Understanding the performance of container execution environments

Guillaume Everarts de Velp, Etienne Rivière and Ramin Sadre
ramin.sadre@uclouvain.be
EPL, ICTEAM, UCLouvain, Belgium

Abstract

Many application server backends leverage container technologies to support workloads formed of short-lived, but potentially I/O-intensive, operations. The latency at which container-supported operations complete impacts both the users' experience and the throughput that the platform can achieve. This latency is a result of both the bootstrap and execution time of the containers and is impacted greatly by the performance of the I/O subsystem. Configuring appropriately the container environment and technology stack to obtain good performance is not an easy task, due to the variety of options, and poor visibility on their interactions.

We present in this paper a benchmarking tool for the multi-parametric study of container bootstrap time and I/O performance, allowing us to understand such interactions within a controlled environment. We report the results obtained by evaluating a large number of environment configurations. Our conclusions highlight differences in support and performance between container runtime environments and I/O subsystems.

CCS Concepts • **Cloud computing**;

ACM Reference Format:
Guillaume Everarts de Velp, Etienne Rivière and Ramin Sadre. 2020. Understanding the performance of container execution environments. In *Containers Workshop on Container Technologies and Container Clouds (WOC '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3429885.3429967

1 Introduction

Containers have emerged as a standard approach for environments supporting on-demand, short-lived execution of computational tasks. Examples of such environments include Function-as-a-Service (FaaS) platforms [11] and edge computing environments [8, 9]. More specifically, our motivating example is an automatic grading platform named INGInious [2, 3]. This platform is used extensively at UCLouvain and other institutions around the world to provide computer science and engineering students with automated feedback on programming assignments, through the execution of series of unit tests prepared by instructors. It is necessary that the student code and the testing runtime run in isolation from each others. Containers answer this need perfectly: They allow students' code to run in a controlled and reproducible environment while reducing risks related to ill-behaved or even malicious code.

Service latency is often the most important criteria for selecting a container execution environment. Slow response times can impair the usability of an edge computing infrastructure, or result in students frustration in the case of INGInious. Higher service latencies also result in higher average resource utilization and, therefore, in lower achievable throughput. In the context of a FaaS platform, this can result in lower return-on-investment. For INGInious, where large-audience coding exams may involve hundreds of submissions per minute, it results in increased resource requirements.

Many factors influence service latency. We focus in this paper on the impact of the I/O subsystem. We consider its impact on latency both in the *bootstrap* phase (i.e., prior to executing a function, service a request with an edge service, or running an INGInious task) and in the actual *execution* phase (i.e. when accessing a database or using the file system).

A difficulty that deployers of container-based execution environments face is the diversity of choices available to support their workloads, i.e. the technical components of their container execution environment. This choice starts, obviously, with the actual container *runtime environment*. In addition to container solutions based on Linux's `namespaces` and `cgroups` functionalities, such as *runc*, *crun* or *LXC*, alternatives are rapidly emerging that blur the lines between OS-level and machine-level virtualization. Lightweight machine virtualization technologies such as Firecracker [1] reduce the delay for bootstrapping an actual virtual machine, while retaining its better isolation properties compared to OS-level virtualization. Other solutions, such as Kata Containers [7], allow running containers directly over an hypervisor, be it Firecracker or KVM. Alternatives are not limited to the runtime environment. Deployers must also make choices for the storage subsystem, that influence the performance of I/O intensive container workloads: We identify for instance no

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WOC '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8209-0/20/12…$15.00
https://doi.org/10.1145/3429885.3429967

less than 8 different storage drivers that can interface with the different abovementioned container runtime solutions.

Contribution We are interested in this paper in the impact of the container environment on service latency, both for bootstrap and execution. It is difficult to identify which component impacts most these latencies in production environments given their intrinsic complexity and the influence of multiple external factors, such as the existing load on the platform or the co-existence of other workloads.

Our first contribution is therefore a benchmarking tool that helps to isolate the factors related to the configuration of the container execution environment by performing a systematic experimental analysis of a large number of valid configurations, i.e. combinations of possible technological choices at the various levels of the container execution environment stack. We identify the components that influence I/O performance and therefore service latency as: the container manager, the container runtime, the storage driver, the base image used for the container, and finally the control group mechanism. Considering all compatible alternatives for these components, the benchmarking tool evaluates a total of 72 valid configurations. These configurations reflect the requirements of the INGInious application, but are also characteristic of other service backends requirements (i.e. the need for isolation, resource management, network access and a writable filesystem).

As our second contribution, we present in this paper the key insight and results that we obtained by running our tool with five workloads representative of containers making intensive use of the I/O subsystem. In a nutshell, we highlight the importance of using lightweight and low-level container runtimes and the important influence of the storage drivers on I/O subsystem performance and, as a result, on perceived bootstrap and execution latencies. Our complete results are available in the companion report [5]. Our datasets and benchmarks are also publicly available [4].

Outline We first present our methodology (§2). We then detail the most significant results from our experiments (§3). We finally review related work (§4) and conclude (§5).

2 Methodology

We detail how we select different container execution environments how we evaluate their performance.

2.1 Metrics and tested components

We build a benchmarking tool that helps to quantify the impact of different container technologies on the performance of containerized applications. Our primary performance metrics are the startup latency and the time applications need to complete their tasks (which we refer to as bootstrap and execution latencies). Although other types of metrics, such as CPU or memory utilization, might be of interest for certain use cases, there are not our primary focus. Consequently, the tool assigns every test run the same amount of resources, that is one logical CPU core and 1 GB of memory.

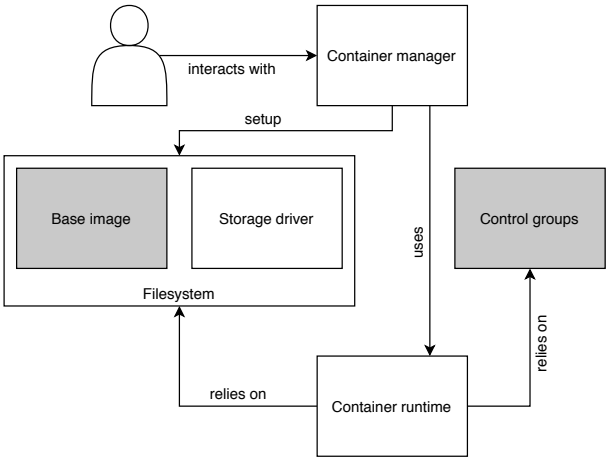

Figure 1. Components of a container execution environment.

Deploying a container-based application requires a combination of different complementary technologies. In the following, we will call such a combination a *container execution environment*. Figure 1 gives an overview of its components:

- The **container manager** is a user-friendly tool which allows to manage the lifecycle of containers, attach their execution to a terminal etc.
- The **container runtime** is the system support mechanism responsible for creating, starting, and stopping containers and allowing other low-level operations.
- A **storage driver** provides and manages the filesystem of a container. It ensures that each container filesystem is isolated from that of other co-located containers.
- **Control groups** (`cgroups`) in Linux is used for OS-level virtualization for controlling resource usage for the processes belonging to a specific container.
- The **base container image** contains the operating system to run inside the container.

For each of these components, various concrete implementations are available in the container ecosystem. Table 1 details the ones that we have considered in our benchmarking tool. We consider three container managers and five container runtime environments, three based on OS-level virtualization (*runc*, *crun* and *LXC*) and two based on lightweight machine-level virtualization (Kata containers over QEMU or Firecracker). Two of the tested storage drivers provide file-based copy-on-write (*aufs* and *overlay2*) or block-based copy-on-write (*btrfs*, *devicemapper*, *lvm*, *zfs*). The other drivers (*directory* and *vfs*) do not support copy-on-write. For the control groups, we prefer, where possible, *cgroups v2* which is the newer version of *cgroups* (since Linux kernel version 4.5) and tackles the flaws of the older version while

Component	Implementations
Container manager	*Docker*, *Podman*, *LXD*
Container runtime	*runc*, *crun*, *LXC*, *Kata containers default runtime (with QEMU)*, *Kata containers with Firecracker*
Storage driver	*aufs*, *btrfs*, *devicemapper*, *directory*, *overlay2*, *vfs*, *zfs*, *lvm*
Control groups	*cgroupv2 (for Podman with crun)*, *cgroup*
Base container image	*Alpine*, *CentOS*

Table 1. Studied container components.

keeping all of its functionalities. For the base container image, we consider images provided by Docker based on the lightweight *Alpine* distribution and the popular *CentOS*.

Having three different container managers, five different container runtimes, eight storage drivers, and two base images to choose from, could give the impression that there are 240 possible container execution environments to evaluate. This is not the case, as some of the components can only be combined with certain others. For example, the container manager *LXD* requires the *LXC* runtime and vice-versa and is the only container manager that supports the *directory* storage driver. On the other hand, *Kata Containers with Firecracker* only supports the *devicemapper* storage driver. In total, 72 configurations are valid out of the 240 combinations. We provide a complete list of constraints and impossible combinations in the companion report [5].

2.2 Workload

The benchmarking tool consists of the necessary scripts and configuration files to evaluate the different container execution environments under a set of predefined workloads. We selected those workloads to highlight and isolate different aspects of containerized applications, such as the bootstrap latency or the execution latency of I/O intensive applications. In the following, we describe five selected workloads that we use in this paper — a more comprehensive list is available in the companion report [5].

1. **Hello World**: Measures the time required to create a container which will print `Hello World` on standard output. This test mainly measures the startup latency of a container application.
2. **Database read**: Measures the time required to read the entire content of a SQLite database stored inside the container filesystem. This test is performed for 5 different database sizes: 151.6 kB, 536.6 kB, 2.6 MB, 11.9 MB and 111.6 MB.
3. **Database write**: Measures the time required to write content to a database stored inside the container filesystem. We use the same data sizes as previously.

Processor	Intel(R) Core(TM) i5-2410M
RAM	8GB DDR3 1333MT/s
Storage	256GB SSD SATA III (6GB/s)
Operation system	Ubuntu server 18.04.4 LTS
Linux kernel version	4.15.0-101-generic

Table 2. Hardware configuration.

4. **I/O read**: Measures the time required to read a large amount of files of small size (4 kB) with random content. This is repeated with 5 different numbers of files: 10, 100, 1000, 10 000 and 100 000 files.
5. **I/O write**: Measures the time required to write a large amount of files of small size (4 kB) with random content. As in the previous test, this is repeated five times for different numbers of files. To avoid being bottlenecked by the creation of random content (depending on the entropy capability of the system) an uncompressed `tar` archive is simply extracted in this test.

Each test is repeated 20 times for each configuration. For the results reported in the next section, we have used a relatively modest testing environment whose characteristics are detailed in Table 2. This represents a use case where containers should be launched on edge-computing class hardware. That said, the benchmark tool and its scripts have been designed to be reused as easily as possible, even helping with the configuration of new test environments with Ansible configuration files (playbooks). In this way, the experiments can be easily repeated by interested researchers on other types of hardware, for example of data-center class.

3 Results

In the following, we show interesting and noteworthy results that illustrate the usefulness of our benchmarking tool for comparing different container execution environments.

3.1 Impact of the storage driver

The choice of the storage driver directly impacts the starting delay of the container. Firstly for the creation, where storage drivers without any copy-on-write mechanism need to copy the whole file system (like *vfs*), then for starting the container during which the container filesystem needs to be accessed for read operations, and finally during the execution of an application to access the necessary files.

As mentioned in §2.1, components have different constraints on the storage drivers they can use. To illustrate the impact of the drivers on container performance, we have therefore chosen a configuration that supports most of the presented storage drivers and therefore allows a direct comparison: the container manager *Docker* with the *runc* runtime and the *Alpine* image.

Figure 2 shows the time needed to create, start and execute the *Hello World* workload using the different storage drivers. Note that the *bootstrap* time is the combination of

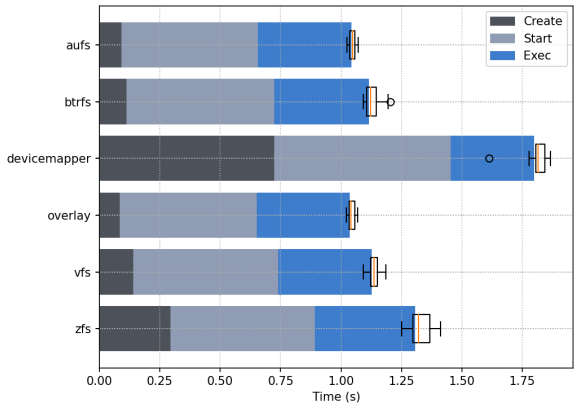

Figure 2. Creation, Start, and Execution time of the *Hello World* workload using Docker, runc and the Alpine image.

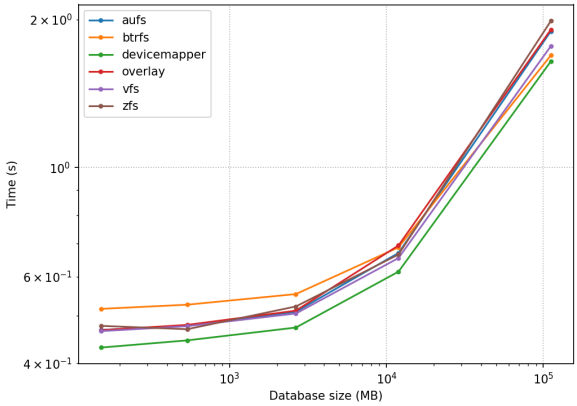

Figure 3. *Database Write* benchmark (execution time) with *Docker* and *runc*. Note the logarithmic y-axis.

the create and start times. The two fastest drivers *aufs* and *overlay2* rely on the same union file system mechanism, with *overlay2* being slightly faster than *aufs*. The *devicemapper* driver needs considerably more time to create and start the container and should therefore not be used if low latency is desired. Execution times are similar for all drivers.

Figure 3 shows the execution time of the *Database write* benchmark. The file-based copy-on-write of *overlay2* and *aufs* makes them slower than *btrfs* and *devicemapper* with block-based copy-on-write because the former have to copy the whole database table before they can modify it. Note that *vfs* makes a full copy of the container filesystem at container creation time, which explains its good performance here and, given the small size of the *Alpine* image, also in the *Hello World* benchmark above. The reason for the poorer performance of *zfs* with large database sizes is not clear.

3.2 Base image

In general, the size of the base image has only little influence on the container runtime performance when used with a copy-on-write storage driver. Although the Alpine image provided by Docker is only 5.6 MB, while the CentOS image size is 203 MB, they behave similarly, as shown in the left and center plots in Figure 4 for the *I/O read* benchmark. For this figure, we chose the *btrfs* and *overlay2* storage drivers. Both drivers showed a good overall performance in §3.1, but, as visible in the figure, *overlay2* can handle workloads that involve a large number of files better than *btrfs*.

However, the base image can also have sometimes a complex and difficult to debug impact on container performance. This is illustrated by the right plot in Figure 4, which shows the benchmark results for the *I/O write* benchmark. Obviously, the benchmark runs much slower with Alpine than with CentOS. This effect is likely caused by differences in the `tar` tool provided by the distributions' respective package repositories. We noticed that the implementation in the Alpine image makes a lot more system calls (about three times more) than the CentOS one.

Overall, the choice of CentOS for the base image can then be justified by its maturity. It had been around for a while, it is widely used outside of container applications, and is more likely to include optimizations in the different tools available. When those tools are not required though, the minimalist design and light weight of Alpine can make it a good choice.

3.3 Container runtime

The container runtimes supported by the benchmarking tool differ considerably in their capabilities and implementations. For Figure 5, we selected four runtimes and report the results for the *Hello World* benchmark using the *overlay* storage driver, which is the best performing driver for this benchmark according to Figure 2. We have also included results for the slower *devicemapper* driver since this is the only driver supported by Kata Containers with Firecracker.

Clearly, Kata Container solutions are the slowest, although their performance is relatively close to OS-virtualized solutions. Unlike the latter, they run the application in small virtual machines, which provide better isolation than ordinary containers, but come with a greater overhead in the startup phase because a whole kernel has to be loaded and started. Their usage can be easily justified for applications with long container lifetime and where a better isolation is needed. Another point worth to mention is the longer start time of Kata Containers with Firecracker (kata-fc) compared to Kata Containers with QEMU (kata-runtime), even when both are using the same storage driver. It should be however noted that Firecracker has not been conceived to run under Kata Containers' hood and might perform better when running standalone. Furthermore, Firecracker has advantages that are not subject to our benchmarking tool, for example the memory footprint, which is smaller than with QEMU. Finally, Kata Containers uses a feature of QEMU (vNVDIMM) that accelerates access to the filesystem and that is not present in Firecracker.

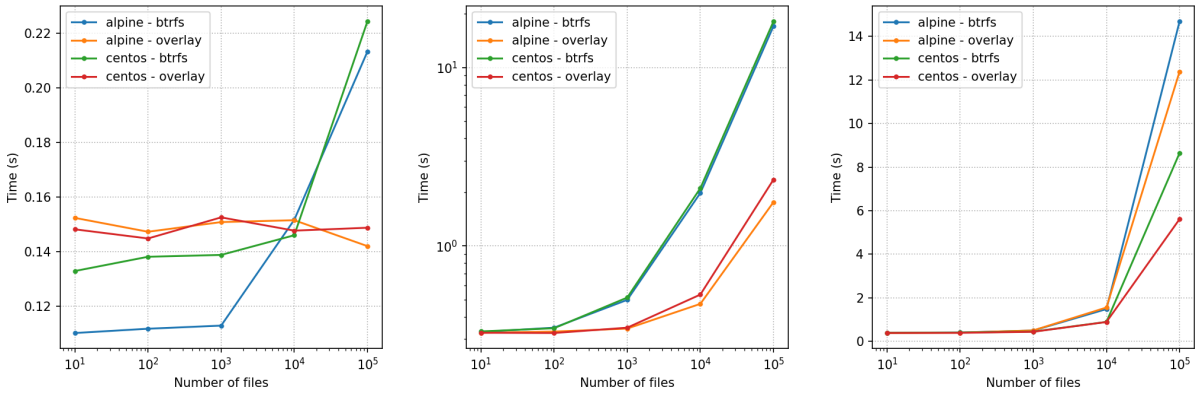

Figure 4. Comparison of Alpine and CentOS with the I/O benchmarks. Left: Creation time, *I/O read* benchmark. Center: Execution time, *I/O read* benchmark. Right: Execution time, *I/O write* benchmark. All experiments use *Docker* and *runc*.

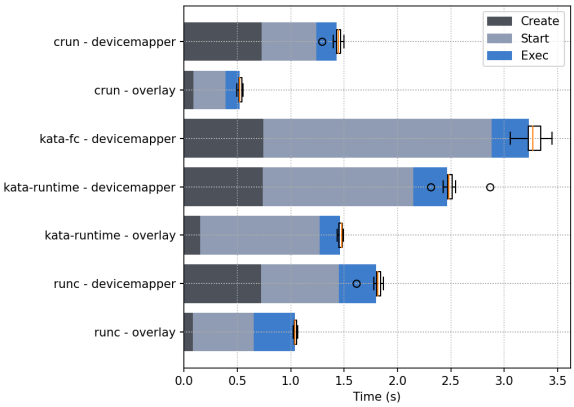

Figure 5. Creation, Startup, and Execution time of *Hello World* for different container runtimes with *Docker* and the *Alpine* image.

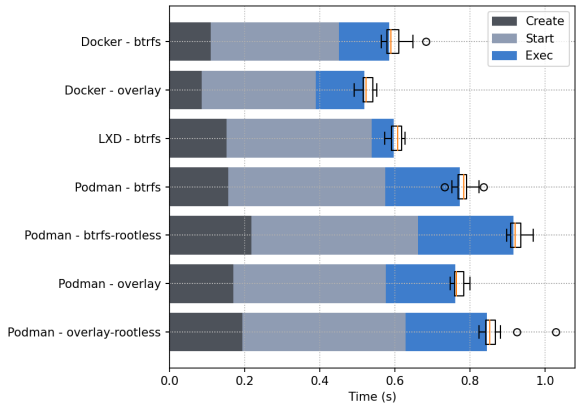

Figure 6. Creation, Startup, and Execution time of *Hello World* for different container managers with *crun* (resp. *LXC* for *LXD*) and the *Alpine* image.

For applications where the lifecycle of the container is short and the isolation provided by simple namespaces is good enough, *crun* does a better job than *runc*. The performance difference can be possibly explained by the fact that the former is implemented in C, while the latter is implemented in Go. The difference is more obvious during the start and execution as those are the phases where low-level operations are made by the container runtime (entering namespaces, setting control groups, forking processes).

3.4 Container manager

All tested container managers provide good performance with the considered workloads. This is shown in Figure 6 for the *Hello World* benchmark using *crun* (with the exception of the *LXD* manager which requires the *LXC* runtime). We notice that *Podman* is slightly slower than the other managers, but given its relatively young age, its focus might not be yet fully on performance, but rather on features.

An interesting feature of *Podman* is its support for *rootless* containers. Processes launched in rootless containers are not owned by root (although they are mapped to the root `uid` inside the container) and the container runtime is not executed as root either. In *Docker*, the default choice is to run the container manager and the processes inside the container as root. Unfortunately, having rootless containers also comes with a cost as we can also see in Figure 7: Creating a rootless container takes more time (left image). In addition, rootless containers can not use OverlayFS. An alternative based on a user-space implementation using FUSE shows significantly lower performance (center and right image) compared to the original *overlay2* driver.

4 Related work

Performance of container environment is not a new concern. Back in 2013, Xavier *et al.* compared the different containerization technologies existing back then (LXC, OpenVZ, VServer) and Xen [13]. They showed that virtualization with Xen was giving really poor performances (notably I/O performances) but a much better isolation. Later, in 2017, Kozhirbayev *et al.* made an updated comparison of container technologies, comparing LXC and Docker, the two main actors in play at that time [10].

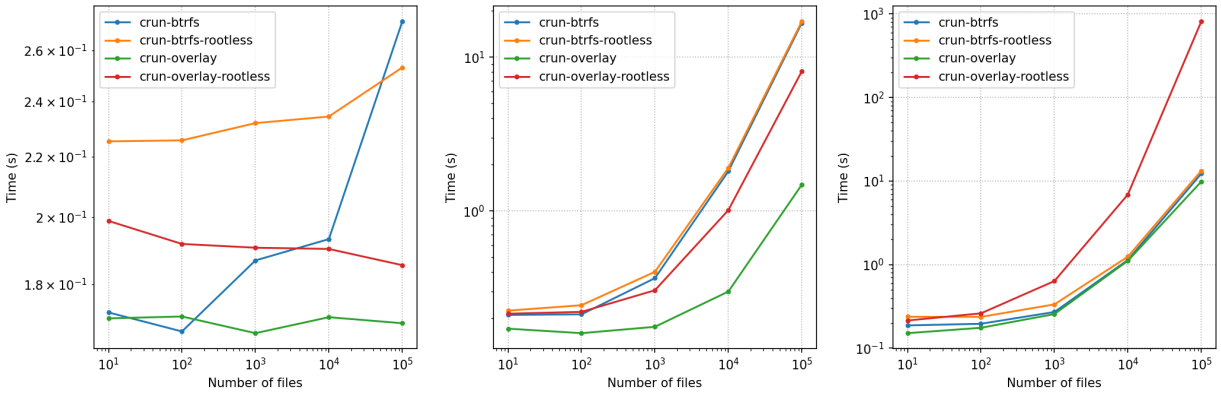

Figure 7. Comparison of rootfull to rootless containers with the I/O benchmarks. Left: Creation time, *I/O read* benchmark. Center: Execution time, *I/O read* benchmark. Right: Execution time, *I/O write* benchmark. All experiments use *crun*.

In 2015, Felter *et al.* also made a point about the performance degradation in terms of I/O in both virtualized and containerized environment comparing them to native performances [6]. They also showed that when using volumes, containerized environment can offer nearly the native performances. We did not elaborate this point as in our considered use cases all resources available to the container must be contained within its isolated file system.

Recent new design proposals for container support, with a focus on the cold start problem, have been proposed recently, such as SOCK [11] and Nuka [12]. Integrating such tools in our benchmarking solution should be an easy task, should these systems become mainstream and compatible with other available technologies.

5 Conclusion

We presented in this paper the design of a benchmarking tool for evaluating the relative performance of container execution environments, with a focus on I/O-related performance. Our ambition was to allow selecting the most appropriate combination of technologies within a controlled environment, rather than in the difficult-to-understand production environment where multiple factors can influence performance outside of the I/O subsystem. In the context of our motivating example, INGInious, the results of this study may help designers select the most cost-effective solution for running automated execution of students' code, and our findings also apply to other domains such as edge clouds or function-as-a-service environments.

While we analyzed a large number of configurations, our benchmarking tool could be augmented to support other technological choices and cover more representative cases. For instance, we did not consider the possibility to use unprivileged containers with LXD on the basis of our use case requirements, and we did not consider other machine virtualization solutions than QEMU and Firecracker for supporting Kata containers. We believe, nonetheless, that adding support for these technologies should be relatively straightforward to implement in our publicly available code [4].

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications *(NSDI)*.

[2] Olivier Bonaventure, Quentin De Coninck, Fabien Duchêne, Anthony Gego, Mathieu Jadin, François Michel, Maxime Piraux, Chantal Poncin, and Olivier Tilmans. 2020. Open educational resources for computer networking. *ACM SIGCOMM CCR* 50, 3 (2020).

[3] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. 2015. Automatic grading of programming exercises in a MOOC using the INGInious platform. *European Stakeholder Summit on experiences and best practices in and around MOOCs* (2015).

[4] Guillaume Everarts de Velp. [n. d.]. Benchmarking tool for container support solutions and dataset. "https://github.com/edvgui/LEPL2990-Benchmark-tool".

[5] Guillaume Everarts de Velp. 2020. Improving the performance of INGInious : a study of modern container technologies (Master thesis). https://dial.uclouvain.be/memoire/ucl/object/thesis:25128

[6] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. 2015. An updated performance comparison of virtual machines and linux containers *(ISPASS)*. IEEE.

[7] Open Stack foundation. [n. d.]. Kata containers: The speed of containers, the security of VMs. "https://katacontainers.io".

[8] Fabian Gand, Ilenia Fronza, Nabil El Ioini, Hamid R Barzegar, and Claus Pahl. 2020. Serverless Container Cluster Management for Lightweight Edge Clouds. *(CLOSER)*.

[9] Kuljeet Kaur, Tanya Dhand, Neeraj Kumar, and Sherali Zeadally. 2017. Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. *IEEE wireless communications* 24, 3 (2017).

[10] Zhanibek Kozhirbayev and Richard O Sinnott. 2017. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems* 68 (2017).

[11] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (ATC)*.

[12] Shijun Qin, Heng Wu, Yuewen Wu, Bowen Yan, Yuanjia Xu, and Wenbo Zhang. 2020. Nuka: A Generic Engine with Millisecond Initialization for Serverless Computing *(IEEE JCC)*.

[13] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. 2013. Performance evaluation of container-based virtualization for high performance computing environments *(EuroMicro PDP)*.