UCLouvain

École polytechnique de Louvain

# Improving the performance of INGInious: A study of modern container technologies

Author: **Guillaume Everarts de Velp**
Supervisor: **Ramin Sadre**
Readers: **Olivier Bonaventure, Anthony Gego**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

# Acknowledgements

A thesis is not a one-man job, this one has been made possible thanks to the intervention of many people. My deepest thanks go to my promoter, Professor Ramin Sadre, who guided me along this long project, providing support and advices and taking all the time required to do so accordingly. A special thanks goes also to Professor Etienne Riviere who showed great interest in my work, and provided valuable advices.

Of course, this project would not even have seen a day if Professor Olivier Bonaventure did not talk me into it initially. It made me discover a new world, for this I am very grateful. Thank you also to Anthony Gego for answering my numerous questions relating to INGInious, to Mathieu Jadin for helping me in my first steps with cgroupv2, to Guillaume Rosinosky for setting me up an access to one of UCL's nuc. Outside UCL, other people helped me too: thank you to Corentin Badot-Bertrand for introducing me to Ansible, thank you to the *Kata Containers* community and to the *containers* organization for helping me with different issues.

And finally, I would like to thank my family and friends for supporting me throughout this project, showing interest and offering support by all times. This is especially addressed to Anne, Nicolas and Tom.

Guillaume Everarts de Velp, June 2020

# Abstract

I studied different containerization solutions available, in different configurations, intending to determine the most appropriate one for INGInious. That solution would be the one that can provide the best responsiveness while meeting the safety requirement that the platform has. Based on my observations, I then present what would be the ideal solution for INGInious, with its performance gain over the best currently available solution and its drawbacks. Finally, I address the performance cost of enforcing the isolation level, as some solutions offer, and the opportunities it brings to the platform.

# Contents

# Chapter 1

# Introduction

In this chapter, I will simply introduce the subject of this master thesis, showing some basic concepts and key aspects related to it.

## 1.1 INGInious

INGInious is a web platform developed by the UCL. It is a tool for automatic correction of programs written by students. To check the validity of a student's code, the latest will be executed, fed with some inputs, and the outputs generated will be checked. It currently relies on Docker containers to provide good isolation between the machine hosting the site and the execution of the student's codes. This allows us to avoid that a problem in a program submitted by a student could have any impact on the platform. Docker also allows managing the resources granted to each code execution. For now, INGInious can meet the demand and provide honest performances and responsiveness. But looking at the growing usage of the platform, we might soon come to a point where we reach the limits of the current implementation.

### 1.1.1 Architecture

INGInious counts four main components:

1. The front-end: the website with which each student interacts when submitting a task.

2. The back-end: a queue of all the tasks that need to be graded.

3. The docker agent: responsible for the container assignment to the pending tasks when resources are available.

4. The docker containers: one for the student code and one for the teacher tests evaluating the student's code behaviour.

The journey of a task submitted on INGInious is represented on Figure 1.1.

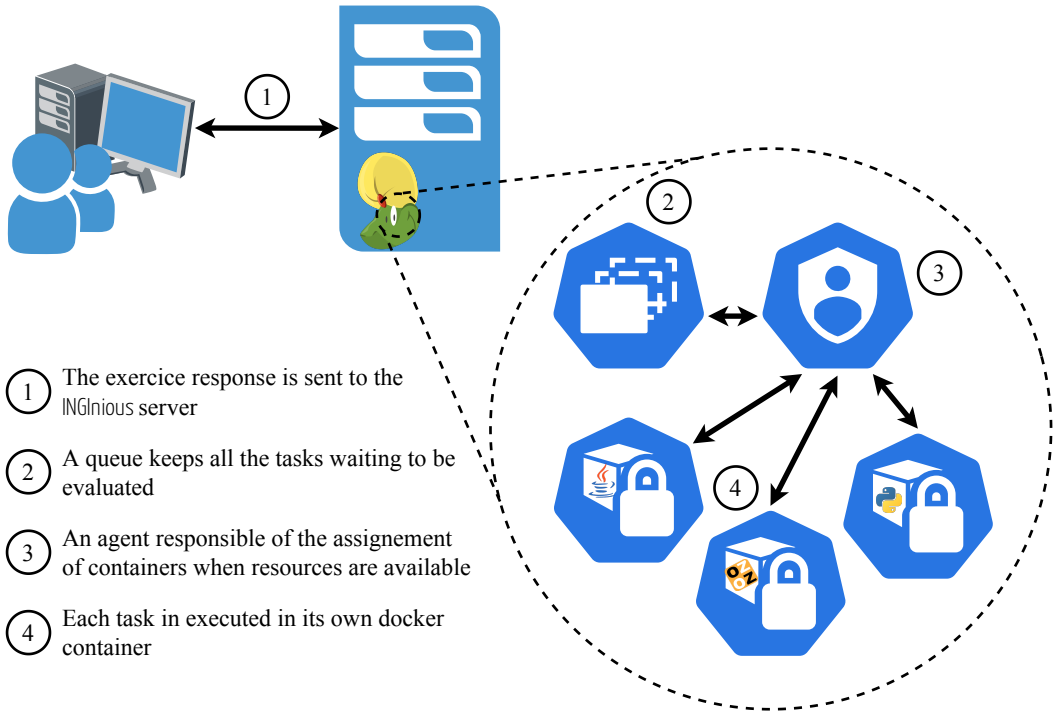


Figure 1.1: INGInious global architecture

### 1.1.2 Key features

The key features of INGInious, that allows it to meet the requirement of a code grading platform, are the following:

- Isolation between the student's code and the platform.
- Resource limitation (CPU, RAM, Network, Pids) for the student's code execution.
- Modularity and versatility regarding the tasks that INGInious can correct. Multiple programming languages are supported, and new ones can easily be added.

### 1.1.3 Bottlenecks

When a task is submitted, we count five delays before the answer can be delivered to the student:

- The sending time: the time it takes for the task to be sent to the back-end.
- The waiting time: the time the task will spend in the queue, waiting for an available container.
- The booting time: the time it takes to the container to boot and be ready to evaluate the task.
- The grading time: the time it takes for the code to run and for the teacher's container to grade it.
- The response time: the time to send the response back to the student.

For the first and the last one, supposing that the machine hosting the website is not overwhelmed, the delay depends entirely on the network. It is a bit out of our hands. The waiting time is directly related to the current load on the platform. This is more a symptom of the server overwhelming than its cause, it could be directly solved by using a scaling strategy (see section 1.2). And then we come to the booting time and the grading time, which directly depends on the containerization technology used and on the hardware performances, this is where we are going to try to see if we can improve things in this thesis.

## 1.2 Scaling

Currently, the resources provided to INGInious vary, depending on the load that the platform is expected to be facing. Typically, when the grading of an exam is done by INGInious, the platform is scaled up, and during the holidays it is scaled down. When it comes to scaling, two strategies can be used; vertical scaling and horizontal scaling.

**Vertical scaling** consists of adding more resources on a single machine, to improve its performances when needed. For example, when the number of tasks arriving at the server grows, we could increase the number of virtual Cores allocated to the Virtual Machine (VM) hosting the platform to be able to treat more of them concurrently. If the size of the waiting queue is increasing, we might want to make more RAM available.

**Horizontal scaling** consists of sharing the workload across multiple machines so that each machine can handle a small part of it. This is a solution widely used nowadays as it allows to scale up virtually indefinitely, which is not the case with vertical scally where we depend on the maximum capacity of the hardware. This requires rethinking the architecture of the platform globally.

## 1.3 Intentions

This master thesis aims at improving INGInious, regarding its performances and its scalability. To do so, I will search and compare different containerization technologies and configurations that could be used instead of Docker's current configuration. The goal is to find if an alternative could decrease the booting time (and the grading time) without losing any of the key features of the platform. If such an alternative is found and proven to be worth the change, INGInious could then be refreshed with it.

Based on my research, I will answer to those three main questions:

1. Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it? Do some solutions involve tradeoffs in terms of maturity, support or maintainability?

2. Could there be a solution tailor-made for the specific case of INGInious? What would it be? What would it cost to use it?

3. What would be the cost of providing stronger/safer isolation to the containers used by INGInious? Which opportunities could it bring?

# Chapter 2

# Background knowledge

In this chapter, I will put some basis, and explain some concepts related to my work. Reading this chapter should allow the reader to understand what I did in this work, supposing he is already familiar with Linux.

## 2.1 Virtualisation, Containerisation, Runtime isolation

We always have the same goal: being able to execute some code, with no interaction with someone else's one. As of course, we cannot use one bare-metal machine for each application we want to run, some mechanism has to be used to provide isolation between coexisting processes on the same physical machine.

We can distinguish three level of isolation: Virtualisation, Containerisation and Runtime isolation (Figure 2.1).

### 2.1.1 Virtualisation

The isolation layer is located either between the hardware and the guest OS (hypervisor of type 1) or between virtualization of the hardware and the guest OS, with a hypervisor (type 2) running in another OS. In both cases, the guest OS will run as if it was on a bare-metal machine, executing its kernel, managing its drivers, its file system, etc. This is the stronger isolation we can get. Cloud providers can rent you some Virtual Machines, this is what we call IaaS (Infrastructure as a Service). You can use many different hypervisor solutions. Usually, Cloud providers have their very own solution.

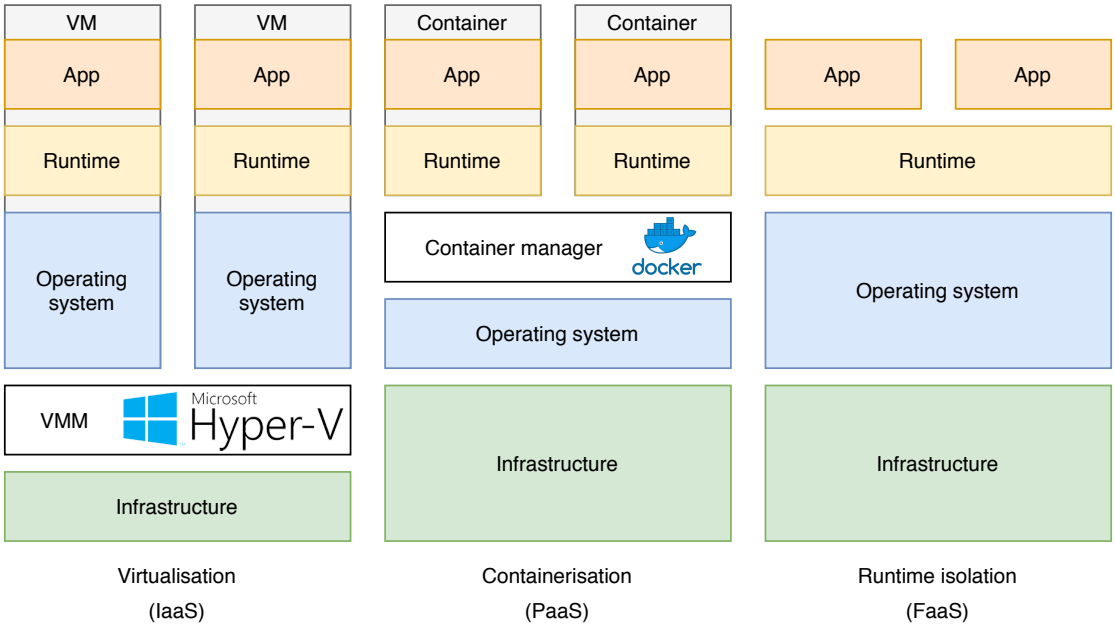


Figure 2.1: Different isolation level for an application.

### 2.1.2 Containerisation

Here we go one level higher: the isolation layer is located between the OS and the guest application. This isolation mechanism is called a container, and is a set of processes, running in common namespaces, with a root filesystem different from the host. Containers share the same kernel as the host though, which means that a process running in a container can be seen from the host. Containers belong to the world of PaaS (Platform as a Service), a system where Cloud providers offer you to run your application (that you provide under a containerized form) without having to worry about all the infrastructure that needs to be deployed along with it, and sometimes even with some great scalability mechanism support.

### 2.1.3 Runtime isolation

This is the highest (weakest) level of isolation you can get. The isolation is provided by the runtime on which the application is running (ex: The JVM for a Java program, a Python environment, ...). Multiple guest applications share the same OS, file system, without any mechanism to hide it from them. Cloud providers propose such service under the name of FaaS (Function as a Service). For this, you can provide a small program, that you want to be executed on demand. Cloud providers usually allocate a container to this program, to provide safer isolation,

but will execute multiple instances of this program in the same containers, to avoid the overhead of creating a new container each time. The type of programs you usually run in these is computationally intensive parts of your application. These need to have low response time and great scalability (for example resizing of user-uploaded images). This model is often referenced as the Lambda model (from Amazon Lambda service) or as serverless computing.

### 2.1.4 INGInious specific case

The case of INGInious is a special one. The type of the workload caused by the evaluation of the tasks (fast response time, varying demand, potentially big computation, logic independent of the core application) would suggest using a serverless strategy, but the isolation requirement are those of PaaS (as the various inputs of the functions are code of student to safely execute, isolated from one another). Currently, the application relies on IaaS, running in multiple VMs where are hosted the core application and multiple docker agents.

## 2.2 Containers

The isolation being the most important requirement, we will then look into containerization solutions: presenting the main concepts behind containerization, the global container ecosystem, the current solution that INGInious uses and the alternatives we have.

### 2.2.1 Core concepts

Containers rely on three components: namespaces, control groups and chroot.

**Namespaces** are a feature of the Linux kernel since 2002. A namespace can be associated with a context, which is a partition of all the resources of a system that a set of processes has access to, while other processes cannot. Those sets of resources can be of seven kinds:

- **mnt** (Mount): This controls the mount points. Processes can only have access to the mount point of their namespace.
- **pid** (Process ID): When processes are in a pid namespace, the latest hides from them all the processes that are not in it. The assignment of pid as seen by the processes inside the namespace is independent of the rest of the system. The first process in the namespace will be the pid 1, the second, 2,

etc. From a parent pid namespace though, those processes can be seen, and have another pid (obviously pid 1 and 2 are already taken).

- **net** (Network): This provides a virtualized network stack.
- **ipc** (Interprocess Communication): This allows processes of the same namespace to communicate with one another, for example by sharing some memory.
- **uts** (Hostname): This allows to have different hostnames on the same machine, each hostname being considered as unique by the processes of its namespace.
- **user** (User ID): This allows to change the user id in a namespace. This way you could have a user root in the namespace, which is unprivileged outside of it.
- **cgroup** (Control group): This allows to change the root cgroup directory. This virtualizes how process's cgroups are viewed.

When a Linux machine starts, it initiates one namespace of each type in which all the processes run. The processes can then choose to create and switch namespaces.

**Control groups** are a feature of the Linux kernel that allows limiting and controlling the resources allocated to some processes. You can for example control the CPU usage, the memory consumption, the io... Recently (since kernel 4.5) a new version of cgroups (cgroups v2) appeared, which comes to tackle the flaws of the original implementation while keeping all of its functionalities. The main change is the use of a new unified hierarchy to manage the control groups, where all the resources are centralized and where the allocation of some resources is done by adding a sub-resource-group in the hierarchy the current user has access to. However, the adoption of the newer version is a process and takes time. Still now, many applications use the original version of cgroup.

**Chroot** allows us to change the root of the root filesystem for some processes. For example, we could create a directory `/tmp/myroot/` which contains all of the usual directories present in the original root folder (`/`) and set this as the new apparent root for the chosen processes. This is not a complete sandbox, and not real isolation on its own, files from outside of the chroot could still be accessed.

### 2.2.2 Storage drivers

When it comes to handling a container file system, different solutions can be used. The goal of each is to provide the most efficient writable root directory (`/`) for each

container but keeping each container unaffected by the modification done in the other containers.

To do so, three main strategies can be used:

- **Deep copy**: for each container, the whole image is copied during the creation of the container. This is simple but gets excessively slow as the file system size increases.
- **File based copy on write**: for each container, will be copied only the files that are edited during the container life cycle. This is more complex but gets more efficient as the container's size grows.
- **Block based copy on write**: for each container, will be copied only the blocks (in the filesystem) that are edited during the container life cycle. This is even more complex but get more efficient as some small part of big files are edited.

Containers are a specific kind of workload in the sense that a lot of information, data, is redundant in different containers. For example, for a simple application, we could use several containers with different responsibilities and tools embedded in it, but all based on the same Alpine image. This brought a new space problem, as we want to avoid duplicating too much data. To face this, **union filesystems** are used, along with layered container images. This means that different container images but with the same basis will share the common part of their file system, avoiding the need to duplicate it. It differs from the file system deduplication mechanism by the level at which the action is taken. For deduplication, when a block is written on disk, we check if similar data is not already located somewhere. Whereas for a union file system, the common base image will be read-only, and when an attempt to modify a file is made, a union mount will be made with a copy of the file on top of the original one, so that the common base stays preserved. In this work we will consider two union filesystems: AUFS and the newer OverlayFS, on which are based respectively aufs and overlay storage drivers.

The storage of a container has to be backed by a file system, which will eventually already provide some interesting mechanisms for the container. Those four main filesystems used are:

- **BTRFS**, presented in 2013, [19] is a general-purpose file system, based on copy-on-write and with efficient snapshots capabilities and strong data integrity.

– **zFS** is similar to BTRFS (but was there before), but handles some mechanisms differently, like the management of blocks (blocks vs. extends[1]), snapshots (birth-times vs reference-counting[2]), ... It also supports deduplication, which can be useful for systems where a lot of data is redundant and disk space is valuable.

– **xFS** is a high-performance journaling file system, with the capability of managing cluster of drives and combining them in partitions. It is especially good for parallel IO operations. sIt manages to deliver better performances and availability than other solutions available at the time it was created (1993). [20] It has no focus on any copy-on-write mechanism as the first two though.

– **ext4** was meant to be the replacement of ext3, as the "Linux Filesystem" when it was presented in 2007. [16] It aims at providing a good balance between scalability, reliability, performance and stability. It does not provide any of the fancy features of the previous file system.

The Table 2.1 presents a short summary of the different storage driver solutions available today that we can use with container managers.

| **Storage driver** | **C-o-W**[3] | **FS**[4] |
|---|---|---|
| `overlay` | file based | ext4, xFS |
| `aufs` | file based | ext4, xFS |
| `devicemapper` | block based | direct-lvm[5] |
| `btrfs` | block based | BTRFS |
| `zfs` | block based | zFS |
| `vfs` | no | any |
| `directory` | no | any |
| `lvm` | block based | ext4 |

Table 2.1: Summary of storage driver solutions.

[1] zFS can have blocks to up to 128KB, while BTRFS will only use the extend strategy (point to the next block).

[2] When doing snapshots, zFS will store on the block the time at which a new use of the block has been added, while Btrfs handles it with a reference-counting mechanism.

[3] Copy-on-write

[4] Backing file system

[5] Note that this is a logical volume manager, not a file system, which uses in our case the xFS file system

### 2.2.3 Container runtime

A container runtime is a tool that creates containers, executes processes in it, and deletes dead containers. It will have the responsibility to create the namespaces, change the root directory, and attach processes to a control group. The most common container runtime nowadays is `runc`, created by the Open Container Project.

### 2.2.4 Container manager

A container manager will use the container runtime, to provide a "user-friendly" interface to manage containers. It has the responsibility to set up the network interfaces, to provide the image of the containers and any Copy-on-write mechanism that could go along with it. Some of them offer the possibility to create custom images, to create pods, or swarm, which are entities of multiple interconnected containers.

### 2.2.5 Rootless containers

As we saw previously, creating a container requires to create a bunch of namespaces, and launch some processes in it, limiting their resources with control groups. If you do not do this manually, all of this is taken care of by the container runtime. There are two modes in which you can run containers:

**Rootfull** This is the default choice when using Docker, the processes launched in the container are owned by root, and the container runtime is executed as root as well. The container manager also needs root permissions then. It means that either the user willing to launch a container has to be root, or some trick has to be used to allow the user to gain the privileged required to launch the containers. Docker does it by allowing any user member of the group `docker` to send requests to the daemon, which runs as root and can do the required operations.

**Rootless** The processes launched in the container are not owned by root (but are mapped to the root uid inside the container) and the container runtime is not executed as root either. This requires the use of the second-generation control group to have resource management capabilities, as the unified hierarchy allows any user to create more sub-resource-group, to manage the resource it has access to. It means that we do not need to use a daemon or any other trick to give a user permission to do "root stuff" on the host.

There is also an intermediate solution, which maps the root user of the container to an unprivileged user on the host, as in the rootless case. It means that processes

running as root in the container will not be root on the host, which is already better, but the container runtime still runs as root.

## 2.3 Ecosystem overview

A small overview of the current container ecosystem can be found in Figure 2.2. Note that solutions like Kubernetes[6] which are more oriented towards hosting and continuous deployment of container-based applications than to single container provisioning are not presented here[7].

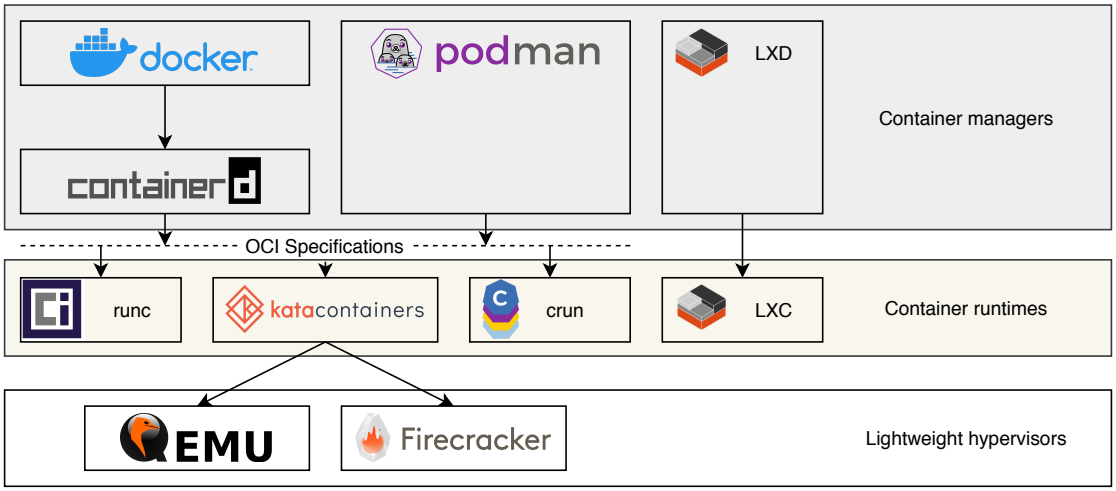


Figure 2.2: Small overview of the current container ecosystem

**OCI** (Open Container Initiative) is "an open governance structure for the express purpose of creating open industry standards around container formats and runtime." [3] They currently have two specifications: the format of the image that has to be given to an OCI compatible runtime, and the commands to interact with such runtime.

[6]Kubernetes is "an open-source system for automating deployment, scaling, and management of containerized applications" [5]

[7]A more detailed overview for that kind of applications is provided by Containerd at the following address: https://containerd.io/img/architecture.png

## 2.4 Containerisation solutions

**Docker** [17] is today probably the most known container manager solution for containerization. Since its apparition in 2014, its interest in the PaaS sector has not ceased to grow. It consists of a daemon, running on the host, which allows us to easily manage different containers. It relies on Containerd, "An industry-standard container runtime with an emphasis on simplicity, robustness and portability" [1], which is a more complete container runtime than what I presented before, with embedded network management, storage driver management, and other mechanisms. Containerd is not meant to be used standalone though, which is why we still use Docker. By default, Docker uses `runc` as container runtime but is compatible with any runtime that fulfills the OCI [3] requirements.

The main advantages of Docker are its simplicity of use, its production-grade quality, the huge fleet of ready-to-run containers publicly available and a lot of useful tools (like `docker-swarm` that come along with it). This is the solution currently used by INGInious.

"**Podman** is a daemonless container engine for developing, managing, and running OCI [3] Containers on your Linux System." [6] Podman presents itself as a viable true alternative to Docker. Its main difference is the fact that it runs daemonless, it runs containers as detached child processes and containers can be rootless by default (while it is only a recent feature coming up to Docker). It can also manage pods, which are groups of containers deployed on the same machine. Its default runtime is `runc` as well. Podman is an Open-Source project and is still growing a lot, the latest version at this day (2020-05-30) is v1.9.3, released 8 days ago, and already 797 commits have been done since then!

"**LXC** is a userspace interface for the Linux kernel containment features." [7] This solution is developed and maintained by Canonical Ltd. Docker used to be based on LXC until it created its own execution environment. LXC came out recently with a new solution; LXD, which offers about the same features as Docker does. A bunch of ready to run images publicly available, and a nice command-line interface to interact with containers. The only missing feature of LXD compared to Docker, regarding our use case, is the possibility to launch a container with a command and stop it when it is finished. LXC starts a complete init process for each container, in which you can then come and execute your command.

**Linux-VServer** [2] is a bit of a special one. No namespaces are used to provide isolation here, they rely on contexts to separate multiple virtualized systems. They have modified the Linux kernel to provide some utilities like process isolation,

network isolation and CPU isolation. The main advantage of this approach is scalability as it can run a large number of containers, the drawback is the portability. It requires a modified version of the kernel and it also misses some features like checkpoint (save the current state of a running container) and resumes (resume a paused container) that we can have with more classical containerization and real virtualization. This is a project much older than Docker and its contemporaries and it differs in goal. It is not appropriate for the INGInious workload. This is an experience more similar to what classical VM provides, where you will set up multiple applications in one environment. The container is not built for one single purpose.

**OpenVZ** [4] is another container-based virtualisation solution. But this time it uses namespaces to provide it. It also has its way of dealing with system resources management, more complete than cgroup. Even though this is more close to what runc and LXC provide in terms of isolation, this is still not the kind of experience appropriate for INGInious use case. This is designed, as for Linux-VServer, for a more complete server experience.

**Kata Containers** is not another container manager. It is an OCI compatible runtime. With the specificity that it does not run mainstream containers as `runc`, but lightweight virtual machines, using KVM virtualization and with Qemu (originally), Firecracker or Cloud Hypervisor (still in its early days) as the hypervisor. Those two first currently available hypervisors are not equivalent though, as the original one has more features (like devices assignment) and the second one has a lower memory footprint and smaller attack surface (smaller codebase). They put forward four features of their solution:

- **Security**: Thanks to their virtualization solution, each runtime has its own kernel, with real network, i/o and memory isolation.
- **Compatibility**: They support industry standards, as the OCI [3] and legacy virtualization technologies.
- **Performance**: Their performances are consistent with classical containerization solutions.
- **Simplicity**: They eliminate the need to have a virtual machine dedicated to host containers.

**crun** is a complete equivalent of `runc`, yet another OCI compatible runtime, but this one is implemented in C, which give it a small performance advantage over

`runc`, implemented in GO. `crun` has also full support for cgroupv2, which is still lacking for `runc` for the time being (2020-04-22).

With the variaty of container manager, container runtimes and storage drivers, we can compose many different solutions. Those are presented later, in Chapter 4. Here is a quick summary of the different elements I presented here:

| **Name** | **Tool type** | **Appropriate**[8] | **OCI**[9] |
|---|---|---|---|
| Docker | Container manager | Y | Y |
| Podman | Container manager | Y | Y |
| LXD | Container manager | Y | N |
| Linux-VServer | Container *hypervisor* | N | N |
| OpenVZ | Container *hypervisor* | N | N |
| LXC | Container runtime | Y | N |
| runc | Container runtime | Y | Y |
| crun | Container runtime | Y | Y |
| Kata Containers | Container runtime | Y | Y |

[1]Could be use with INGInious
[2]Comply with the OCI specifications

# Chapter 3

# Related work

This is not the first time that researchers try to compare different containerization solutions. In this chapter, I will present other works that have been done in that matter and show how what I am doing is different. I will end up by presenting some improving solutions that some people tried to bring to the container solution panel.

## 3.1 Performance studies

**2013** *Performance evaluation of container-based virtualization for high-performance computing environments* [21]: They compared a native experience to LXC containers, OpenVZ, VServer and Xen. They showed that virtualization (with Xen) was giving really poor performances compared to the others (which give nearly-native experience), in terms of I/O, memory bandwidth, and even network latency and bandwidth. What they have to grant to virtualization though, is the better isolation it provides, as an application running in a VM could run almost seamlessly when the system is stressed (in another VM) with varying tests. While for other solutions the impact can vary a lot, even making it impossible for the application to start in some case.

**2014** *Virtualization vs containerization to support paas* [11]: Virtual Machines have been kept away from the PaaS world, because of the greater overhead that Virtualization has on booting time and resources consumptions of the isolated system created. This paper made a detailed point about it, emphasizing three points that needed to be improved for the next generation of containers: Security, OS Independence and Standardization. The last one meaning that containers runtime should establish some common requirements, come up with a common container format. We have seen this appear since then, under the OCI (Open Container Initiative) specifications. They also showed that

a trade-off with the performances (mostly I/O) of the application running isolated is much more evident with virtualization than with containerization.

**2015** *An updated performance comparison of virtual machines and Linux containers* [12]: In this paper, they show the overhead that both containerization and virtualization have, compared to a bare-metal system. The most important point is the I/O comparison, where they show that virtualization (here with KVM) provides much lower performances than containerization (with docker, using aufs). And docker performances can even be improved to nearly meet the native ones when using volumes to store the solicited files.

**2017** *Performance overhead comparison between hypervisor and container based virtualization* [14]: In this paper, they showed that even though container-based virtualization (with Docker) is more lightweight than real virtualization, its performances are not always better (they are worse when trying to read or write single bytes to disk). And they plan to compare it into more details later.

**2017** *A performance comparison of container-based technologies for the cloud* [13]: They compared here two different containerization solutions (Docker and LXC) against the native experience with a focus on CPU usage, memory consumption, network and I/O performances. Once again, overheads are detected for I/O operations, where high demand for input-output operations have a greater latency that a native experience would have.

## 3.2 Improvement studies

### 3.2.1 SAND

SAND [9] aims at improving the performance of serverless computing with two techniques: application-level sandboxing, and a hierarchical message bus:

**Application Sandboxing**: The key idea here is to differentiate multiple executions of different functions with multiple executions of the same function. In the first case, as usual, a new container is launched on the new function demand and the function is executed inside of it. In the second case, instead of launching a new container with the same configuration as an already running one, they only fork a process inside the running container to deal with the new demand, which is much faster than creating a new container. This would not be good for us as we would lose the isolation between student code execution, which is not desirable.

**Hierarchical Message Queuing**: The goal to this is to facilitate communication between functions that interact with one another (i.e. the output of a function is the input of another one). To do so they use a two-level communication bus: global and local. Global means that communication is made between different functions on different hosts, while local means different functions on the same host. As accessing the local bus is much faster than the global one, it can decrease latency for functions on the same host. In the case of INGInious, student and teacher containers always run on the same host and communicate though the mean of file descriptors. Therefore, this is not something for us.

### 3.2.2 SOCK

"Sock (roughly for serverless-optimized containers), [is] a special-purpose container system with two goals: (1) *low-latency invocation* for Python handlers that import libraries and (2) *efficient sandbox initialization* so that individual workers can achieve high ready-state throughput." [18] The final product created here is not something we could use for INGInious, it is a serverless solution (based on the Lambda model), targeting specifically python applications. Though, in the process of creating this solution, they started from containers and deconstructed their performances, identifying their bottlenecks. And from this we can :

- Bind mounting is twice as fast as AUFS. Even though AUFS (used by Docker) has a useful copy-on-write capability, we do not need to write to most of our files in our case, so using a read-only bind mounting for those files could allow us to avoid copying all file before start-up, while still being able to edit the files we want to, avoiding the cost of copying the file to a higher layer.

- Network namespaces creation and clean-up are costly, due to a single lock shared across all network namespaces. We might gain some performances buy not adding network interfaces to containers that do not need it. This is already done by the docker agent of INGInious.

- Reusing a cgroup is at least twice as fast as creating a new one each time. We could keep a pool of initialized cgroups, and only change the current container it controls.

### 3.2.3 LightVM

LigthVM is a complete redesign of Xen's toolstack which tried to bring some container's characteristics to VMs. Such as fast instantiation (small start-up time) and high instance density capability (high number of instances running in parallel

on the same machine). [15] Xen is a Type-1[1] hypervisor presented in 2003 with really low virtualization overheads and high hosting capacity, allowing a machine to host up to a 100 guest OS. [10]

LightVM achieves such container's like performances by:

- reducing the image size and the memory footprint of virtual machines. They do so by including in the VM only what is necessary to the application that is meant to be executed in it.
- introducing noxs (no XenStore[2]), a new implementation of Xen, without XenStore (which was a real bottleneck for fast instantiation of multiple VMs).
- splitting the Xen's toolstack into what can be run before the VM creation and what as to be done during it. Allowing to pre-initialize VMs.
- replacing the Hotplug Script by xendevd, a binary daemon that can execute pre-defined setup more efficiently.

The paper presents four use cases with this solution, one of them being more interesting for us: lightweight computation services, for which they rely on Minipython unikernel, to run computations written in Python, as a Faas could propose to do. This would still be hard to use for INGInious, for the same reason as for SOCK (§3.2.2).

### 3.2.4 Firecracker

Firecracker is a new VMM (hypervisor) created specifically for serverless and container applications. [8] This is a solution provided by AWS, that very recently got deployed for two of their web services: Lambda (Faas) and Fargate (Paas). We are getting here the good isolation of virtual machines and nearly as good performances and low overhead of containers. Firecracker is based on KVM and provides minimal virtual machines (MicroVMs). The configuration is done through a REST API. Device emulation is available for disks, networking and serial console. The network can be limited and so can disk throughput and request rate. If proven to be easily usable in INGInious's case, this would provide a better alternative to classical containerization regarding security.

[1] A Type-1 hypervisor is a hypervisor that runs on a bare-metal machine, without an additional host.

[2] The XenStore is a registry containing informations about running VMs and their devices.

### 3.2.5 Summary

In this section, on Table 3.1, are quickly reminded the several new solutions we explored in this chapter.

| **Name** | **Pub.**[3] | **Update**[4] | **Open-Source** | **Com.**[5] | **Isol.**[6] |
|---|---|---|---|---|---|
| SAND [9] | 2018 | ? | ? | No | Cont. |
| SOCK [18] | 2018 | ? | ? | No | Cont. or Runt. |
| LigthVM [15] | 2017 | 2017 | Yes | No | Virt. |
| Firecracker [8] | 2019 | 2020 | Yes | Yes | Virt. |

Table 3.1: Summary table of the different solutions explored in this chapter.

## 3.3 My master thesis

In this chapter, I have presented an overview of the work of other people in the field of containerization, with a focus on performance improvement or measurement. Nevertheless, these solutions cannot be used without testing and some more questions are rising which were not answered in the consulted literature.

First of all, most of those performance measurements are quite old, and none of them considered all of the solutions that we could use for INGInious's case. For example, Podman is not mentioned once (which is normal as version 1.0.0 of Podman was only released in January 2019). Neither is the interesting case of Kata Containers. The apparition of a lightweight hypervisor like Firecracker is a game-changer, but the paper did not provide any performance comparison with containerization solutions.

Secondly, none of the research here had a real focus on the time required for each solution to provide a ready to run environment. We have got a lot a CPU and memory overhead analysis, but those are not our main concerns.

Therefore, the direction I will take in this report is trying to answer to the question: "What would be the most appropriate containerization (or virtualization) solution for INGInious use case?", or in other words, how to get the shortest

[1]The year of the publication of the project

[2]The year of the last update on the project

[3]Whether the solution has a "commercial grade" quality

[4]The type of isolation of the solution; cont. for containerization, virt. for virtualization, runt. for runtime isolation

instantiation time, along with the best theoretical isolation possible and as low overall performance overhead as we can get.

# Chapter 4

# Testing

As previously said, the goal of this thesis will be to identify if some improvements could be made to INGInious, by changing the containerization solution in use. To do this, I have created some tests and listed different possible configurations that should face those tests. Based on the results of those tests, I should be able to determine whether or not we can bring improvement to INGInious.

## 4.1 Setup

### 4.1.1 Testing environment

To be fair in the result comparison, all the different configurations have to be tested with as much in common as possible. This will allow us to determine the influence of varying parameters in those configurations. To do so, the same machine will be used for all the tests, with its configuration being updated accordingly to the requirements of each solution.

The testing environment used for the final results presented in this work is an old laptop, refurbished with the following configuration:

| | |
|---|---|
| **processor** | Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz (x4) |
| **memory** | 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 |

The choice of going with a bare-metal setup was not the original one. But the early results I got showed that some solutions (especially the ones using virtualization) did perform way worse when running inside of a VM[1]. To get the

[1]Or in some cases it mysteriously just doesn't work, cf. https://github.com/kata-containers/runtime/issues/2587

best out of them, the setup has then been moved out of VMs.

**Note about cpu:** The cpu is a little bit weak, and not representative of what a real production server would use to host a container workload. Unfortunately, this was all I had available, and it will have to do. This means that the results presented in this work can still be a little bit less performant than what we could get typically in the cloud.

**Note about memory:** This is also typically less than what we would have in the cloud, but this is just enough for our case, as we only execute one container at a time, with limited access to memory.

**Note about storage:** The storage type is important, as it will highly influence the performance of my tests, given the high solicitation of I/O they require. Using an SSD is therefore essential, both to avoid the need to run the tests for several weeks, and to have measurements more in pair with what cloud provider proposes.

**Note about operating system:** The main reason I have chosen Ubuntu is that I am familiar with it, and it made things easier for me to set up. Also, Ubuntu is often (if not always) an option that cloud providers offer when it comes to choosing the OS to install in a VM. Also note that Ubuntu server 20.04 was used for the tests requiring cgroupv2 (see the Table of configurations later) as too many issues were faced with Ubuntu 18.04.

### 4.1.2 Configuration of the environment

As a lot of different solutions are going to be tested, and all of them with their requirements (sometimes conflicting with the ones of other solutions), some work had to be done to make it easier for someone to replicate them. The configuration of the environment is done with Ansible[2] playbooks, with one playbook for each solution to test.

All the different configurations are presented in section 4.2.1. The different playbooks can be found in the repository of this project at this address: https://github.com/edvgui/LEPL2990-Benchmark-tool/tree/master/ops.

[2]https://docs.ansible.com/

## 4.2 Tests

As we already said previously, time is an important aspect of the experience INGInious provides to its users. The time for a task to be evaluated and the time before this task is evaluated. To improve those two, in those tests we will focus on the booting time of containers and their IO handling (still related to the time overhead that some solution could have compared to another).

While we are at it, we will also verify if some solutions handle network much poorly than others. Each test is presented in more detail in section 4.2.2.

### 4.2.1 Candidates

A candidate solution is a combinaison of different elements, variabilities, choices to make. We will consider here those ones:

- **Container manager**: *Docker*, *Podman*, *LXD*, those are the different user friendly solution that can be used to manage container, and that we will compare here.
- **Container runtime**: *runc*, *crun*, *LXC*, *Kata containers* (Qemu or Firecracker), the (less user-friendly) container runtimes, taking care of all the isolation that a container require.
- **Control group**: *cgroup* and *cgroupv2*, there is no real choice to make here: cgroupv2 is the successor of cgroup, and should be used, when supported by the other elements of the configuration.
- **Storage driver**: *aufs*, *btrfs*, *devicemapper*, *directory*, *overlay*, *vfs*, *zfs*, *lvm*, those are the different strategies that can be used to manage the file system of the container.
- **Base container image**: *Alpine* because it is the default choice when it comes to conceive containerized applications today or *Centos* because it is the current choice made by INGInious.
- **Rootless container**: Whether if we run the container in rootless mode.

Though, we cannot compose a candidate solution with one element of each category, randomly picked. Some elements of the solution might only be usable with some of the possibilities for a variability. The different constraints that we have are listed here:

- Docker does not support cgroupv2 yet (actually the support has to be added by containerd).

– Docker only supports aufs (deprecated), btrfs, devicemapper (until Docker 18.06), overlay, vfs, zfs as storage driver.

– LXD does not support cgroupv2 yet.

– LXD only supports LXC as runtime, and LXC is only supported by LXD.

– LXD only supports btrfs, zfs, directory (dir) and lvm as storage driver.

– Kata Container with Firecracker only supports devicemapper as a storage driver.

– zfs, lvm, aufs cannot be used with rootless containers.

– cgroup (v1) cannot be used with rootless containers.

– runc does not support cgroupv2 yet.

The list of different candidate solutions we can compare is presented in the table below. Each container will be launched with a CPU limitation of one single-core, and memory limitation of 1GB. This is more than enough for each test that will be done. A network interface will be only added to the container if it needs it for the specific test.

| # | **Manager** | **Image** | **Storage** | **Cgroup** | **Runtime** | **Rootless** |
|---|---|---|---|---|---|---|
| 1 | Docker | Alpine | aufs | v1 | runc | No |
| 2 | Docker | Alpine | btrfs | v1 | runc | No |
| 3 | Docker | Alpine | devicemapper | v1 | runc | No |
| 4 | Docker | Alpine | overlay2 | v1 | runc | No |
| 5 | Docker | Alpine | vfs | v1 | runc | No |
| 6 | Docker | Alpine | zfs | v1 | runc | No |
| 7 | Docker | Alpine | aufs | v1 | crun | No |
| 8 | Docker | Alpine | btrfs | v1 | crun | No |
| 9 | Docker | Alpine | devicemapper | v1 | crun | No |
| 10 | Docker | Alpine | overlay2 | v1 | crun | No |
| 11 | Docker | Alpine | vfs | v1 | crun | No |
| 12 | Docker | Alpine | zfs | v1 | crun | No |
| 13 | Docker | Alpine | aufs | v1 | kata-runtime[3] | No |
| 14 | Docker | Alpine | btrfs | v1 | kata-runtime | No |

[3]Default Kata Containers runtime, using qemu as hypervisor.

| | | | | | | |
|---|---|---|---|---|---|---|
| 15 | Docker | Alpine | devicemapper | v1 | kata-runtime | No |
| 16 | Docker | Alpine | overlay2 | v1 | kata-runtime | No |
| 17 | Docker | Alpine | vfs | v1 | kata-runtime | No |
| 18 | Docker | Alpine | zfs | v1 | kata-runtime | No |
| 19 | Docker | Alpine | devicemapper | v1 | kata-fc[4] | No |
| 20 | Docker | Centos | aufs | v1 | runc | No |
| 21 | Docker | Centos | btrfs | v1 | runc | No |
| 22 | Docker | Centos | devicemapper | v1 | runc | No |
| 23 | Docker | Centos | overlay2 | v1 | runc | No |
| 24 | Docker | Centos | vfs | v1 | runc | No |
| 25 | Docker | Centos | zfs | v1 | runc | No |
| 26 | Docker | Centos | aufs | v1 | crun | No |
| 27 | Docker | Centos | btrfs | v1 | crun | No |
| 28 | Docker | Centos | devicemapper | v1 | crun | No |
| 29 | Docker | Centos | overlay2 | v1 | crun | No |
| 30 | Docker | Centos | vfs | v1 | crun | No |
| 31 | Docker | Centos | zfs | v1 | crun | No |
| 32 | Docker | Centos | aufs | v1 | kata-runtime | No |
| 33 | Docker | Centos | btrfs | v1 | kata-runtime | No |
| 34 | Docker | Centos | devicemapper | v1 | kata-runtime | No |
| 35 | Docker | Centos | overlay2 | v1 | kata-runtime | No |
| 36 | Docker | Centos | vfs | v1 | kata-runtime | No |
| 37 | Docker | Centos | zfs | v1 | kata-runtime | No |
| 38 | Docker | Centos | devicemapper | v1 | kata-fc | No |
| 39 | LXD | Alpine | btrfs | v1 | LXC | No |
| 40 | LXD | Alpine | zfs | v1 | LXC | No |
| 41 | LXD | Alpine | directory | v1 | LXC | No |
| 42 | LXD | Alpine | lvm | v1 | LXC | No |
| 43 | LXD | Centos | btrfs | v1 | LXC | No |
| 44 | LXD | Centos | zfs | v1 | LXC | No |
| 45 | LXD | Centos | directory | v1 | LXC | No |
| 46 | LXD | Centos | lvm | v1 | LXC | No |
| 47 | Podman | Alpine | aufs | v1 | runc | No |
| 48 | Podman | Alpine | btrfs | v1 | runc | No |
| 49 | Podman | Alpine | overlay | v1 | runc | No |
| 50 | Podman | Alpine | vfs | v1 | runc | No |
| 51 | Podman | Alpine | zfs | v1 | runc | No |

[4]Kata Containers runtime, using Firecracker as hypervisor

| | | | | | | |
|---|---|---|---|---|---|---|
| 52 | Podman | Alpine | aufs | v2 | crun | No |
| 53 | Podman | Alpine | btrfs | v2 | crun | No |
| 54 | Podman | Alpine | overlay | v2 | crun | No |
| 55 | Podman | Alpine | vfs | v2 | crun | No |
| 56 | Podman | Alpine | zfs | v2 | crun | No |
| 57 | Podman | Alpine | btrfs | v2 | crun | Yes |
| 58 | Podman | Alpine | overlay | v2 | crun | Yes |
| 59 | Podman | Alpine | vfs | v2 | crun | Yes |
| 60 | Podman | Centos | aufs | v1 | runc | No |
| 61 | Podman | Centos | btrfs | v1 | runc | No |
| 62 | Podman | Centos | overlay | v1 | runc | No |
| 63 | Podman | Centos | vfs | v1 | runc | No |
| 64 | Podman | Centos | zfs | v1 | runc | No |
| 65 | Podman | Centos | aufs | v2 | crun | No |
| 66 | Podman | Centos | btrfs | v2 | crun | No |
| 67 | Podman | Centos | overlay | v2 | crun | No |
| 68 | Podman | Centos | vfs | v2 | crun | No |
| 69 | Podman | Centos | zfs | v2 | crun | No |
| 70 | Podman | Centos | btrfs | v2 | crun | Yes |
| 71 | Podman | Centos | overlay | v2 | crun | Yes |
| 72 | Podman | Centos | vfs | v2 | crun | Yes |

### 4.2.2 Experiments

For all the experiments presented here, time measurements are taken at four important steps of the execution of the container:

$t_0$ Before its creation, this is time zero.

$t_c$ After its creation. Container's state is *created*.

$t_s$ After its startup. Container's state is *running*.

$t_e$ After the end of the execution of the command. Container's state is *running*.

This gives us three interesting measurements:

1. Creation time: $t_{create} = t_c - t_0$
2. Starting time: $t_{start} = t_s - t_c$
3. Execution time: $t_{exec} = t_e - t_s$

The command available to interact with each step of the lifecycle of the containers for each solution can be found in Appendix A, along with some more explanation about what is done in each step.

To do this, each container will be running as first process a shell[5], which will never receive any input, but is here to ensure that the container will stay up as long as we need it. When we do not need the container anymore, we simply kill all the processes in it.

Each experiment will be executed 20 times for each configuration. The final results presented in the next chapter present either the mean of the 15 best executions for each configuration of the boxplot for all the executions.

### Booting time

This test aims at determining the candidate solution that can provide a ready to run container the fastest. To measure this, a simple container composed of only the base image is created, in which the simple command `/bin/echo Hello World` is executed. The best candidate would be the one that has the shortest creation and starting time ($t_{create} + t_{start}$). The execution time ($t_{exec}$) is also a good illustration of the overhead caused by all the steps taken before actually starting the process in the container.

### Basic I/O performances

Those tests aim at determining mostly the influence of the usage of each storage driver on the performances of an application that relies highly on it. Four different tests are done here: big file read and write, and a great amount of file read and write. The best candidate solution would be the one with the shortest creation, starting and execution time ($t_{create} + t_{start} + t_{exec}$).

For the first one, five SQLite databases have been created, of five different size (`151.6 kB`, `536.6 kB`, `2.6 MB`, `11.9 MB` and `111.6 MB`, about one order of magnitude apart from each other). We use an SQLite database as everything will be stored in one file, and no daemon needs to be running, which simplifies the setup of the tests. Plus this is a more realistic workload than just reading/writing one gigantic file. The reading test will do a select operation on each table of the database without any filter, displaying all of the table content. The writing test will duplicate each table in the database. The database chosen as basis to generate all the presented one is `tpcc`[6], which is a benchmark database, perfect for our case. The biggest database used is an export from the tpcc database, the other ones are created by removing the content of some tables from this same database.

[5]except for LXD, which natively has an init process running in each container

[6]ttps://relational.fit.cvut.cz/dataset/TPCC

For the second one, five loads of small file have been created (10, 100, 1000, 10 000 and 100 000 files). The content of the file is alphanumeric and completely random (no file should be a duplication of another one), each file is 4096 character (and bytes) long. This represents the best the usual kind of file a user could access, to read or write content from/to it. The reading test will load the content of each file in memory, then discard it and move to the next one. The writing test will extract an uncompressed archive (`tar`) containing all those files.

### Network setup time

This test aims also at determining the candidate solution that can set up a ready to run container the fastest, but with the increased complexity of adding a network interface, and mapped ports, so that the host machine can send requests to an HTTP server running inside of the container. I have chosen a `lighttpd` server, for its lightness, and its wide use in container application. It takes one more time measurement here, the time at which we received the first response at the HTTP requests we send to our server ($t_r$). The best candidate would be the one that has the shortest delay before this response ($t_{create} + t_{start} + t_{exec} + t_{response}$, with $t_{response} = t_r - t_e$). The execution time corresponds to the execution of the command that launches the server as a detached process.

### Ping response time

This really simple test aims at determining if some solution provides a much weaker network solution to the container than the others. We simply perform a ping request to `1.1.1.1`, and compare the response time we get for each solution. The best solution would be the one with the shortest ping response time.

# Chapter 5

# Results interpretation

Based on the results of the different experiments I made, I will try to answer in this chapter to three main questions:

1. Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it? Do some solutions involve trade-offs in terms of maturity, support or maintainability?

2. Could there be a solution tailor-made for the specific case of INGInious? What would it be? What would it cost to use it?

3. What would be the cost of providing stronger/safer isolation to the containers used by INGInious? Which opportunities could it bring?

## 5.1 Current INGInious situation

We will here consider the first question:

"*Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it?*"

The current configuration of INGInious is the following:

| | |
|---|---|
| **Container manager** | Docker |
| **Base image** | Centos |
| **Storage driver** | overlay2 |
| **Container runtime** | runc |
| **Control group version** | v1 |
| **Rootless containers** | no |

This configuration is quite decent and gives good performances, there is mainly one change that can improve those. But let's analyze this step by step.

### Storage driver

Overlay2 (referenced for the rest of this discussion as simply overlay) is the storage driver that Docker recommends to use by default (when OverlayFS is supported on the host). Its layer mechanism allows you to limit the redundancy of information when you use the same base image to create different images. And its file-based copy-on-write strategy gives overall good performances.

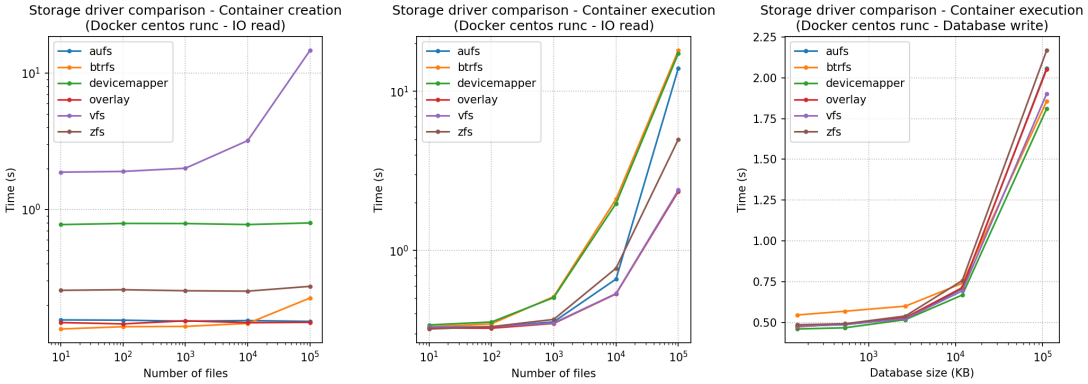


Figure 5.1: Storage driver performance comparison for Centos containers, launched with Docker and runc

In Figure 5.1, we can see how much it cost to have a full-copy mechanism like vfs for each container creation. We can also see that overlay, aufs and btrfs offer similar performances for the creation of containers. The main differences between those three are shown by their performance when soliciting I/O.

- Aufs, even though performing similarly to overlay in all the other cases (not shown here) and relying on the same union file system mechanism as overlay, gets far behind the latest when a lot of reading operations are done. This most likely comes from one big difference in the implementation of those two union file system: when a file is opened with OverlayFS, all the operations on it are directly managed by the underlying file systems. This simplifies the implementation and improves the performances quite a lot as we can see in the second graph of the figure. (For this test, for each `openat` system call, four `read` system calls are done)
- On the last graph of the figure (on the right), we can see the advantage of using block-based copy-on-write, when modifying big files. Indeed, when

doing so, where overlay and aufs will need to copy the whole file before modifying it, the other solutions will only need to copy the file system blocks needing to be modified (or not even copy the file in the case of vfs). We can see that devicemapper handles this a little bit better than btrfs, but given the much higher container creation cost of the first one, it will most likely be more interesting to use btrfs in most cases.

- One more thing to pay attention to is that storage drivers with block-based copy-on-write mechanisms seem to face some difficulties when creating containers with a large number of files in it. This might be harder to see here for devicemapper and zfs, but it is the case. Zfs seems to also suffer from big files in the container filesystem when creating containers.

One more reason to justify the use of overlay is that in most container use cases, read operations are the most important ones. Normally, no large amount of data should be written in the container file systems, volumes (external storage from the host, mounted into the container file system) should specifically be used for that purpose as it provides nearly native writing performances and the data persists, even after the end of the container life cycle.

In terms of maintainability, one thing to pay attention to is that Docker sometimes stops the support for some storage drivers. It already occurred for devicemapper since version 19.03, and might be the case soon for aufs too.

### Base image

Alpine is quite popular in container file systems. Thanks to its minimalist default configuration, its image is quite light compared to the ones of more complete file systems. The Alpine base image provided by Docker is only 5.61`MB`, while Centos's one is 203`MB`. But the size taken by images on disk is not something we worry about in our case as it will not have any impact on the user experience.

From the graphs of Figure 5.2, we can notice several things:

- We can see on the first graph (on the left) that, once more, btrfs offers better creation performances when the number of files is smaller. Therefore, Centos being a more complete Linux distribution than Alpine, with more functionalities, and so, more files, the creation performance with this storage driver will be poorer than when Alpine.

- On the second graph, it also appears that simple read operations are a little bit faster when using Alpine as a base image. This trend has also been observed with big file reading, with the *Database read* test.

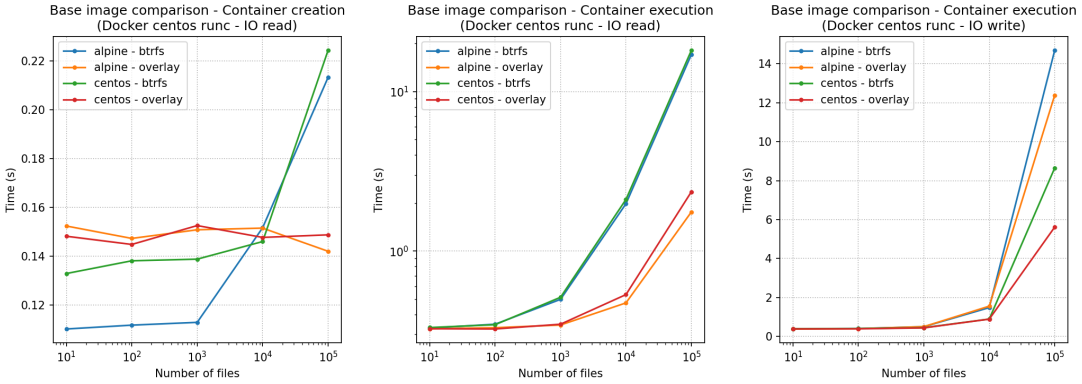


Figure 5.2: Base image performance comparison for containers launched with Docker and runc

- One thing really interesting that we can see on the last graph, however, is that the trend seems to be inverted for write operations. This difference is likely to be caused by the difference in the implementation of `tar` (used for this test) in each distribution's package repository. Indeed, the implementation coming from Alpine makes a lot more system calls (about three times more) than Centos's one.

The choice of Centos as a base image can then be justified by the maturity of such distribution. It had been around for a while, it is widely used outside of container applications, and is more likely to count optimization in the different tools at disposal. When those tools are not required though, the minimalist design and lightweightness of Alpine make it a better choice.

### Container runtime

On Figure 5.3 is shown the influence of different container runtime solutions on the different execution steps of the container.

By first comparing crun to runc, we see how much of a difference it makes to use an efficient C implementation, over a less efficient Go one. The difference is much more obvious for the `start` and `exec` phases as those are the steps where real low-level operations are made by the container runtime (entering namespaces, setting cgroups, forking processes).

Then we have the Kata Containers solutions, based on virtualization. It obviously will lead to a greater overhead for creating and starting containers, as a whole kernel has to be loaded. However, the `exec` phase seems to have much lower overhead. We can not miss how much worse is `kata-fc` (using Firecracker

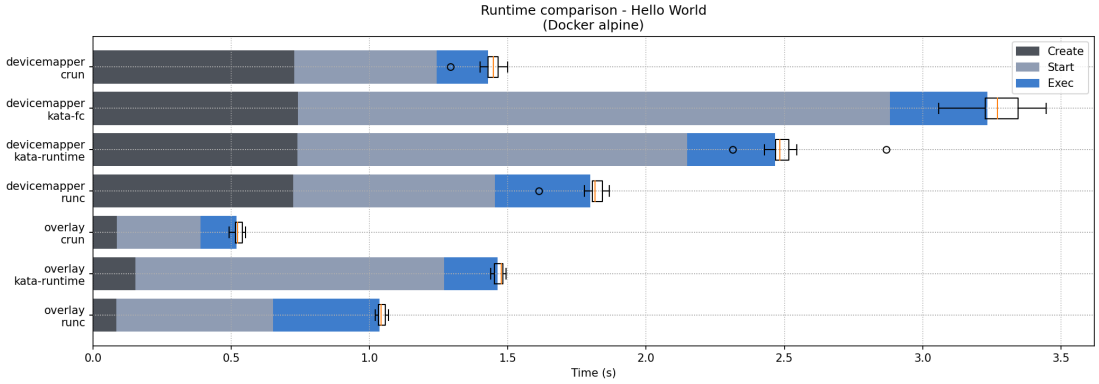


Figure 5.3: Runtime performance comparison for Alpine containers, launched with Docker

hypervisor) compared to `kata-runtime`, and it might be disappointing given the recent popularity that has embraced Firecracker. In response to that here are some important elements:

1. Firecracker has not been conceived to run under Kata Containers' hood. It might perform better when running standalone.

2. Some advantages that Firecracker has and which are not obvious in our study case are, firstly, the memory footprint, which is way smaller than with Qemu, and secondly, the reduced code base, which ensures a much lower attack surface. Those are also reasons why Firecracker is so popular.

3. After some discussion with Kata Containers community[1], it turned out that Kata Containers make use of one feature of Qemu that Firecracker does not have, that would justify this difference in performance: vNVDIMM. Thanks to vNVDIMM, the root file system of the guest VM is fully loaded in memory and directly accessed in it, which is faster than reading it from disk as Firecracker does.

All those runtimes are completely mature alternatives, with great support, a community behind it and a large user base.

### Container manager

On Figure 5.4 are shown the different container manager considered in the experiments, in the configuration that my tests revealed as most performant. Docker

[1]The full discussion can be found at this link: https://github.com/kata-containers/runtime/issues/2642

and Podman using crun as runtime, and overlay or btrfs as storage driver. LXD uses LXC as runtime and btrfs as a storage driver.

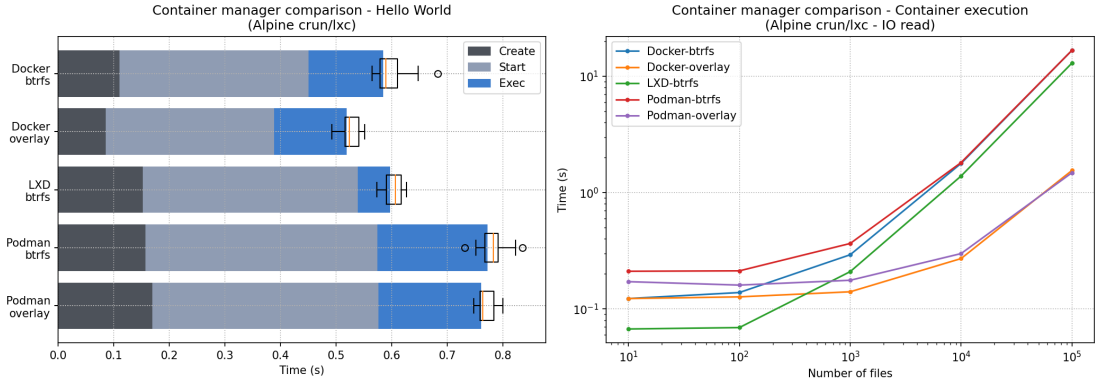


Figure 5.4: Container manager performance comparison for Alpine containers

We can see that Docker is still the best solution for us, even though none of the other solutions presents really bad performances. Given the relatively young age of Podman, its focus might not be yet fully on performances, but rather on functionalities, catching up all the things that the two other alternatives already have. We can then hope for improvements on this as time goes. It would be worth check on them later this year, or the next one.

All those three solutions are reliable, have great support, and a community to back them up. LXD is maintained by canonical, which makes it a great fit for Ubuntu environments. Docker is maintained by Docker Inc. and also seems to bring greater support for Ubuntu. Podman is supported by the containers[2] organization and is served as default Docker alternative on Fedora. Using Podman on Ubuntu has been a bit more complicated than the two others, especially for the rootless solution, which requires cgroupv2. Thankfully the community offered great support for all of my problems, and in most cases, the issues appeared to be with systemd or Ubuntu, rather than with Podman[3].

One last positive point to give to Docker is also its documentation. Even though there is also documentation for the other tools, it is much more complete on Docker's side. And Docker being also the most popular solution, a lot of complementary information, blog posts, stack overflow threads, are available online too.

[2]https://github.com/containers

[3]The full discussion for the issue I had can be found here: https://github.com/containers/libpod/issues/6368

### Final configuration

Based on the previous observations, the ideal configuration would then be:

| | |
|---|---|
| **Container manager** | Docker |
| **Base image** | Alpine/Centos |
| **Storage driver** | overlay2 |
| **Container runtime** | crun |
| **Control group version** | v1 |
| **Rootless containers** | no |

As we can see, the final configuration is not that much different from the original one.

The quick answer to the original questions "*Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform?*" would be: The current configuration is good. It could be improved but it is not the worst one. The choice of going with Docker was the obvious choice when INGInious was created, and it is still the case now. The current storage driver offers great performances for almost every case, only some specific case, soliciting a lot of IO operations on big files, can give an advantage to another storage driver, btrfs. The choice of a Centos base image is not bad either, but the size of the image being greater than Alpine's one makes it less ideal for containerization applications. Except for the situations where Centos offered better write performances, using Alpine seems to be better. The move here might then be to go for a hybrid solution, using Centos only in situations where its small writing performance advantage becomes meaningful.

*How much better could it be?* As we have seen, changing the current container runtime makes a significant difference. The c implementation of `crun` is claimed to be twice as fast as the go implementation of `runc` by `crun`'s contributors. And we can see the difference. The change of base Image though does not show an obvious improvement.

*How easy would it be to improve it?* This is the real good news, it truly is super easy to apply the most significant change. You only need to install `crun`, reconfigure Docker to use it by default, and we are good to go, no change as to be done to INGInious! For the choice of the base image though, the story is different, as it would require to change all the containers images, which is a lot of work.

*Do some solutions involve trade-offs in terms of maturity, support or maintainability?* All of the solutions I explored here are reasonable to use, none of them have weak support or lack of maintainability or stability. But as discussed, some storage driver might lose their support on Docker's side and Docker is better documented than the other container manager.

**Note about Http Serveur and Ping test**: The results of the different configuration to those tests were not presented here and will not be shown later. This is because no additional information could be extracted from those. The performances to Http Server test are in pairs for each configuration with its result to the Hello World test. Regarding the Ping test, the variations were too small to assume a difference in performance rather than the varying network load. If this test has to be reproduced, it might be interesting to ping a target in a private network that we have under control, rather than the internet. More graphs can be found on the repository of this project at this address: https://github.com/edvgui/LEPL2990-Benchmark-tool/tree/master/plots.

## 5.2 INGInious ideal solution

We will here consider the second question:

"*Could there be a solution tailor-made for the specific case of INGInious? What would it be? What would it cost to use it?*"

The different container managers presented here are very versatile solutions, they all have a great package of functionalities that allows them to deal with a wide range of different applications. This is good for them, but bad for us, as we do not need most of those functionalities, but still pay the cost through a heavier container manager.

The workload that INGInious handles with its container manager is not that extensive, it is even quite redundant, and predictable, as each student's code is expected to have a specific behavior. And that expected behavior can define, in advance, which files in the container filesystem are going to be modified, and which are not. We can play with that knowledge.

I have then created a concept alternative solution, Contingious, that would just fit the needs of INGInious, and provide better performances. It relies on two things:

1. We do not need a fully functional and grown-up container manager. Our needs are simply containers, with resource management. Those are provided by tools like crun and runc and cgroup v1 and v2. We could then gain performance by only using those.

2. We do not need a complete writable filesystem for each container, we can restrict write access to certain parts of it. This means that we could copy the file that will be modified when creating the container (avoiding the need of using any copy-on-write mechanism), and bind-mount the rest of it, with read-only permissions.

The full current implementation and some more implementation details can be found in the repository of this project at this address: https://github.com/edvgui/LEPL2990-Benchmark-tool.


Figure 5.5: Ideal INGInious container manager solution compared to the current configuration of the platform and to the recommended new solution, Hello World test

In Figure 5.5 we can see that the gain in performance is real, and we did not give up any of the key requirements of INGInious. We have the same level of isolation that Docker provides as we rely on crun, we even made it one step further as this solution is completely rootless.

In Figure 5.6 we can see that the only battle we lose is with the current INGInious solutions for the writing test, as its `tar` implementation is more efficient. Otherwise, reading file is way faster when you do not need to manage different layers of your file system as with overlay, and writing is faster too as, again, we do not need to care about different layers.

In Figure 5.7 we see that this solution is not magical though, copying files before editing them is still costly. But doing this when creating the container still provides overall better performances than copying the file to a higher layer, as overlay would, just before writing it. The only alternatives that could beat us for more extreme situations are block-based copy-on-write solutions, as they will not copy the whole file. That being said, we could probably improve this solution by

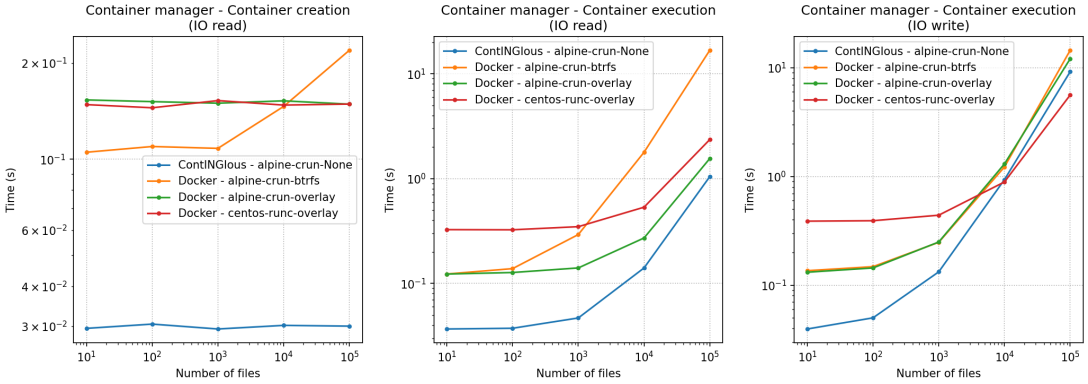


Figure 5.6: Ideal INGInious container manager solution compared to the current configuration of the platform and to the recommended new solution, IO tests

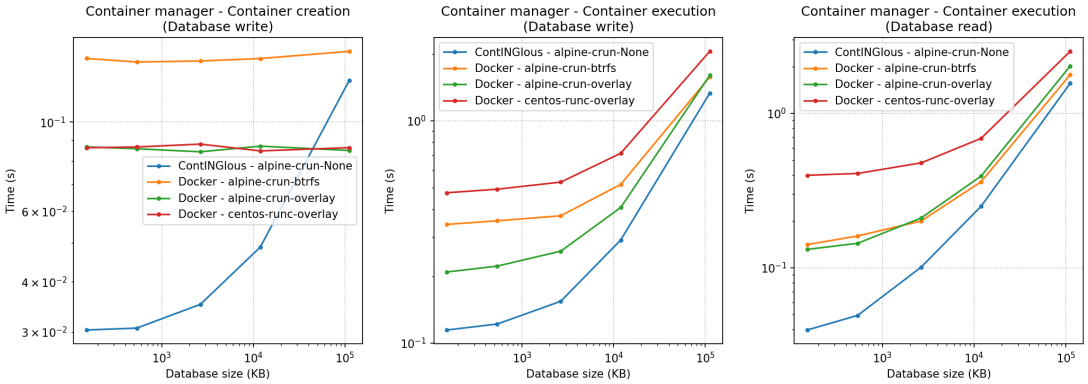


Figure 5.7: Ideal INGInious container manager solution compared to the current configuration of the platform and to the recommended new solution, Database tests

making use of that feature too, relying on an underlying filesystem that has such capabilities.

This prototype shows the improvements expected, but there is still work to do before integrating it into INGInious.

- Determine for each container which should be the writable part of the file system. And eventually, optimize those images by centralizing all the writable content.
- Create a new image management system. We do not need to go from scratch

though, Buildah[4] could be used, and would even allow us to continue using Docker Hub to store images.

- Integrate all the management of the steps of creation and execution of containers in one reliable tool. For that we do not need to start from scratch, it might be worth taking a look at conmon[5].

Whether the performance improvement of this solution is worth the effort it would require to put it in place is not my call to make. I like that idea, and I think it would make a great project, but it would also cost some time to set up and maintain.

## 5.3 INGInious opportunities

We will here consider the last question:

"*What would be the cost of providing stronger/safer isolation to the containers used by INGInious? Which opportunities could it bring?*"

By saying "*providing a stronger-safer isolation*" what I mean is more mitigating the risk we take when we let a student execute code in a container, on the host where the platform is running. The most unsafe solution (still making use of containers) is to run the container as root, with the root user of the container in the hand of the student and mapped to the root of the host. In theory, this does not do any harm but given past vulnerabilities that have been found in runc, it can happen. To mitigate this, INGInious currently creates a new user in the container, which has no root privileged (neither inside the container nor outside of it), for the student code to execute. This limits the range of possibilities of tasks to evaluate with INGInious. We cannot for example simulate a cluster of machines where the student has root access to deal with any network assignment.

### Rootless containers

The first possibility to improve this situation is to go with rootless containers. We have two alternatives for this: Podman's rootless containers and LXD unprivileged containers. Docker is also working on a solution but has only limited support for now. With rootless containers, not only the root inside the container is not root on the host, but also the container runtime is not running as root, which limits

[4]Buildah is a tool that allows us to easily build and manage OCI images. https://github.com/containers/buildah

[5]Conmon is the container runtime monitor on which Podman relies. https://github.com/containers/conmon

a lot the damage that can be done to the infrastructure if a vulnerability in the container runtime is found and exploited. It also mitigates potential attacks the other way around, for a user on the host that would try to gain root access through a vulnerability of the container manager. This is only valid for Podman though, as LXD's daemon is running as root, even for unprivileged containers. Podman could then be installed on the student's machine in computer rooms without too much concern.

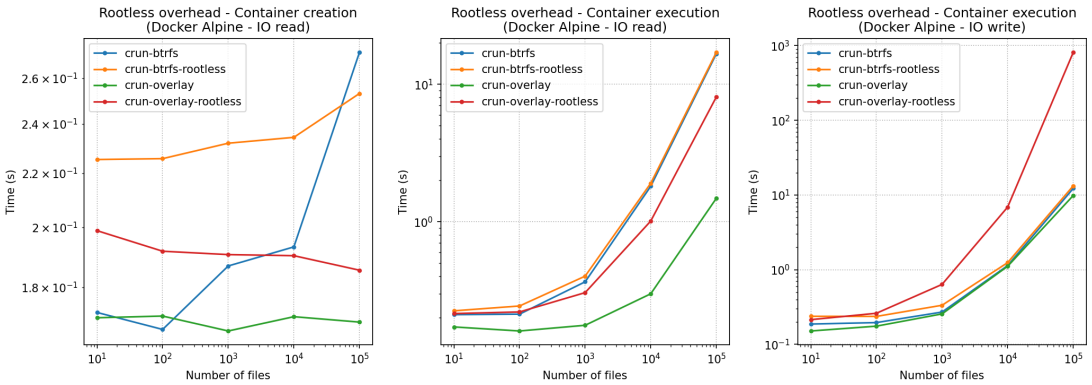


Figure 5.8: Overhead of rootless containers compared to rootfull ones, IO tests

Unfortunately, having rootless containers also comes with a cost as we can see in Figure 5.8.

- There is a cost to create, start and execute command into rootless containers with Podman compared to going rootfull with the same container manager. This is very likely caused by the extra steps required to launch a rootless container with all the functionalities of a rootfull one. Podman needs first to unshare user and mount namespaces, to be able to make bind mounts on the container filesystem. It also requires adding a cgroup scope to all the commands that set up the containers, to be sure that the container runtime will later be able to move process created to the cgroup attached to the container.

- Because of the lack of *real* root permissions on the machine, rootless containers can not use OverlayFS, instead they use a FUSE[6] implementation of the latest, which, as we can see on the right-most graph, induces great performances loss. It gets even worse when doing write operations.

[6]FUSE, or filesystem in user space, allows for a non-privileged user to create a filesystem in user space instead of kernel space.

- One limitation of a rootless container is also the options available for managing the storage of containers (storage drivers), for example, devicemapper is not available.

### Virtualization

Another solution is to simply strengthen the isolation that protects the host from the student. And the best solution to do this is to make use of virtualization. Kata Containers is a great deal for us, it allows us to keep the infrastructure of the platform, only changing the runtime used by Docker, and everything can still be used as before. Once again, this stronger isolation level could open the gate to new assignments that would require the student to have root access in the container.

However, as we already saw earlier when comparing runtimes, virtualization adds a cost, mainly to container start-up, but also on creation and execution.

In Figure 5.9, we can notice that unlike for crun, the overlay storage driver behaves way worse than devicemapper. This is also the case for all other storage drivers. It seems that runtime solutions with virtualization take much more advantage of devicemapper than other drivers. And passed a certain point, Kata Containers using Qemu even perform better than crun, both in read (compared to crun with devicemapper) and write (compared to crun with overlay or devicemapper).

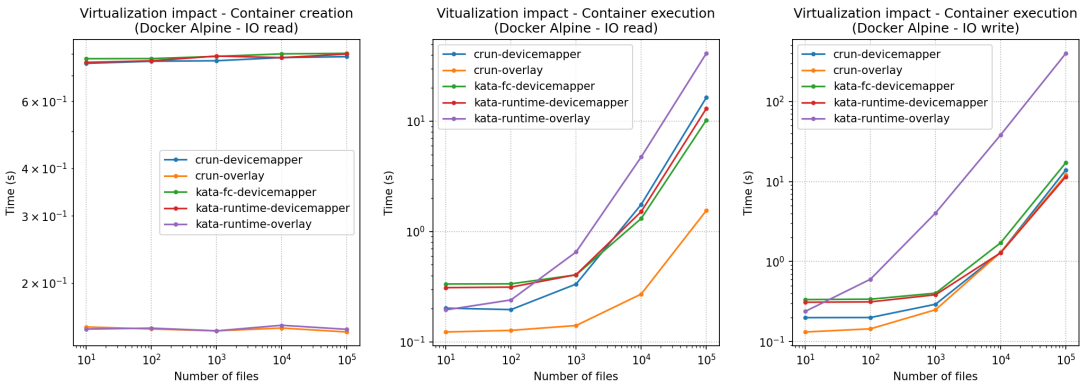


Figure 5.9: Cost of virtualization on container creation and execution, IO tests

We can make the same observation in Figure 5.10 with the exception than, for some reason, Kata Containers using Firecracker seems to handle read operations on big files much better than when using Qemu.

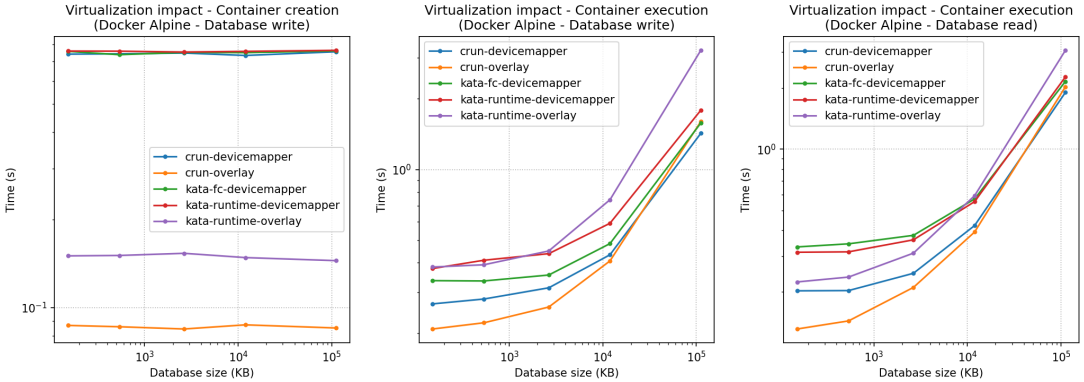


Figure 5.10: Cost of virtualization on container creation and execution, IO tests

One more great thing about Kata Containers is that every container in the same pod (a group of containers, linked together, that serves a common global purpose) will be grouped inside the same VM (still in separate containers), this will reduce the overhead by container caused by virtualization as the size of the pod grows. This could be convenient for assignments where multiple containers need to be managed by a single student. Unfortunately, to make use of this feature, the container manager needs to have some kind of pod mechanism, which is not the case of Docker. Podman, Kubernetes or others can do that.

# Chapter 6

# Conclusion

In this manuscript, I have addressed the following questions and answered them based on results to a suite of micro-benchmarks I created. Those micro-benchmarks aimed at being representative of the current load faced by INGInious when dealing with student's submissions and bring to light how different configurations could affect the responsiveness of the platform.

1. Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it? Do some solutions involve tradeoffs in terms of maturity, support or maintainability?

2. Could there be a solution tailor-made for the specific case of INGInious? What would it be? What would it cost to use it?

3. What would be the cost of providing stronger/safer isolation to the containers used by INGInious? Which opportunities could it bring?

To respond to the first question, I have questioned the current configuration that INGInious has chosen to face the responsiveness challenge of the platform. I have identified different changes that could be made to it and shown whether or not they would be worth it. The most interesting change would be to change the current container runtime in use, `runc`, to `crun`, which performs as much as twice as fast in some tests.

For the second question, I have presented what would be the best solution for INGInious's case and measured its performance gain, in my micro-benchmarks, over the best existing alternative. Showing that the difference we can still gain is significant, performing up to three times as fast as INGInious's current configuration.

And finally, for the last question, I have discussed some of the safer isolation solutions that are existing, virtualization and rootless containerization. I presented

new opportunities it creates for INGInious, like creating tasks where the student would have full root permissions inside of a container. Here is a summary of part of the configuration I tried, and what they are worth in terms of performance, isolation, ease of use and support (through documentation, community, forums).

| **Manager** | **Runtime** | **Rootless** | **Isol.**[1] | **Resp.**[2] | **Usab.**[3] | **Support** |
|---|---|---|---|---|---|---|
| Docker | runc | No | + | +++ | +++ | +++ |
| Docker | crun | No | + | ++++ | +++ | +++ |
| Docker | kata-runtime[4] | No | +++ | ++ | ++ | +++ |
| Docker | kata-fc[5] | No | +++ | + | ++ | +++ |
| Podman | runc | No | + | ++ | +++ | +++ |
| Podman | crun | No | + | +++ | +++ | +++ |
| Podman | crun | Yes | ++ | ++ | +++ | +++ |
| LXD | LXC | No | + | +++ | +++ | ++ |

As there is no bad solution, there is no bad score either, but the goal is to have as many "+" as possible.

### Personal enrichment

A master thesis is much more than reading a bunch of papers, writing nice code and plotting some graphs. It has been a real challenge to discover this whole new world that is containerization and I have learned many more things than what my results show. It was my first time dealing with such a big project, and I was the only one directly contributing to it. It required more planning than what I usually need. It was also the first time that I was not relying on a course's support to learn new things. I had to find other trustful content and build my knowledge of the situation myself. This project has also been a great opportunity for me to enjoy the greatness of open-source. All the tools I have handled are open-source, and I could get support from the community when I required it, which was a big help and made me feel less alone in this.

### Follow up questions

Even a year of work is not enough for me to cover everything on the subject I presented here. Partly because the containerization world is continuously growing and new things to consider appear regularly. And partly because I started this

[1] Isolation
[2] Responsiveness
[3] Usability, ease of use
[4] Kata Containers with Qemu hypervisor
[5] Kata Containers with Firecracker hypervisor

year with no experience or even basic knowledge on the subject, I did not know from the start what I was going to do and going further in my work has not ceased to open me new doors to look behind. Here is then a small enumeration of thing that, if I had one more year, I would consider going into:

1. As the different tools I compared are continuously evolving, and new opportunities appear, the amount of configuration to consider grows really big. It might be interesting to create a tool that could easily integrate into its benchmark new upcoming solutions. It would allow us to get periodically updated results as some tools grow.

2. LXD can also be used to manage virtual machines, I do not know how it works, what it relies on and if the performances are anywhere near ones of containers, but it could be worth taking a look at.

3. LXD has also its own "rootless" containers feature, with unprivileged containers, I did not have the chance to take a look at this but it could be interesting as well.

4. My knowledge of virtualization technologies is weaker than containerization one, and there is one result that I can not explain, which is the really bad behavior of storage drivers others than devicemapper with Kata Containers.

# Appendix A

# Container life cycle

During its life cycle, a container will go through five stages: creation, launch, execution, stopping and cleaning. Those five steps are explained below and presented in figure A.1, along with the corresponding command for each presented tool. Other optional stages (like pause/unpause) are not presented here.

1. **Create** This is the set-up phase, depending on the tool, the file system of the new container might be copied, or any other things that need to be done before launching the container. This phase can only be done one time for each container.

2. **Launch** This is the proper instantiation of the container, the namespaces are created, the new file system is adopted. Depending on the container, a complete init process might be executed. This state can be repeated as many times as we want for a container, as long as it is in a stopped state.

3. **Execute** This corresponds to the execution of a process in the container. This can be done as many times as we want for a container, as long as the container is running.

4. **Stop** This is when the container needs to be stopped, all running processes are killed, and the namespaces are exited. No modification should be done on the filesystem, the storage of the container remains.

5. **Clean** This is the un-create phase, where we delete everything that was done during the creation phase. The storage will be lost after this phase.

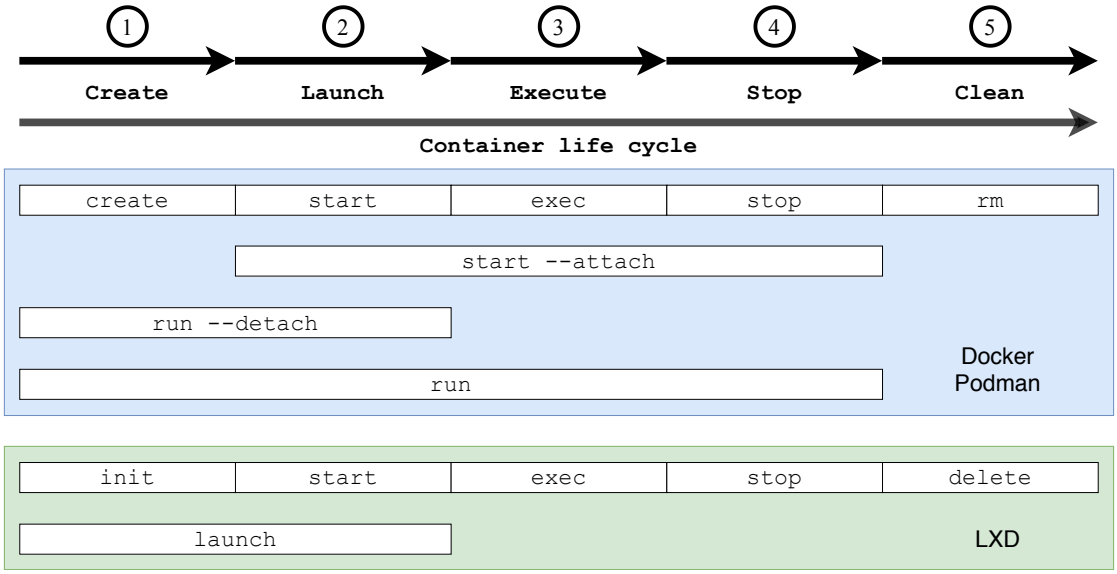


Figure A.1: Container life cycle and how to interact with it.

# Bibliography


[1] An industry-standard container runtime with an emphasis on simplicity, robustness and portability. `https://containerd.io/`. Accessed: 2020-02-26.

[2] Linux vserver. `http://linux-vserver.org/Paper`. Accessed: 2020-04-25.

[3] Open container initiative. `https://www.opencontainers.org/`. Accessed: 2020-03-04.

[4] Openvz. `https://openvz.org`. Accessed: 2020-04-25.

[5] Production-grade container orchestration. `https://kubernetes.io/`. Accessed: 2020-02-26.

[6] What is podman? `https://podman.io/whatis.html`. Accessed: 2020-02-28.

[7] What's lxc? `https://linuxcontainers.org/fr/lxc/introduction/#whats-lxc`. Accessed: 2020-03-01.

[8] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications.

[9] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[10] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[11] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[13] Zhanibek Kozhirbayev and Richard O Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182, 2017.

[14] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962. IEEE, 2017.

[15] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[16] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[17] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[18] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[20] Randolph Y Wang and Thomas E Anderson. xfs: A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 71–78. IEEE, 1993.

[21] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**