GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Comparison of various runtimes in Kubernetes

Jule Anger

MatrNr: 21968167

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2024

Abstract

Containers and Kubernetes are being used more and more frequently by more and more companies. The classic containers and container runtimes such as containerd are very performant, but less secure than virtual machines. New approaches have therefore been developed in recent years to combine the performance of containers and the security of virtual machines. Three important innovative runtimes are Kata Containers, Firecracker and gVisor. These three runtimes and containerd were installed in a test setup and the startup time of containers was measured and compared with experiments from other works. This showed that gVisor has the fastest startup time and the highest security, while Firecracker in particular has a higher startup time.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☑ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☐ To create individual passages, altogether to the extent of 0% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

OCI Open Container Initiative

VM Virtual Machine

CVE Common Vulnerabilities and Exposures

GPU graphics processing unit

VMM Virtual Machine Monitor

CRI Container Runtime Interface

GVM Go version manager

OS Operation system

API application programming interface

CNI Container Network Interface

1 Introduction

1.1 Motivation

Serverless applications are becoming an increasingly important part of the modern working world. Containers are resource-efficient and very fast, which is why they are often used. An important system for serverless applications is Kubernetes, which can be used to run containerized applications. The containerized applications are executed in a container runtime. However, as containers share resources with the hosts, they are more susceptible to vulnerabilities. In 2017, for example, over 450 security vulnerabilities were found in the Linux kernel that could be exploited by applications running in containers to access host data. Only recently, several Common Vulnerabilities and Exposuress (CVEs) were discovered that make containers vulnerable. The CVEs CVE-2024-21626, CVE-2024-23651, CVE-2024-23652 and CVE-2024-23653 are all classified as serious and can be used to access sensitive data such as customer information or to execute further attacks. Although the vulnerabilities were fixed immediately, the question arises as to whether traditional containers are secure enough for day-to-day work or whether new concepts are needed. [Stö24] [Guo+23]

Since Virtual Machines (VMs) are more secure by design but slower, various approaches are being developed to combine the performance of containers and the security of VMs. In the following paper, three innovative, more secure alternatives to traditional container runtimes, namely Kata Containers, Firecracker and gVisor, are presented and compared with the traditional container runtime containerd in terms of design and performance. This is done in the context of Kubernetes.

1.2 Outline

The following work starts with a theoretical look at what precisely Kubernetes and container runtimes are. The four runtimes mentioned will then be discussed in more detail: Their development, their concepts and their implementations. Next, the practical setup is explained. A Kubernetes cluster will be set up, with each of the four worker nodes having a different runtime. The installation of the four runtimes is then described. The performance measurement tool used is also presented here. The results of the practical experiments and the theoretical concepts of the different runtimes are then compared and analyzed with the results of other studies.

Containers are normally less secure than VMs. To avoid this security risks, different organizations implement lightweight hypervisors to use the well performance of containers in combination with an increased security [Guo+23]. Three of this implementations with different approaches will be presented next.

The following research questions are considered in particular:

R.1 How easily can the various runtimes be installed in a Kubernetes cluster?

R.2 How does the performance of the runtimes in a Kubernetes cluster differ?

R.3 How do the security mechanisms of the innovative runtimes differ?

2 Theoretical

2.1 Kubernetes

Kubernetes is a system for the automated execution and scaling of containerized applications. It does the orchestration of containers and storage. Kubernetes also has active monitoring, i.e. errors are detected and automatic attempts are made to rectify them, e.g. by restarting the crashed application or its container. Kubernetes also performs load balancing. This means that if a machine is very highly utilized, this load is automatically distributed to other machines. [Autg] [JJ19]

All objects in Kubernetes can be created and customized using `yaml` configuration files. The containers are organized in so-called pods. Within a pod, containers can communicate with each other more directly and share a namespace. If there are one or more volumes in the pod, the containers can access them. In many cases, a pod contains an administration container and a container with the desired application. Pods and other object types are managed by so-called deployments. In a deployment, it can be specified which pods are to be created with which containers and volumes, which properties, which quantity (this can be both a fixed number and rules for automatic scaling) and which other objects are to be created. [JJ19]

Figure 1 shows the basic structure of a Kubernetes cluster with the most important components. In a Kubernetes cluster, there are one or more machines called master nodes that manage the cluster and one or more machines called worker nodes on which the containerized applications run. In a minimal cluster, there is only one node that is both master and worker; in a productive cluster, there are several masters and a larger number of workers. The first important component on the master node is the application programming interface (API) server. This is the interface through which the developer or administrator configures the cluster and its objects, such as pods or deployments. Next, there is the Controller Manager and its controllers, which are responsible, among others, when a node goes down. The scheduler decides on which worker node a new pod should run, taking into account set rules and available resources. The etcd database is a consistent, highly available key value store that stores all cluster data. On each worker node, the kubelet agent receives instructions from the API server and ensures that these are implemented correctly, e.g. that the containers run in pods. The kube proxy is a network poxy that enables users to communicate with the containerized applications. A network plugin is responsible for assigning IP addresses to pods and thus enables communication between the pods within the cluster. The most important component for this work is the container runtime, which is described in more detail in the following section. [Aute]

2.2 Container Runtimes

Kubernetes administers the containers using a container runtime. Any software that implements the Open Container Initiative (OCI) standard, a widespread, open industry standard for container formats, can be used for this. [Aute] [Fou]

A container runtime manages the container life cycle, which includes creating, starting, stopping and deleting. These are some of the minimum operations that a runtime must provide to fulfill the OCI standard [ope16]. The runtimes that only provide the basic

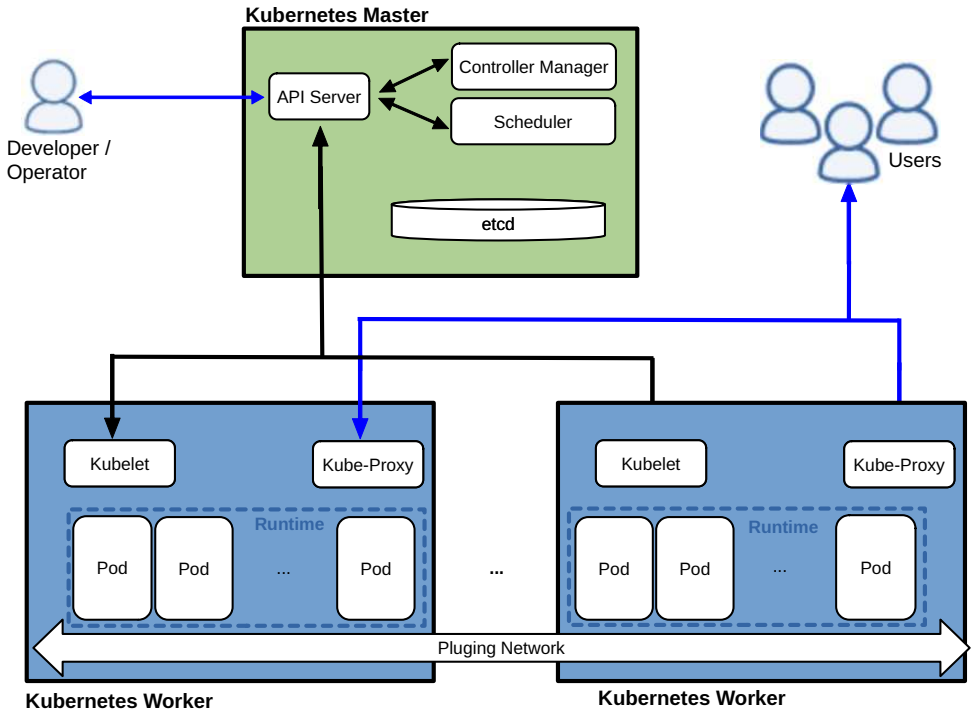

Figure 1: Structure of Kubernetes [Kht].

operations are called low-level runtimes. Common examples are runC, cRun and LXC [Ren] [Lewa].

There are also many other runtimes that provide many other functions in addition to the basic functions. They are called high-level runtimes. These functions include, for example, loading a container image from a remote repository, monitoring various local system resources, building, packaging and sharing container images [Lewa]. They run on top of a low-level or other high-level runtime and expand or improve the exisiting functionality. Common examples are Docker, containerd and CRI-O. Figure 2 shows that the high-level runtime docker-containerd uses the low-level runtime docker-runc [Lewb].

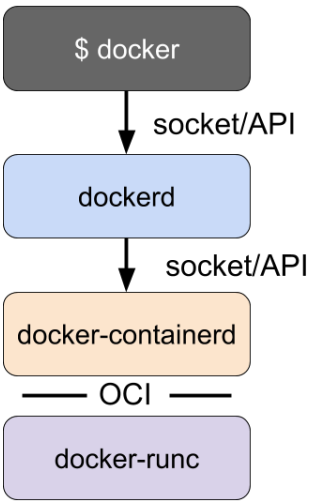

Figure 2: Stack of Docker runtime [Lewb].

2.2.1 containerd

containerd is a well-known, widely used high-level runtime. Along with CRI-O and Docker, it is one of the Kubernetes standard runtimes [Autf]. containerd's low-level runtime in RunC.

containerd uses two Linux features to isolate containers: cgroups and namespaces. With the help of cgroups, the use of resources such as CPU and memory can be controlled and monitored. Namespaces abstract a system resource so that a process running within this namespace appears to use this resource exclusively. containerd creates cgroups and namespaces for the containers and binds the container process to them. [Guo+23]

2.2.2 Kata Containers

Kata Containers is an innovative more secure container runtime using their own lightweight virtual machines called Kata VMs.

It was developed by a community since 2017 and published as open source with open design, development and community on GitHub[1] as a "open governance project under the Open Infrastructure umbrella" [Ada20] [kat24b]. The GitHub repository has around 4 000 pull requests. Contributors are of Intel, IBM, Google, Microsoft and many more [Ada20]. Kata Containers is a merger of Clear Containers, that was launched 2015 by Intel, and Hyper.SH runV. Both are published under Open Infrastructure Foundation [Ada20]. Intel was focused on performance (<1000ms boot time) and enhanced security, while Hyper was focused on compatibility and intended to be technology-agnostic by supporting many different CPU architectures and hypervisors [Fou17]. It is mostly written in Rust.

Kata Containers works "seamlessly with Kubernetes and Docker and is a drop in replacement for runc" [Ada20]. It supports different architectures like x86, ARM, IBM Power and IBM s/390x and different Hypervisors such as QEMU, Cloud Hypervisor, Firecracker [Ada20]. It can be used together with high-level runtimes like containerd [Comc]. Kata Containers supports graphics processing unit (GPU) since V1.3.0 which was released in September 2018 [Ada20]. It is highly salable. As an example, the Ant Group use Kata Containers "running on thousands of node and over 10K cores" [Ada20].

Kata Containers doesn't support live migration and other Linux distributions than Clear Linux [Guo+23].

While Traditional Containers have a kernel shared with the host, Kata Containers is more isolated in its own lightweight VM. Due to their light weight, the VMs have a similar performance to containers, but virtualize their own kernel with hardware virtualization and thus an additional layer on top of the traditional namespace-based container concept, which increases security. Within a Kubernetes cluster, a Kata VM is created for each pod in which the associated containers run. [Ada20]

2.2.3 Firecracker

Firecracker is an innovative more secure virtualization technology with own lightweight virtual machines called MicroVMs. Its most important component is a Virtual Machine Monitor (VMM), which completely replaces QEMU. To create and run the MicroVMs, the VMM uses the Linux Kernel's KVM virtualization infrastructure (see figure 4). It therefore uses the same basic idea for isolation as Kata Containers, but is implemented in a

[1] `https://github.com/kata-containers/kata-containers`

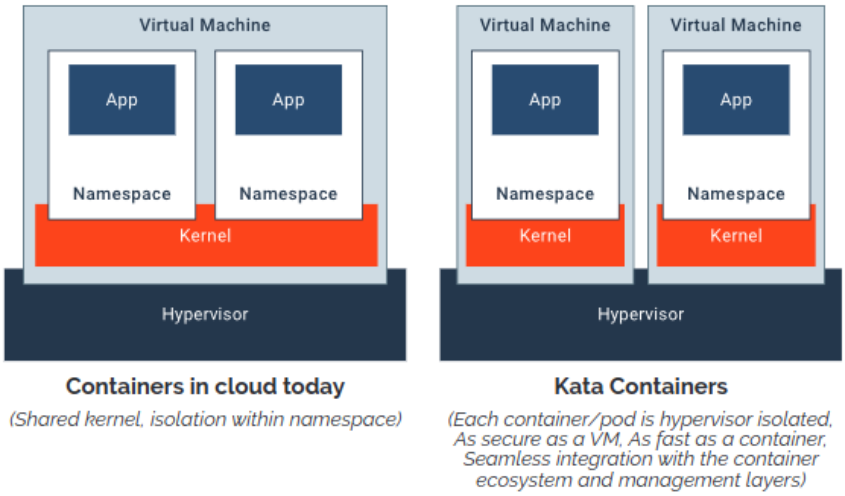

Figure 3: Security model in Kata Containers [Fou17].

significantly different way. While Kata Containers is a secure lightweight VM, Firecrackers VMM creates a secure environment for guest Operation system (OS). [Aga+20] [WDL22]

Unlike Kata Containers, Firecracker itself is not a runtime. However, the MicroVMs can be used with containerd or Kata Containers, for example. [Aga+20]

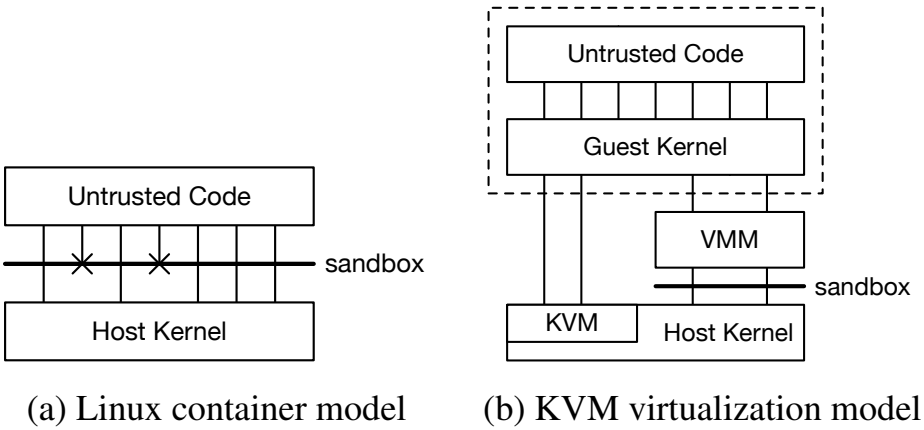

Figure 4: Security model of Linux containers (a) and KVM-style virtualization like Firecracker (b) [Aga+20].

Firecracker has been developed by Amazon and it's community since 2014. It was released as open source on GitHub[2] in Dec 2018 with the Apache 2 license and was used since then used in production in AWS Lambda. It was not developed explicitly for use in Kubernetes but for use in their own Cloud services AWS Lambda [Aga+20].

Firecracker does not support GPU usage. There is discussion within the community as to how it should be integrated. However, this will not be possible in the near future, as the concepts used by Firecracker contradict GPU support. [Coma]

[2] `https://github.com/firecracker-microvm/firecracker`

2.2.4 gVisor

Google's gVisor is an application kernel, that "provides an additional layer of isolation between running applications and the host operating system" [Autd]. Its approach to security is different from the lightweight VMs that Kata and Firecracker use. Instead, gVisor creates a virtualized environment and builds a sandbox around the container. Inside each sandbox is a kernel that the container interacts with instead of the host kernel, minimizing the risk of container escape exploits. This concept saves resources and effort for virtualization, but has reduced application compatibility and higher per-system call overhead. It is delivered together with a runtime called runsc, which implements OCI, see figure 9). [Autd] The runtime has GPU spport [Auta].

It was developed by Google and its community since 2019 and was published open source on GitHub[3]. The GitHub repository has around 8 000 pull requests.

Figure 5 shows the basic structure of gVisor. A sentry is present in every sandbox. The sentry is basically a kernel that implements all the necessary functions, such as syscalls, memory management, signal delivery, and more. However, no syscalls are forwarded to the host kernel, but are processed directly by the sentry, although the sentry itself makes syscalls to the host kernel. Since the sentry is started in a restricted seccomp container, it has no access to file system resources. The gofer, which runs in every container of a sandbox, provides access to file system resources. It communicates via the 9P protocol. seccomp is the secure computing mode in the Linux kernel to restrict the actions available within the container. [Autd]

gVisor improves the security of containers by protecting the system API in particular. The System API allows an application to interact with the system, e.g. via system calls. The gVisor runtime attempts to prevent the possibility of attacking a system via the system API by placing a sentry between the container with the potentially insecure code and the host kernel (see figure 5). This means that the application must communicate with the sentry's API instead of the host's API, thus preventing direct communication. This principle is also used for VMs. In contrast to a VM, the API of the sentry is based on the host; in a VM it is based on virtualized hardware and the guest operating system. In addition, the system API that the sentry can access is redirected to a secure set. This means that a user can never use the actual system resources, only the virtualized ones. The creation of new sockets or the opening of files, for example, are not included by default. [Autc]

gVisor doesn't support live migration [Guo+23].

3 Methods

3.1 Setup

A test setup was set up to test the innovative container runtimes. Five VMs were created on the GWDG's OpenStack[4] for this purpose. All VMs have 4GB RAM, 4 virtual CPUs and a 40GB disk. One VM is used as the Kubernetes master node. The other four are added to the Kubernetes cluster as worker nodes. A different runtime is used for each one. Preparations were made on each node according to [Mut] in order to run Kubernetes:

[3] `https://github.com/google/gvisor`
[4] `https://cloud.gwdg.de`

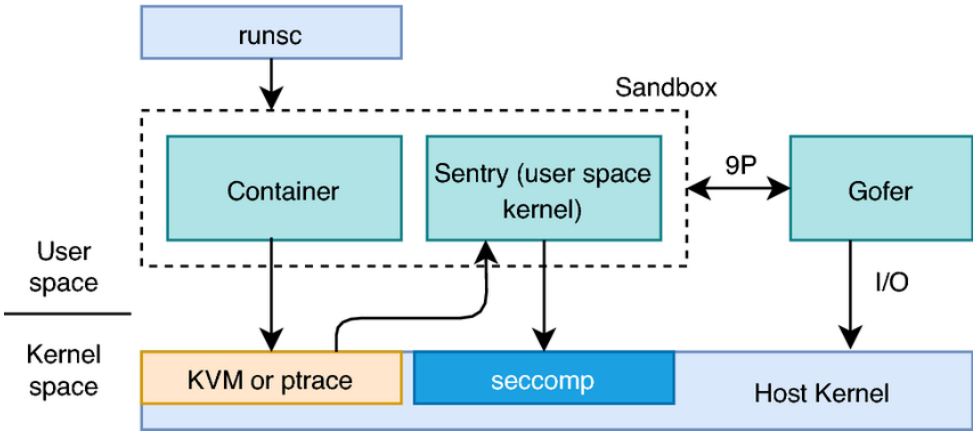

Figure 5: gVisor using an isolation layer between apllication and host kernel [Guo+23].

The packages kubeadm, kubectl and kubelet, all with version v1.28.2 provided via the repository `https://apt.kubernetes.io/` were installed using apt, swap disabled and the kernel modules `br_netfilter` and `overlay` enabled. Also `sysctl` was configured, so that Kubernetes can use the network.

```
cloud@master:~$ kubectl get nodes
NAME            STATUS   ROLES           AGE   VERSION
master          Ready    control-plane   31d   v1.28.2
worker-contd    Ready    <none>          16d   v1.28.2
worker-fire     Ready    <none>          15d   v1.28.2
worker-gvisor   Ready    <none>          10d   v1.28.2
worker-kata     Ready    <none>          25d   v1.28.2
```

Figure 6: An overview of the structure of the Kubernetes cluster with one master and four worker nodes.

Each worker node receives the label `type={contd|fire|kata|gvisor}` depending on the installed container runtime. This label can later be used to specify that a pod or deployment should be executed on a specific node. An example use of this can be found in Appendix B.1.

3.1.1 Installing the master node and a worker with containerd

The master node and the containerd worker node were configured according to the above guide. After the preparations, containerd version 1.6.26 was installed on the master node and the first worker node. To do this, the package was installed with apt, configured with the default configuration file and started.

On the master node, kubelet was activated and a cluster was initialized using the command `sudo kubeadm init` and the subsequent commands requested in the output. All settings were left at default. Calico was then installed as the network plugin.

The join command is output with the init command and can be displayed again if required when a new token is created using `sudo kubeadm token create -print-join-command`. This join command adds the worker node to the cluster that has just been created.

3.1.2 Installing a worker node with Kata Containers

The worker node with Kata Containers was installed using one of the official guides, that installs Kata Containers together with containerd.

One way to install Kata Containers is to join the node with containerd to the Kubernetes cluster and then install two yaml configuration files Kata automatically. This has failed as the pods created from it get stuck without an error message. [Comd]

Therefore another installation method was choosen. Kata Containers and containerd was first installed with the help of a script `kata-manager.sh`. Since nothing else was explicitly requested, QEMU is used as the hypervisor here [kat24c]. Next, the Container Network Interface (CNI) plugins and cri-tools were installed. For this, the two required GitHub repositories were downloaded and built with make and a provided build script. Next, the CNI plugins and cri-tools were installed. For this, the two required GitHub repositories were downloaded and built with make or a build script. The containerd configuration file `/etc/containerd/config.toml` was then adapted according to the given template, as was the cri-tools configuration file under `/etc/cni/net.d/10-mynet.conf` and that of cri-tools under `/etc/crictl.yaml`. [Comc]

This completed the setup and containers could be successfully started with the Kata Containers Runtime using the ctr command.

```
$ sudo ctr image pull docker.io/library/busybox:latest
$ sudo ctr run --cni --runtime io.containerd.run.kata.v2 -t --rm docker.io/library/b
```

The node was then added to the Kubernetes cluster using `kubeadm` as mentioned in section 3.1.1. Figure 7 shows all containers and their respective runtime after a successful join.

```
cloud@worker-kata:~$ sudo ctr --namespace k8s.io containers ls |  cut -d\
IMAGE                                    RUNTIME
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-npc:latest    io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.kata.v2
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-npc:latest    io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runtime.v1.linux
docker.io/library/nginx:latest           io.containerd.kata.v2
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
registry.k8s.io/kube-proxy:v1.28.4       io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runtime.v1.linux
registry.k8s.io/kube-proxy:v1.28.4       io.containerd.runtime.v1.linux
```

Figure 7: A list of all containers running on the Kubernetes node with the Kata Runtime.

3.1.3 Installing a worker node with Firecracker

Firecracker is often used together with Kata Containers, especially for an installation for Kubernetes. A promising variant without Kata Containers is firecracker-containerd, in which the firecracker MicroVMs are controlled by containerd. However, the connection to Kubernetes and the Container Runtime Interface (CRI) conformance required for this has not yet been guaranteed, as described in the Roadmap section of [fir24].

Nevertheless, this installation variant was tried out first. The current versions of Docker and Go were installed first. Then a current Linux kernel was downloaded from

Amazon. Then the firecracker-containerd GitHub repository was downloaded. The next step was to build in the repository, but the make command failed with an unhelpful error message consisting only of the name of a Go module and error code 1, which made it very difficult to fix. It was finally fixed by using an older Go version: v1.17.13 instead of v1.21.8. This allowed me to run all make commands. After all the required components were successfully built and linked in the `PATH`, an additional containerd configuration file had to be created under `/etc/containerd/`, for which a template was provided. Next, the devmapper snapshotter must be configured. This requires a thinpool device. A script that performs the complete configuration is provided in the repository and only had to be executed. The containerd runtime plugin was then configured under `/etc/containerd/config.toml` using a given template. An image could then be pulled as a test with firecracker-ctr and then started, whereby the configuration file and the socket had to be specifically specified. A Kata firecracker container could then be started. [kat24a]

To use this runtime after the successful installation as Kubernetes runtime, the join command from the 3.1.1 section was supplemented by the configuration parameter `-cri-socket unix:///run/firecracker-containerd/containerd.sock`, so that not the normal containerd but the one used by firecracker is used. This was needed as the normal containerd and the containerd with firecracker cannot run side by side as they require contradictory settings (e.g. the root parameter, which specifies the path to the binaries, and others). The join command results in the following error:

```
[ERROR CRI]: container runtime is not running: output: time=
"2024-02-14T16:15:05+01:00" level=fatal msg="validate service connection:
CRI v1 runtime API is not implemented for endpoint
\"unix:///var/run/firecracker-containerd/containerd.sock\": rpc error:
code = Unimplemented desc = unknown service runtime.v1.RuntimeService"
```

This means that this runtime is not compatible with the Kubernetes interface CRI.

Therefore, Firecracker is used in this work together with Kata Containers, as in many other works, e.g. in [Guo+23] and [Den22]. Firecracker is the virtualization technology and Kata is the low-level runtime. This means that Firecracker replaces QEMU compared to the setup in 3.1.2. The corresponding runtime is called `kata-fc`.

Kata Containers was installed first, as described in section 3.1.2. However, a different hypervisor, i.e. Firecracker, was selected here. Then the binaries for Firecracker and Jailer, a program to isolate the Firecracker process, should be downloaded according to the instructions. The specified download paths were not (or no longer) up-to-date. Instead, a tar archive containing the two desired binaries was downloaded and unpacked. These were then linked in the `PATH`. The containerd plugin devmapper must then run without errors, which has not yet been the case. In order to configure it correctly, a script was supplied which issued further instructions. After these were executed, devmapper ran without an error message. Next, as with the Kata containers, the devmapper snapshotter must be configured using a supplied script. Then Kata Containers had to be configured so that Firecracker can also be used correctly. To do this, the configuration file created during the build had to be copied to a directory read by the kata-runtime. The last thing to be adjusted was containerd. To do this, a shim file `/usr/local/bin/containerd-shim-kata-fc-v2` must be created as specified and the containerd configuration file must be extended to include this runtime. [kat24a]

Now an image can be pulled with the `cri` command and a container can be started with it:

```
cloud@worker-fire:~$ sudo ctr --namespace k8s.io containers ls |  c
IMAGE                                    RUNTIME
docker.io/weaveworks/weave-kube:latest   io.containerd.runc.v2
registry.k8s.io/pause:3.6                io.containerd.runc.v2
docker.io/library/nginx:latest           io.containerd.kata-fc.v2
docker.io/weaveworks/weave-kube:latest   io.containerd.runc.v2
registry.k8s.io/kube-proxy:v1.28.4       io.containerd.runc.v2
registry.k8s.io/pause:3.6                io.containerd.kata-fc.v2
registry.k8s.io/pause:3.6                io.containerd.runc.v2
docker.io/weaveworks/weave-npc:latest    io.containerd.runc.v2
```

Figure 8: A list of all containers running on the Kubernetes node with the Kata-Firecracker Runtime.

```
$ sudo ctr images pull --snapshotter devmapper \
    docker.io/library/ubuntu:latest
$ sudo ctr run --snapshotter devmapper --runtime \
    io.containerd.run.kata-fc.v2 -t --rm docker.io/library/ubuntu
```

In the following, the combination of Kata Containers and Firecracker is referred to simply as Firecracker and the combination of Kata Containers and QEMU as Kata Containers.

3.1.4 Installing a worker node with gVisor

gVisor was installed manually using the official documentation. A short bash script was supplied for the installation, which downloads the runsc runtime and moves it to `PATH`. An install command for the runsc runtime must then be executed. [Autb].

Then containerd had to be configured using its configuration file so that this runtime can also be used correctly [Comb]. The suggested tests failed. First a sandbox was created and then a container was created in this sandbox. However, the sandbox became inactive after just one second, which meant that the container could not be created.

After the nodes were added to the Kubernetes cluster like the other workers, it turned out that the runtime in the cluster still worked as desired. Figure 9 shows the running containers and their runtime.

```
cloud@worker-gvisor:~$ sudo ctr --namespace k8s.io containers ls |  cut -d\
IMAGE                                    RUNTIME
registry.k8s.io/pause:3.6                io.containerd.runc.v2
docker.io/weaveworks/weave-npc:latest    io.containerd.runc.v2
docker.io/weaveworks/weave-npc:latest    io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runc.v2
docker.io/library/nginx:latest           io.containerd.runsc.v1
registry.k8s.io/kube-proxy:v1.28.4       io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.runc.v2
registry.k8s.io/kube-proxy:v1.28.4       io.containerd.runc.v2
registry.k8s.io/pause:3.6                io.containerd.runtime.v1.linux
docker.io/weaveworks/weave-kube:latest   io.containerd.runtime.v1.linux
registry.k8s.io/pause:3.6                io.containerd.runsc.v1
```

Figure 9: A list of all containers running on the Kubernetes node with the gVisor Runtime.

3.2 Test tool clusterloader2

The clusterloader2 is an official, highly configurable testing tool from Kubernetes for performance and scalability. It was published on GitHub[5]. It can test the availability of cluster's control plane, reaction of the cluster to failed nodes, time required to create objects (e.g. deployments and pods), the CPU and memory usage, the database size of etcd and many other metrics.

First, the GitHub repository was cloned. Go was then installed using the Go version manager (GVM). Instead of a Kind managed cluster,, as suggested in the guide, the local cluster described in the previous chapters is used. Other options are a cluster managed by gce, gke, kubemark, aws, vsphere and skeleton. The configuration file described was adopted and deployment was adapted to the cluster so that a test run is always executed on a specific node with a specific runtime (see Appendix B.1). [kub23]

The following command is used to run tests:

```
go run cmd/clusterloader.go --testconfig=${HOME}/perf-configs/config.yaml \
    --provider=local --kubeconfig=${HOME}/.kube/config --v=2
```

The test results are written to standard error. First there are many descriptive debug messages output. When the tests are completed, the test details are output in JSON format. Three different test sets are output. Once for so-called stateless pods, which are the standard pods, then once for statefull pods, which are not used in this setup and therefore have 0 entered everywhere as the result, and once a summary of both, which is identical to the first in this setup. Different metrics are output for each test set. The metric used here is called `pod_startup`. For each metric, three values are specified next to the unit: The time required until 50%, 90% and 99% of the pods under consideration have fulfilled the corresponding metric. Finally, there are many descriptive debug messages again.

A self-written script simplifies the execution of systematic tests (see Appendix B.2). Various things can be configured in the script: There is a list of nodes, each node has a list of runtimes. There is the number of pods that are to be created simultaneously, as well as the number of runs per node-runtime combination. The script then executes the clusterloader2 test multiple times on each node-runtime combination. The relevant data is extracted from each execution, so that at the end a list with the respective results is output for each node-runtime combination. The script was executed in a tmux session so that the script can continue to run despite a disconnection of the VM.

4 Results

4.1 Installation

An important aspect of the usability of container runtimes in Kubernetes is the difficulty of installation (research question R.1). Containerd is very easy to install and there are many instructions and forum posts that help, describe in detail and address possible problems. Kata Containers is a little more complex. Although there is also a seemingly very simple method, it is difficult to debug and there is less help with errors. The method used is more complex, but the instructions are very detailed, which means that hardly

[5] `https://github.com/kubernetes/perf-tests/tree/master/clusterloader2`

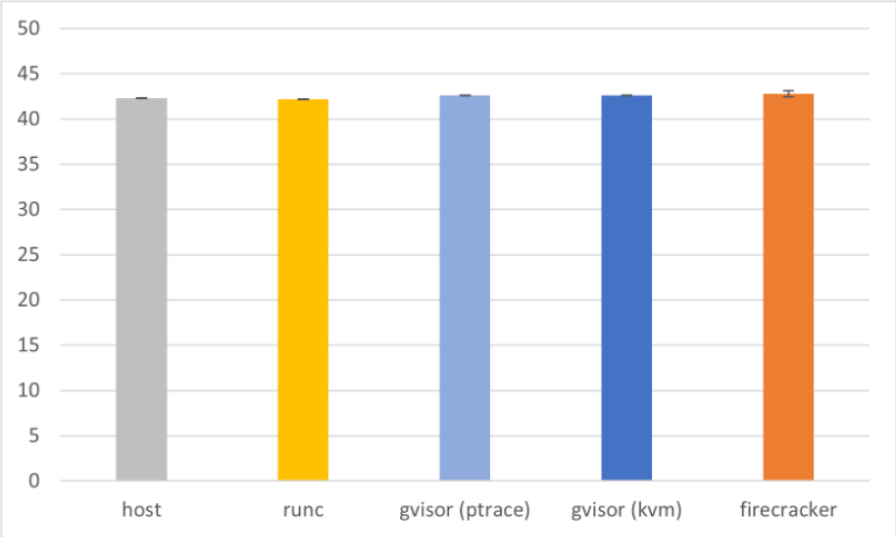

Figure 10: CPU benchmark for different runtimes [Den22].

any problems have arisen. The installation of Kata Containers is therefore fine. The installation of gVisor is short, but the suggested tests fail. As a result, I spent a lot of time troubleshooting, only to find that the runtime still works in the Kubernetes context. So overall, installing gVisor is easy, but the instructions are very misleading. Installing Firecracker was difficult because Firecracker is not designed to be used in a Kubernetes cluster. Using Firecracker together with Kata Containers is complex as two systems need to be installed and set up.

4.2 Performance comparison

As Li et al., Wang at al. and Dendauw all have shown in their experiments, the CPU performance of the four runtimes is almost identical [Guo+23] [WDL22] [Den22]. Figure 10 shows the CPU Dendauw's performance graph.

In the startup time experiments in this work, 10 pods were created simultaneously for each runtime. Measurements were taken with the clusterloader2 tool (see section 3.2). 100 runs were carried out for each runtime. The results can be viewed in Figure 11 and Appendix 2. They show that RunC, Kata Containers and gVisor have similar boot times. Only Firecracker took considerably longer.

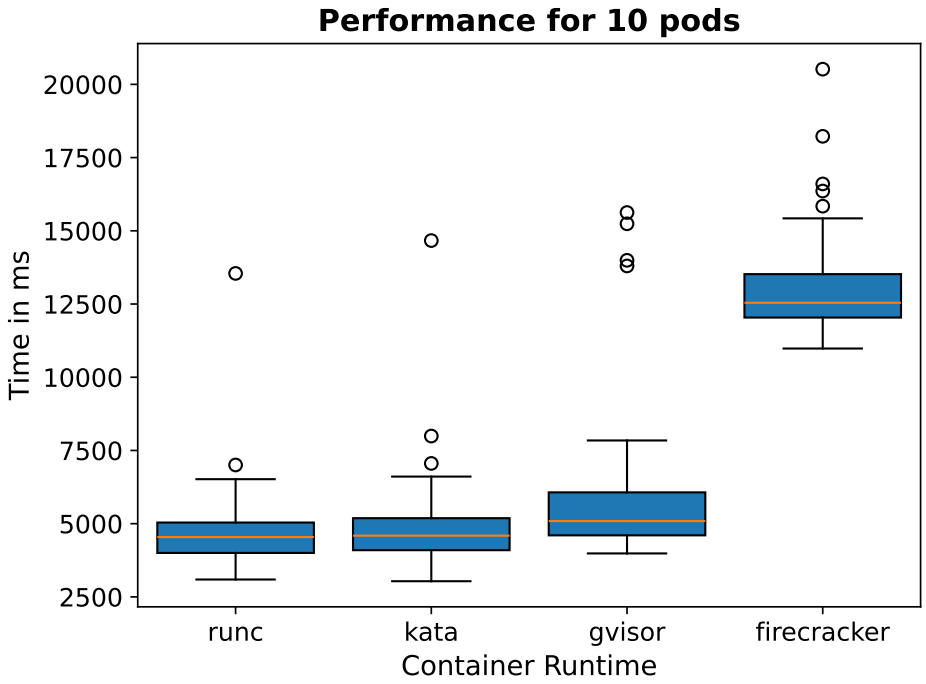

Figure 11: Performance comparison for a deployment with 10 pods for different runtimes, data from 100 test runs in each case are displayed. See Appendix 2 for details.

Table 1: Average container startup times with different runtimes from different experiments.

Source	Runtime	Startup time	Compared to gVisor
Li et al.	RunC	-	-
	Kata Containers	1.43 s	+72%
	Firecracker	1.33 s	+60%
	gVisor	≈ 0.83 s	-
Wang et. al	RunC	1.62 s	−9%
	Kata Containers	2.06 s	+16%
	Firecracker	-	-
	gVisor	1.77 s	-
Dendauw	RunC	551.0 s	−3%
	Kata Containers	-	-
	Firecracker	611.6 s	+16%
	gVisor	568.7 s	-
This work	RunC	4.67 s	−8%
	Kata Containers	4.82 s	−5%
	Firecracker	12.98 s	+129%
	gVisor	5.68 s	-

The startup time was also determined by Li et al. and Wang et al. in a container environment, not specifically in Kubernetes. Li at al. came to the conclusion that gVisor has a very low startup time, while Kata Containers (72% slower than gVisor) and Firecracker (60% slower) have a significantly higher time, with Firecracker being only slightly faster than Kata Containers. Wang et al. also find that gVisor has a low startup time,

9% longer than RunC, which is used by containerd, while Kata Containers takes around 27% longer than RunC. This means that Kata Containers takes around 16% longer than gVisor.

Dendauw also conducted tests in a Kubernetes environment. In his results, Firecracker was only 16% slower than gVisor, while in the tests in this paper it was 129% slower.

All the papers that have compared gVisor and RunC conclude that gVisor is a little slower than RunC, the difference is less than 10%. All papers comparing gVisor and Firecracker conclude that Firecracker is slower than gVisor, but the differences vary between 16% and 129%. In the comparison between gVisor and Kata Containers, the difference varies between −8% and 72%.

4.3 Security model comparison

Wang et al. examined the security of the various runtimes with the help of. This showed that, as expected, RunC clearly has the least isolation and is therefore the most vulnerable. gVisor has the stronger isolation and thus the higher security than RunC and Kata Containers, but this comes at the loss of performance overhead for memory allocation, network and system call. Kata Containers is more secure than RunC , but less secure than gVisor Containers. [WDL22]

5 Conclusion

The analysis of the three innovative runtimes has shown that all three significantly improve security compared to traditional container runtimes such as containerd with RunC. However, the innovative container alternatives have an increased startup time and in some cases further performance restrictions. gVisor has the lowest startup time, Firecracker the highest. In order to decide which container runtime makes the most sense, the specific situation must be considered. If the best possible security and fast startup time is required, gVisor can be a good choice. For other use cases, other factors such as memory bandwidth, memory usage or network throughput must be considered, which were not tested in this study.

It can also be seen that the innovative runtimes are already very advanced, but that further development work is still required in some cases, e.g. for using Firecracker without Kata Containers in Kubernetes.

References

[Ada20] Eric Adams. 2020. URL: https://www.katacontainers.io/collateral/kata-containers-onboarding-deck.pptx.

[Aga+20] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[Auta] The gVisor Authors. *gVisor: GPU Support*. Accessed on: 2024-03-27. URL: https://gvisor.dev/docs/user_guide/gpu/.

[Autb] The gVisor Authors. *gVisor: Installation*. Accessed on: 2024-01-13. URL: https://gvisor.dev/docs/user_guide/install/.

[Autc] The gVisor Authors. *gVisor: Security Model*. Accessed on: 2024-03-26. URL: https://gvisor.dev/docs/architecture_guide/security/.

[Autd] The gVisor Authors. *What is gVisor?* Accessed on: 2024-03-26. URL: https://gvisor.dev/docs/.

[Aute] The Kubernetes Authors. *Kubernetes Components*. Accessed on 2023-11-27. URL: https://kubernetes.io/docs/concepts/overview/components/.

[Autf] The Kubernetes Authors. *Kubernetes Container Runtimes*. Accessed on 2023-11-28. URL: https://kubernetes.io/docs/setup/production-environment/container-runtimes/.

[Autg] The Kubernetes Authors. *Kubernetes: Overview*. Accessed on: 2024-03-28. URL: https://kubernetes.io/docs/concepts/overview/.

[Coma] Firecracker Community. *GPU Support*. Accessed on: 2024-03-29. URL: https://github.com/firecracker-microvm/firecracker/issues/849.

[Comb] gVisor Community. *Containerd Quick Start*. https://gvisor.dev/docs/user_guide/containerd/quick_start/. Accessed on: 2024-01-05.

[Comc] Kata Containers Community. *How to use Kata Containers and Containerd*. https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/containerd-kata.md. Accessed on: 2023-12-18.

[Comd] Kata Containers Community. *kata-deploy*. Accessed on: 2024-02-18. URL: https://github.com/kata-containers/kata-containers/blob/main/tools/packaging/kata-deploy/README.md.

[Den22] Willem Dendauw. *A comparative study of secure container runtimes in a cloud computing environment*. 2022. URL: https://libstore.ugent.be/fulltxt/RUG01/003/063/383/RUG01-003063383_2022_0001_AC.pdf#subsection.5.3.2.

[fir24] firecracker-microvm. "Firecracker-Containerd". In: *GitHub repository* (2024). Accessed on: 2024-03-25. URL: https://github.com/firecracker-microvm/firecracker-containerd.

[Fou] The Linux Foundation. *About the Open Container Initiative*. Accessed on 2023-11-29. URL: https://opencontainers.org/about/overview/.

[Fou17] OpenStack Foundation. "Kata Containers: The speed of containers, the security of VMs". In: (2017). URL: https://katacontainers.io/collateral/kata-containers-1pager.pdf.

[Guo+23] Guoqing Li et al. "The Convergence of Container and Traditional Virtualization: Strengths and Limitations". In: *SN Computer Science* 4 (May 2023). DOI: 10.1007/s42979-023-01827-9.

[JJ19] John Arundel and Justin Domingus. *Cloud Native DevOps mit Kubernetes*. dpunkt.verlag, 2019.

[kat24a] kata-containers. "Configure Kata Containers to use Firecracker". In: *GitHub repository* (2024). Accessed on: 2024-03-24. URL: https://github.com/kata-containers/kata-containers/blob/main/docs/how-to/how-to-use-kata-containers-with-firecracker.md.

[kat24b] kata-containers. "Kata-Containers". In: *GitHub repository* (2024). Accessed on: 2024-03-24. URL: https://github.com/kata-containers/kata-containers.

[kat24c] kata-containers. "Utilities: Kata Manager". In: *GitHub repository* (2024). Accessed on: 2024-01-19. URL: https://github.com/kata-containers/kata-containers/blob/main/utils/README.md.

[Kht] Khtan66. *File:Kubernetes.png*. Accessed on 2023-11-25, changed. URL: https://commons.wikimedia.org/wiki/File:Kubernetes.png.

[kub23] kubernetes. *ClusterLoader2*. 2023. URL: https://github.com/kubernetes/perf-tests/blob/master/clusterloader2/docs/GETTING_STARTED.md.

[Lewa] Ian Lewis. *Container Runtimes Part 1: An Introduction to Container Runtimes*. https://www.ianlewis.org/en/container-runtimes-part-1-introduction-container-r. Accessed on: 2024-01-12.

[Lewb] Ian Lewis. *Container Runtimes Part 3: High-Level Runtimes*. https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes. Accessed on: 2024-01-12.

[Mut] Josphat Mutai. *How To Install Kubernetes on Ubuntu 20.04 using kubeadm*. Accessed on: 2024-03-27. URL: https://computingforgeeks.com/deploy-kubernetes-cluster-on-ubuntu-with-kubeadm/.

[ope16] opencontainer-runtime-spec. "Open Container Initiative Runtime Specification". In: *GitHub repository* (2016). URL: https://wking.github.io/opencontainer-runtime-spec/runtime-spec.pdf.

[Ren] Christian Rentrop. *Low-Level Container Runtimes*. https://www.dev-insider.de/low-level-container-runtimes-a-7c2c3ce41340e09d0ffc94e705a6f965/ Accessed on: 2024-01-10.

[Stö24] Marc Stöckel. "Hacker können aus Containern auf Hostsysteme zugreifen". In: *Golem* (2024). URL: https://www.golem.de/news/docker-kubernetes-und-co-hacker-koennen-aus-containern-auf-hostsysteme-zugreifen-2402-181875.html.

[WDL22] Xingyu Wang, Junzhao Du, and Hui Liu. "Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes". In: *Cluster Computing* 25 (Apr. 2022), pp. 1–17. DOI: 10.1007/s10586-021-03517-8.

A Data

Table 2: Performance comparison for a deployment with 10 pods for different runtimes.

Runtime	Metric	Mean	Standard deviation
containerd	50% pods ready	4128.88 ms	793.24 ms
	90% pods ready	4533.79 ms	801.09 ms
	99% pods ready	4689.61 ms	1201.43 ms
kata	50% pods ready	4302.24 ms	801.99 ms
	90% pods ready	4653.57 ms	835.57 ms
	99% pods ready	4816.61 ms	1307.15 ms
firecracker	50% pods ready	12086.14 ms	1341.74 ms
	90% pods ready	12680.12 ms	1309.83 ms
	99% pods ready	12972.27 ms	1475.66 ms
gvisor	50% pods ready	4790.04 ms	913.58 ms
	90% pods ready	5215.96 ms	917.74 ms
	99% pods ready	5683.4 ms	2061.3 ms

B Code samples

B.1 Sample deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
      runtimeClassName: kata
      nodeSelector:
              nodetype: kata
```

Listing 1: Example deplomentthat is to be executed on the worker-kata node with the Kata Containers runtime.

B.2 Clusterloader2 script

```
import subprocess, sys, ast, numpy as np

COMMAND = "cd /home/cloud/perf-tests/clusterloader2 && go run
 ↪  cmd/clusterloader.go --testconfig=${HOME}/perf-configs/config.yaml
 ↪  --provider=local --kubeconfig=${HOME}/.kube/config --v=2"
CONFIG = "/home/cloud/perf-configs/config.yaml"
DEPLOY = "/home/cloud/perf-configs/deployment.yaml"

settings = {"cont": [""],
            "kata": ["kata"],
            "gvisor": ["gvisor"],
            "fire": ["kata-fc"],
            }
nr_pods = 10
nr_iter = 5
data = []

def write_deploy(node, runtime):
    f = open(DEPLOY)
    new_lines = []
    for line in f.readlines():
        if "runtimeClassName" in line:
            if runtime == "":
                new_lines += ["      #runtimeClassName: %s\n" % runtime]
            else:
                new_lines += ["      runtimeClassName: %s\n" % runtime]
        elif "nodetype" in line:
            new_lines += ["              nodetype: %s\n" % node]
        else:
            new_lines += [line]
    if not new_lines[-1][-1] == "\n":
        new_lines += ["\n"]
    f = open(DEPLOY, "w")
    f.write("".join(new_lines))
```

Listing 2: Script to systematically execute the clusterloader2 test (1/3).

```
33 def write_config():
34     f = open(CONFIG)
35     new_lines = []
36     for line in f.readlines():
37         if "Replicas" in line:
38             new_lines += ["        Replicas: %d\n" % nr_pods]
39         else:
40             new_lines += [line]
41     if not new_lines[-1][-1] == "\n":
42         new_lines += ["\n"]
43     f = open(CONFIG, "w")
44     f.write("".join(new_lines))
45 
46 def test():
47     data = []
48     for i in range(nr_iter):
49         print("   Start Iteration", i)
50         result = subprocess.run(COMMAND,
51                                 shell=True, executable="/bin/bash",
52                                 capture_output=True, text=True)
53 
54         dict_str = ""
55         read_dict = False
56         for line in result.stderr.split("\n"):
57             if " StatelessPodStartupLatency_PodStartupLatency" in line:
58                 read_dict = False
59             if read_dict:
60                 dict_str += line
61             if " PodStartupLatency_PodStartupLatency" in line:
62                 read_dict = True
63                 dict_str = "{"
64 
65         try:
66             d = ast.literal_eval(dict_str)
67         except SyntaxError:
68             print(result.stderr)
69             continue
70         lst = d["dataItems"]
71 
72         for e in lst:
73             if e["labels"]["Metric"] == "schedule_to_watch":
74                 data += [e["data"]]
75                 print(e["data"])
76     return data
```

Listing 3: Script to systematically execute the clusterloader2 test (2/3).

```
77 if __name__ == "__main__":
78     write_config()
79     output = ""
80 
81     for node, runtimes in settings.items():
82         for runtime in runtimes:
83             print("Start tests at node %s with runtime %s..." % (node,
               ↪  runtime))
84             output += "Start tests at node %s with runtime %s...\n\n" %
               ↪  (node, runtime)
85             write_deploy(node, runtime)
86             data = test()
87             print(data)
88             output += str(data)
89             output += "\n\n"
90             print()
91             f = open("results", "w")
92             f.write(output)
```

Listing 4: Script to systematically execute the clusterloader2 test (3/3).