engine. In HPC, where high performance is required, it is essential for the engine to have a minimal overhead. In IoT, where resources are often constrained, it is also important for the container engine to have low overhead to allow applications fully utilizing the available resources. Our experiments revealed, once again, that HPC solutions outperformed all the other engines. In particular, Sarus and Singularity obtained the lowest overhead among the six container engines with performance that are extremely closed to bare-metal ones (maximum 5% of overhead). Charliecloud also outperformed non-HPC engines. balenaEngine obtained similar performances to Podman, while Docker resulted in the highest overhead (up to 41%). We also found that the overhead introduced by container engines in HPC-specific tests based on MPI is in general very small and more pronounced for small message sizes. As the message size increased, the overhead decreased and eventually disappeared. This suggests that the overhead introduced by container engines is more significant for fine-grained communication patterns.

In IoT settings, energy efficiency and resource utilization are key, as devices often operate under stringent limitations in terms of power, memory, and computational capabilities. Our analysis indicates that balenaEngine might not be the most resource-efficient option when compared to other container engines in terms of overhead and memory usage. Such limitations could impact the operational efficiency of IoT devices, potentially affecting battery life and overall system performance. However, balenaEngine has been engineered with a feature set that is particularly tailored for the unique challenges of the IoT. For example, it strongly optimizes image download and update. It uses differential image updates to minimize network traffic, and reduces both bandwidth consumption and energy usage during updates. It also employs on-the-fly image extraction to optimize storage space and also offers efficient cache management during image downloads.

These features may not directly enhance resource efficiency, but they are important to maintain a robust and secure IoT environment. In terms of security, balenaEngine utilizes unique API keys for each device and supports VPN connectivity to contribute to the secure operation of IoT systems. Moreover, it offers orchestration capabilities to allow for efficient management and scaling of the IoT device fleet, a feature often missing in more resource-efficient alternatives.

Similarly, cloud-based solutions may lack some the optimization of HPC-dedicated engines but they provide a broader feature set (especially in terms of orchestration) and they are easier to use and configure.

In conclusion, our experiments suggest that there is no one-size-fits-all solution for containers. The best choice of container engine will depend on the specific requirements and constraints of the target environment. However, our results provide some guidance on the trade-offs and potential benefits of the different container engines in each context.

5.1. Threats to validity

This section summarizes the most important threats to the validity of presented results by following the structure proposed in Wohlin et al. (2006).

Internal Validity. One key issue that may affect the internal validity of our findings relates to our choice of benchmarks. While we selected a diverse set of benchmarks aimed to offer a comprehensive overview of the performance of container engines, the set is not exhaustive. Other benchmarks could potentially yield different insights and might be considered for future studies. Moreover, the environment configurations we used, particularly for our HPC experiments, might be seen as a potential threat. These experiments were conducted in Azure's cloud-based HPC environments, which inherently involves a layer of double virtualization. Although such a setup could induce performance fluctuations, we mitigated this issue by using benchmarks that repeat tasks across thousands of iterations and confirmed low variance in obtained results.

Another factor that may affect internal validity is the limitation related to running multiple containers concurrently with Charliecloud and Sarus. Our focus was on assessing the native capabilities of each container engine, and Charliecloud and Sarus do not support this feature by default. While workarounds such as using *screen*[26] or *tmux*[27] could be employed, these methods would require manual management of the container lifecycle, diverging from the automated container management features of container engines.

External Validity. As for the generalizability of our findings, our study is based on specific configurations and platforms. While we expect similar performance trends to be observable on other systems, the exact outcomes may differ. Additionally, the methodology we adopted heavily relies on the number of GitHub stars to prioritize projects for evaluation. Although this approach does offer some level of community validation, it might not comprehensively capture all relevant or high-quality projects in the field.

Construct Validity. The metrics used in the study focus on capturing a balanced view of performance across different facets such as start-up time, memory usage, and disk I/O. While these metrics are informative, they may not be fully exhaustive in capturing every performance aspect that could be relevant for the users of container engines. To add robustness to our findings, we calculated confidence intervals for all our measurements. Despite the complexities and the different layers of potential variability inherent in any empirical evaluation, these intervals demonstrate that our results are stable and can be considered reliable.

[26] https://linux.die.net/man/1/screen
[27] https://linux.die.net/man/1/tmux

6. Related work

Containerization emerged as a fundamental enabler for deploying applications in a lightweight and portable way in multiple contexts. For this reason, some significant research work around this technology have been presented in the literature.

Pahl et al. (2019) propose a survey on container-based approaches demonstrating the relevance of the technology and its significant impact on the research landscape.

Plauth et al. (2017) present a performance comparison of containers, unikernels, whole-system virtualization, native hardware, and combinations thereof. In particular, they assess application performance with HTTP servers and databases and evaluate the startup time, image size, network latency, and memory footprint. Their work compares two container technologies (Docker, LXD), three unikernels (Rumprun, OSv, MirageOS), and two virtualization technology (KVM, Xen). While we share a similar goal with this research, our work is different because we focused only on container engines, we analyze their features (and not only performance), and we compare their performance with different benchmarks in different contexts.

Saha et al. (2018) carry out a performance evaluation for executing scientific applications in cloud-based environments with Docker and Singularity. They aim to help practitioners choose the most suitable container engine approach for HPC workloads. They perform four HPC benchmarks (among them also OSU) and find that the performance of different containerization approaches is extremely close to bare metal, a result similar to the one we obtained in our experiments. Another work on this topic is proposed by Arango et al. (2017) and analyzes the features of LXC, Docker, and Singularity. According to their results, Singularity containers are the most suitable containers for HPC. Compared to this paper, our work is broader. It describes more and different container engines, their features, and it does not only focus on HPC.

Liu and Guitart (2021) analyze the performance of containerization in HPC deployments, highlighting how this technology is becoming increasingly important for highly parallel computations. They compare Singularity and Docker against bare-metal executions and they show that for applications that involve intense inter-process communications, containerization provides worse performance compared to bare-metal. In our work, we reach a similar conclusion but only if the communication involves small and frequent messages, while the overhead introduced by containers with larger messages appears to be negligible. As described above, our work does not focus only on HPC and provides a comprehensive comparison among six different container engines.

Potdar et al. (2020) compare the performance of Docker containers and VMs. Their main outcome is that Docker outperforms VMs in every test. Our work is different because compares different container engines with a comprehensive qualitative and quantitative analysis. Similarly, Kozhirbayev and Sinnott (2017) propose a comparison among Docker, LXC and bare-metal executions. They focus on performance and they discover that, for CPU-bounded tasks, the overhead introduced by containerization is negligible. On the contrary, I/O- and network-intensive applications are faster on bare-metal. Compared to our work, they only compare two container technologies and they focus on their performance, whereas we compared both the features and the performance of six container engines.

Salah et al. (2017) compare the performance of applications built with the microservice architecture deployed on VM and container on the Amazon Web Service cloud. For container executions, they rely on Elastic Container Service (ECS), a Container-as-a-Service platform provided by Amazon. They discover that the performance of the application running on VMs is significantly better than the ones executed on ECS. Compared to our work, we focus on six container engines and their performance against bare-metal. We also performed some application-level tests (see Section 4.5) but we did not obtain similar differences in the performance. One possible explanation for this is that ECS introduces additional overhead during the executions.

7. Conclusions and future work

Containerization is a key enabling technology that increases portability across execution environments, eases application management, and speeds up the scaling and reconfiguration of systems. Container engines are the means to create container images, share them on public or private registries, and execute and manage container instances.

This paper analyzes six container engines, compares their features, and presents some experiments to confront their performance. We provide 10 key insights to spread the light on the characteristics of these technologies and on the differences among them. We discovered that, in general, container engines dedicated to HPC are heavily optimized as for performance, cloud-based solutions are richer feature-wise, while IoT tools pose unique challenges that must be tackled by a dedicated approach (e.g., orchestrating a fleet of privacy-sensitive devices).

This paper focuses on container engines as core enablers for running containerized applications, in the future we will provide an analysis of container orchestrators that are becoming increasingly important for managing single or multiple distributed systems in the Cloud and beyond.

CRediT authorship contribution statement

Luciano Baresi: Supervision, Guided the research direction, Writing – original draft. **Giovanni Quattrocchi:** Conceptualization, Supervision, Writing – original draft. **Nicholas Rasi:** Quantitative analysis, Data interpretation, Articulating and presenting the empirical results.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has been partially funded by the national funding for MUR-PRIN projects SISMA (201752ENYB) and EMELIOT (2020W3A5FY).

References

Arango, C., Dernat, R., Sanabria, J., 2017. Performance evaluation of container-based virtualization for high performance computing environments. arXiv:1709.10140.

Balaji, S., Nathani, K., Santhakumar, R., 2019. IoT technology, applications and challenges: a contemporary survey. Wirel. Pers. Commun. 108, 363–388.

Balena, 2022a. Balena OS. URL: https://www.balena.io/os/.

Balena, 2022b. Balena website. URL: https://www.balena.io.

Baresi, L., Quattrocchi, G., 2020. Cocos: A scalable architecture for containerized heterogeneous systems. In: 2020 IEEE International Conference on Software Architecture. ICSA, IEEE, pp. 103–113.

Benedicic, L., Cruz, F.A., Madonna, A., Mariotti, K., 2019. Sarus: Highly scalable docker containers for HPC systems. In: Weiland, M., Juckeland, G., Alam, S., Jagode, H. (Eds.), High Performance Computing. Springer International Publishing, Cham, pp. 46–60.

Bentaleb, O., Belloum, A.S., Sebaa, A., El-Maouhab, A., 2022. Containerization technologies: Taxonomies, applications and challenges. J. Supercomput. 78 (1), 1144–1181.

Botez, R., Strautiu, V., Ivanciu, I.-A., Dobrota, V., 2020. Containerized application for IoT devices: comparison between balenaCloud and Amazon web services approaches. In: 2020 International Symposium on Electronics and Telecommunications. ISETC, IEEE, pp. 1–4.

Buildah, 2022. Buildah. URL: https://buildah.io/.

Celesti, A., Mulfari, D., Fazio, M., Villari, M., Puliafito, A., 2016. Exploring container virtualization in IoT clouds. In: 2016 IEEE International Conference on Smart Computing. SMARTCOMP, IEEE, pp. 1–6.

Charliecloud, 2022. CharlieCloud website. URL: https://hpc.github.io/charliecloud/.

Computing, A., 2022. Torque. URL: https://adaptivecomputing.com/cherry-services/torque-resource-manager.
Containerd, 2022. An industry-standard container runtime with an emphasis on simplicity, robustness and portability. URL: https://containerd.io/.
Docker, 2022a. Docker hub. URL: https://hub.docker.com/.
Docker, 2022b. Docker website. URL: https://www.docker.com/.
Docker, 2022c. Dockerfile reference. URL: https://docs.docker.com/engine/reference/builder.
Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L., 2017. Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering. Springer, pp. 195–216.
Elazhary, O., Werner, C., Li, Z.S., Lowlind, D., Ernst, N.A., Storey, M.-A., 2022. Uncovering the benefits and challenges of continuous integration practices. IEEE Trans. Softw. Eng. 48 (7), 2570–2583. http://dx.doi.org/10.1109/TSE.2021.3064953.
Gantikow, H., Walter, S., Reich, C., 2020. Rootless containers with podman for hpc. In: International Conference on High Performance Computing. Springer, pp. 343–354.
Giallorenzo, S., Mauro, J., Poulsen, M.G., Siroky, F., 2021. Virtualization costs: benchmarking containers and virtual machines against bare-metal. SN Comput. Sci. 2 (5), 404.
Godlove, D., 2019. Singularity: Simple, secure containers for compute-driven workloads. In: Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning). PEARC '19, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3332186.3332192.
Hallyn, S.E., Morgan, A.G., 2008. Linux capabilities: Making them work.
Higgins, J., Holmes, V., Venters, C., 2015. Orchestrating docker containers in the HPC environment. In: International Conference on High Performance Computing. Springer, pp. 506–513.
Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D., 2017. Trade-offs in continuous integration: Assurance, security, and flexibility. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2017, Association for Computing Machinery, pp. 197–207.
Initiative, O.C., Open Container Initiative. URL: https://www.opencontainers.org/.
Kernel.org, 2022. SQUASHFS: a squashed read-only filesystem for Linux. URL: https://www.kernel.org/doc/Documentation/filesystems/squashfs.txt.
Kozhirbayev, Z., Sinnott, R.O., 2017. A performance comparison of container-based technologies for the cloud. Future Gener. Comput. Syst. 68, 175–182.
Liu, P., Guitart, J., 2021. Performance comparison of multi-container deployment schemes for HPC workloads: an empirical study. J. Supercomput. 77 (6), 6273–6312.
Man7, 2022. Overview of linux capabilities. URL: http://man7.org/linux/man-pages/man7/capabilities.7.html.
Merkel, D., 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. Linux J.
Microsoft, 2022. Run your first Windows container. URL: https://docs.microsoft.com/en-us/virtualization/windowscontainers/quick-start/run-your-first-container.
Morabito, R., Kjällman, J., Komu, M., 2015. Hypervisors vs. lightweight virtualization: a performance comparison. In: 2015 IEEE International Conference on Cloud Engineering. IEEE, pp. 386–393.
NVIDIA, 2022. NVIDIA container runtime. URL: https://developer.nvidia.com/nvidia-container-runtime.
OCI, 2022a. Open container initiative image format specification. URL: https://github.com/opencontainers/image-spec.
OCI, 2022b. Open container initiative runtime specification. URL: https://github.com/opencontainers/runtime-spec.
OCI, 2022c. Open Containers Initiative: runC: CLI tool for spawning and running containers according to the OCI specification. URL: https://github.com/opencontainers/runc.
OpenBenchmarking, 2022. A centralized testing ecosystem. URL: https://openbenchmarking.org/.
Pahl, C., 2015. Containerization and the paas cloud. IEEE Cloud Comput. 2 (3), 24–31.
Pahl, C., Brogi, A., Soldani, J., Jamshidi, P., 2019. Cloud container technologies: A state-of-the-art review. IEEE Trans. Cloud Comput. 7 (3), 677–692.
Plauth, M., Feinbube, L., Polze, A., 2017. A performance survey of lightweight virtualization techniques. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (Eds.), Service-Oriented and Cloud Computing. Springer International Publishing, Cham, pp. 34–48.
Podman, 2022. Podman website. URL: https://podman.io/.
Potdar, A.M., Narayan, D., Kengond, S., Mulla, M.M., 2020. Performance evaluation of docker container and virtual machine. Procedia Comput. Sci. 171, 1419–1428.
Priedhorsky, R., Randles, T., 2017. Charliecloud: Unprivileged containers for user-defined software stacks in HPC. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '17, Association for Computing Machinery, New York, NY, USA, http://dx.doi.org/10.1145/3126908.3126925.
Project, M., 2022. An open framework to assemble specialized container systems. URL: https://mobyproject.org/.
Saha, P., Beltre, A., Uminski, P., Govindaraju, M., 2018. Evaluation of docker containers for scientific workloads in the cloud. In: Proceedings of the Practice and Experience on Advanced Research Computing. ACM, http://dx.doi.org/10.1145/3219104.3229280, URL: http://dx.doi.org/10.1145/3219104.3229280.
Salah, T., Zemerly, M.J., Yeun, C.Y., Al-Qutayri, M., Al-Hammadi, Y., 2017. Performance comparison between container-based and VM-based services. In: 2017 20th Conference on Innovations in Clouds, Internet and Networks. ICIN, pp. 185–190. http://dx.doi.org/10.1109/ICIN.2017.7899408.
Sampedro, Z., Holt, A., Hauser, T., 2018. Continuous integration and delivery for HPC: Using singularity and jenkins. In: Proceedings of the Practice and Experience on Advanced Research Computing. PEARC '18, Association for Computing Machinery.
Sarus, 2022a. OSU micro benchmarks. URL: https://sarus.readthedocs.io/en/stable/cookbook/osu_mb/osu_mb.html.
Sarus, 2022b. Sarus documentation. URL: https://sarus.readthedocs.io/en/stable/.
SchedMD, 2022. Slurm workload manager. URL: https://slurm.schedmd.com/.
SGE, 2022. SGE documentation. URL: https://docs.hpc.shef.ac.uk/en/latest/sharc/sge.html.
Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L., 2007. Operating system virtualization: A scalable, high-performance alternative to hypervisors. In: Proc. of the 2nd ACM European Conference on Computer Systems. Vol. 41. ACM.
Suite, P.T., 2022. Open-source, automated benchmarking. URL: http://phoronix-test-suite.com/.
Sylabs, 2022a. Singularity. URL: https://sylabs.io/singularity/.
Sylabs, 2022b. Singularity cloud library. URL: https://cloud.sylabs.io/library.
Tuncer, O., Ates, E., Zhang, Y., Turk, A., Brandt, J., Leung, V.J., Egele, M., Coskun, A.K., 2019. Online diagnosis of performance variation in HPC systems using machine learning. IEEE Trans. Parallel Distrib. Syst. 30 (4), 883–896. http://dx.doi.org/10.1109/TPDS.2018.2870403.
Wohlin, C., Höst, M., Henningsson, K., 2006. Empirical research methods in web and software engineering. In: Mendes, E., Mosley, N. (Eds.), Web Engineering. Springer, Berlin, Heidelberg, pp. 409–430. http://dx.doi.org/10.1007/3-540-28218-1_13.

Luciano Baresi is a full professor at the Politecnico di Milano. His research interests are in the broad area of software engineering and include formal approaches for modeling and specification languages, distributed systems, service-based applications and mobile, self-adaptive, and pervasive software systems.

Giovanni Quattrocchi is working as assistant professor at the Politecnico di Milano. His research interests include self-adaptive systems, distributed systems, and software engineering for AI.

Nicholas Rasi received his master degree from Politecnico di Milano in 2019 where has been research assistant until 2022. His research interests include heterogeneous systems, self-adaptive systems, service-oriented architectures and performance analysis.