Vrije Universiteit Amsterdam

MSc Thesis

Exploring the Performance of Kubernetes-Deployed Containers

Author: Antonios Sklavos (2738064)

1st supervisor:	Animesh Trivedi
daily supervisor:	Matthijs Jansen
2nd reader:	Tiziano De Matteis

A thesis submitted in fulfillment of the requirements for the VU Master's of Science degree in Computer Science

November 21, 2023

Abstract

Cloud computing provides businesses with scalable and cost-effective access to computing resources. There, isolation mechanisms, like containers, play an important role in isolating different users' workloads while aiming to maintain their high-performance expectations.

Although different isolation mechanisms have been extensively compared individually, these mechanisms are not always used in isolation. These mechanisms are typically used through resource management systems, like Kubernetes, where complex environments necessitate coordinated control and efficient distribution of resources. To our knowledge, there lies a noticeable absence of in-depth comparisons of these mechanisms within the environment of resource managers.

Since the orchestration mechanism operates independently of the isolation mechanism, we want to study how different optimisations in isolation and performance of the isolation mechanism affect the performance, scalability and resource utilisation when deploying through orchestration mechanisms.

We conducted rigorous benchmarks on security and performance-focused isolation mechanisms within the context of the Kubernetes orchestration mechanism. We extended the Continuum framework, which automates infrastructure and benchmark deployment, with the support of Kata Containers, a security-focused containerisation technology. We found the Firecracker-based version of Kata containers lacking compared to the default QEMU-based in terms of startup time performance. Moreover, our findings indicate that containerisation technologies that focus on security display compromised performance and scalability characteristics compared to performance-oriented ones. This leads us to the conclusion that no single containerisation technology stands out as universally superior.

The complete code including the infrastructure implementation and experiments is available online on GitHub, at `https://github.com/anskl/continuum`.

Contents

1

Introduction

Cloud computing provides on-demand access to computing resources via the Internet. Given the widespread popularity and occasional misconceptions surrounding the term "cloud computing", it is important to adhere to a well-accepted definition. The National Institute of Standards and Technology (NIST) offers a valuable definition of cloud computing, identifying it by five key characteristics (1): automatic provisioning of resources on demand, access to capabilities over a network through standard mechanisms, the pooling of resources to serve multiple users, swift scaling of resources to meet demand giving an impression of unlimited resources, and the transparent monitoring, reporting, and optimization of resource usage. The cloud computing industry represents a large market with a predicted 600 billion USD in user spending by the end of 2023, according to projections by Gartner (2). This substantial market capitalization illustrates the significance of cloud applications across an extensive range of sectors including education, government, and healthcare, to name a few (3); for example, the U.S. Department of Defence is enabling DevSecOps on F-16s and battleships (4)

In cloud computing, containers are used extensively to isolate workloads from different users and dynamically scale across machines. Containers are resource-efficient and high-performing isolation mechanisms. They allow programs to run in isolated environments, sharing the host operating system kernel, filesystem and resources (5). While offering strong performance compared to alternatives, like virtual machines, containers generally display weak isolation (6). Security is important according to a recent survey by Google Cloud, where it was reported that the top priority for organizations in 2023 is cybersecurity (3). Container security concerns have led to the emergence of security-optimised containers, such as Kata Containers and gVisor.

Container orchestration mechanisms, a significant component of cloud computing, are responsible for managing the deployment and scaling of applications across multiple cloud environments. The container orchestration mechanism we chose to conduct our tests was straightforward: We used Kubernetes (7) since it dominates the container orchestration landscape. In the Cloud Native Computing Foundation's (CNCF) annual survey of 2022, 66% of CNCF end users believe it to be "very" or "extremely important", while 19% consider it "important" (8). Moreover, in the same survey, Kubernetes is quoted as "emerging as the 'operating system' of the cloud". In a different survey by RedHat, 70% of the IT executives polled were employed by Kubernetes-using companies (9).

To evaluate the differences between different isolation mechanisms, we used the Continuum framework, which automates the deployment and benchmarking of infrastructure, software and applications (10).

1.1 Context

The need to isolate workloads on the same machine appeared in the early days of computing. The appearance of virtual machines and virtual machine monitors (VMMs) marked the beginning of the current era of isolation mechanisms. The VMMs are expected to behave similarly to physical hardware: they export an abstraction of the physical hardware (11). That abstraction enables any software able to run on the hardware to be also able to run on the abstraction (12). The abstraction of virtualisation adds a performance overhead that has been deemed unwanted in many scenarios. That sparked the creation of Linux containers. The containers, in their simplest form, use mainly Linux features and utilities to isolate workloads from one another. Generally, different container technologies have one thing in common: they share the host OS kernel. In their case, the isolation is weaker when compared to Virtual Machines; for example, sharing the host kernel means that a security vulnerability in the host kernel affects the total amount of containers deployed on the host (6). In contrast, virtual machines employ their own kernel and do not share the host one. The great performance of containers (usually near-native) has led to them being the de facto form of isolation mechanism in the cloud.

As the technology matured, security became a greater concern, which became apparent with the appearance of tools like Firecracker, which is a cloud-optimized VMM, and Kata Containers which automate the popular technique of wrapping containers inside VMs as seen in (6), (13).

Containers and VMs have been compared extensively (together with other isolation mechanisms like Unikernels), for example in (14), (15), (16). However, in the context of orchestration mechanisms, there is a lack of comprehensive comparison of different technologies, which leads us to our Problem Statement.

1.2 Problem Statement

Container orchestration mechanisms, such as Kubernetes, generally default to using containers with a performance orientation as their isolation mechanism (runtime) due to their advantages, including but not limited to their rapid elasticity property. However, the weak points of such containers, especially when compared to other isolation mechanisms, are thus inherited in the whole orchestration stack, potentially affecting a wide range of users. Security is a growing concern in the context of cloud computing (3), and thus acknowledging these inherited weak points is crucial. These weaknesses could introduce vulnerabilities or performance bottlenecks that are amplified in a clustered environment where multiple applications and services rely on the efficiency of the underlying containers.

We examine the performance implications of containers with a wide range of performance and isolation properties, all in the context of an orchestration mechanism (in our case Kubernetes) with the goal of facilitating informed decision-making based on different security/isolation and performance needs.

1.3 Research Questions

Our objective is to gain a thorough understanding of how the orchestration mechanism can influence its deployed containers. The formulation and addressing of the following Research Questions can aid in this goal. As previously mentioned, we chose to use Kubernetes as our container orchestration mechanism.

- **RQ1:** How does the startup time of different containerised platforms compare in Kubernetes?
- **RQ2:** How does the scalability of different containerised platforms compare in Kubernetes?
- **RQ3:** How does the resource usage of different containerised platforms compare in Kubernetes?

The answers to the above questions are of a quantitative nature, which can help in the understanding of underlying patterns, trends, and relationships within the data. Startup time is an important metric in the context of cloud computing and microservices architecture, where rapid scaling and fast response times are crucial for handling fluctuating workloads. The scalability property of a runtime directly affects consistent performance and reliability. Knowing the resource usage of containerised applications is significant nowadays with a scarcity of available hardware resources, as well as environmental concerns.

1.4 Research Methodology

To answer the above Research Questions, the following methodologies were employed throughout the project:

- **M1:** Benchmarks: the design and execution of appropriate benchmarks. The benchmarks conducted are extensively mentioned in the Section 3.2 (Design).
- **M2:** Collecting operational traces: Low-level information made the use of operational traces necessary.
- **M3:** Open source software and reproducible experiments: The experiments were conducted on open source software, which is publically available and the experiments are reproducible and accessible.

1.5 Thesis Contributions

This thesis makes the following contributions.

- We have developed a framework for deploying and benchmarking, aimed at comparing different container platforms on Kubernetes.
- We compare various container platforms with different benchmarks that lead to the answers to the above research questions.
- We compare storage backends for containerisation technologies on Kubernetes.

1.6 Plagiarism Declaration

I confirm that this thesis work is my own work, is not copied from any other source (person, Internet, or machine), and has not been submitted elsewhere for assessment. Parts of my survey on isolation mechanisms (5) are used mainly in the background section.

1.7 Thesis Structure

In the next chapter, we will present detailed background information about isolation mechanisms, container orchestration and Continuum - the framework that allowed us to automate the deployment of infrastructure and benchmarks in a virtualised environment. The following chapters are concerned with the design, implementation and evaluation of our system. Next, we go through related work and then conclude our report. The appendix contains detailed reproducibility information as well as a self-reflection chapter.

2

Background

In order to explore the performance, scalability and resource utilisation implications of container orchestration mechanisms (in our case Kubernetes), we need to lay the foundational background of isolation mechanisms, container orchestration and the Continuum Framework - the tool that has allowed us to compare different scenarios in a reproducible, scientific way.

2.1 Isolation mechanisms

Isolating processes from different users is challenging without compromising performance, and no one-solution-fits-all mechanism exists (5). In security-critical scenarios, full virtualisation, where processes are isolated in separate, virtualised and unmodified operating systems, can offer a solution. Performance overhead is unavoidable in this case, stemming from the abstraction of virtualisation: The hypervisor which manages the virtualised operating system is an intermediary that must handle the translation of virtualised instructions to physical ones. On the contrary, having relaxed security requirements offers the ability to limit some isolation-related overheads (for example, the need for separate operating systems for different users). Consequently, performance and isolation can be contradictory, forming a decision dilemma for isolation mechanism designers.

2.1.1 Virtual Machines

Virtual Machine Monitors (Hypervisors) are pieces of software that enable the operation of virtual machines by making the hardware resources of their hosts available to them (17). Having privileged access to hardware resources, they act as an intermediary between the hardware and virtual machines and have the ability to enforce security and performance

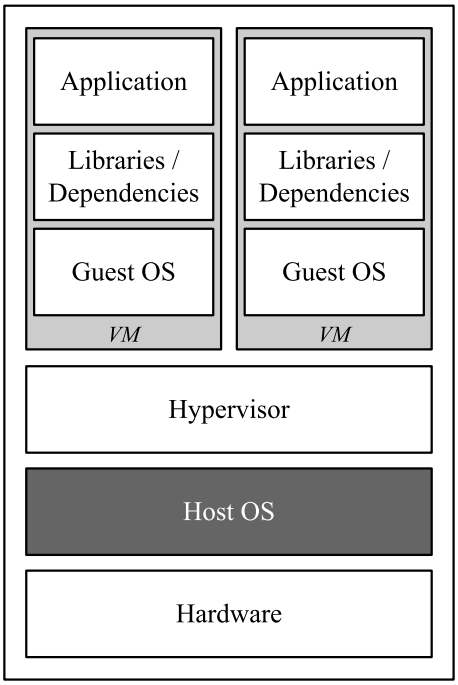

Figure 2.1: Hypervisors (Type-2) Architecture

policies (18) (12). The VMMs are expected to behave similarly to physical hardware (11); they export an abstraction of the physical hardware that enables any software to run on the hardware to be also able to run on the abstraction (12). The isolation they offer between different systems is arguably their most important contribution (18).

Widely used open-source Hypervisors include QEMU, KVM, XEN and Firecracker (5). Hypervisors allow for a high degree of isolation between the hardware and the virtual machine, as well as between different software on the same hardware (11). A significant disadvantage comes from the isolation itself: having each VM carry its copy of the OS and other resources limits the number of VMs one can deploy on the same system (19). Additionally, the abstraction of the hardware that the VMM provides creates an overhead that a higher layer of software will not be able to remove (16).

Figure 2.1 shows a simplified overview of the architecture of Virtual Machine Monitors (VMM). There, we can see that the hypervisor itself is an intermediary between the host OS and the virtual machines. Each virtual machine carries its own guest OS.

Modern VMMs make use of techniques such as para-virtualisation and CPU-supported virtualisation extensions to enhance VM performance. These techniques are not reflected in the simplified Figure 2.1.

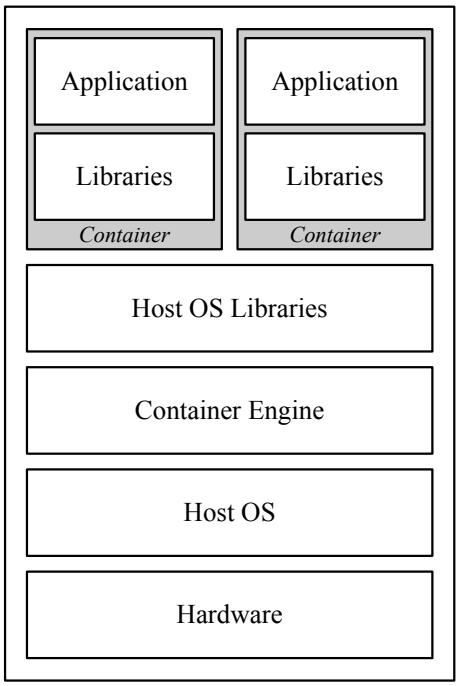

Figure 2.2: Container Architecture

2.1.2 Containers

Containers allow programs to run in isolated environments, sharing the host operating system kernel, filesystem and resources (19) (20) (6). They originally came into view to replace VMs as faster alternatives (19).

Containers use a combination of Unix and Linux kernel extensions and other tools to confine processes into their own execution environment. Most notably, kernel `namespaces` can be used to limit the resources (e.g. process IDs, file names) that a process has access to. The Linux kernel feature `cgroups` is responsible for limiting the resource usage (e.g. CPU, disk access) of a process. The Unix operation `chroot`, is a Linux tool that restricts a process' access to a specific directory. In fact, `chroot`, as well as FreeBSD `jails`, can be regarded as early forms of containerisation. Figure 2.2 shows the general architecture of Containers.

Containers can offer advantages in the performance domain: network, disk, computing, and memory performance overhead can be zero to little (21) (15). In fact, their overhead can be so small that the performance can be comparable with the equivalent bare-metal one (22). This is a direct consequence of the shared kernel: the syscall execution path is significantly shortened in comparison to virtual machines(20). Due to their more negligible

overhead compared to VMMs (albeit not in every case (16)), they allow for increased density - the amount of isolation unit instances that a machine can host, as well as smaller disk images (15).

Containers do not come without disadvantages. Isolation is typically limited (compared to Hypervisors); for example, if a container stresses the kernel with system calls, it will not be able to handle system calls from other containers (performance isolation). A malicious container can take advantage of this by causing a denial-of-service attack (23) (security isolation); Overprovisioning can lead to DoS attacks also (11). Due to its shared nature, a compromised kernel affects all containers (6). In their case, isolation is understood to be lacking enough that deploying a container inside a hypervisor has been common practice (23) (6) (13).

2.1.3 Other isolation mechanisms

Secure containers (also mentioned as sandboxed container technologies) are security-oriented containerisation platforms. They aim to balance performance and isolation using kernel features like `namespaces` as well as leveraging hardware-based isolation (24). As a representative example, we examine Kata Containers.

Kata Containers is a tool that aims to combine the benefits of VMs and containers in a single solution (25) (26) (27). Each workload is run in a container and further isolated in its separate VM (and thus separate kernel - the containers' largest attack surface) (25) (28). As the encapsulating Virtual Machine, Kata Containers defaults to QEMU. Firecracker and Cloud Hypervisor are provided as alternatives. In contrast to QEMU, Firecracker lacks block-based storage drivers and device hotplug support. As a result, when using Firecracker with Kata Containers, updating container resources post-boot and device passthrough are not supported features. Kata Containers is essentially a container-optimised VM (28). Figure 2.3 shows a simplified overview of the Kata Containers runtime architecture. Most importantly, each container is encapsulated in a Virtual Machine.

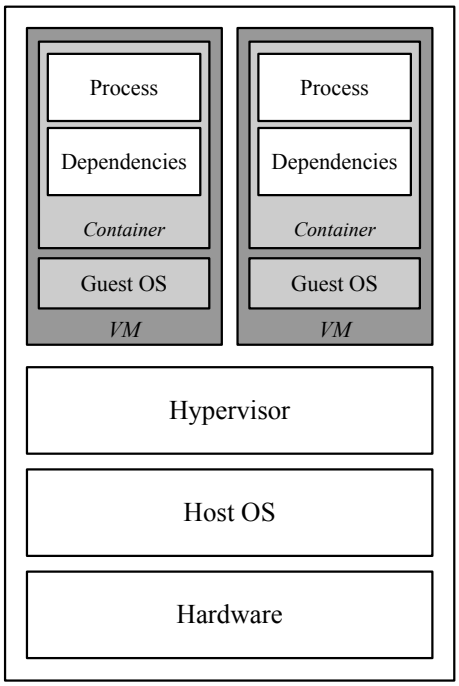

Figure 2.3: Kata Containers Architecture

2.2 Container Orchestration

2.2.1 Docker Swarm

Docker Swarm is a container orchestration mechanism that manages and coordinates container deployments, ensuring seamless communication between containers and utilizing software-defined networks (29). Swarm improves the base node's resource efficiency and distributes containers around several nodes to help with load distribution.

2.2.2 Kubernetes

Kubernetes is an open-source platform designed to automate the management of containers (30). A Kubernetes setup is commonly referred to as a Kubernetes cluster. Every Kubernetes cluster comprises a collection of worker machines, known as nodes and the control plane which makes global decisions about the cluster. Kubernetes evolved from the Borg and Omega container-management systems at Google (7). It is currently the most dominant container orchestration mechanism (8). Figure 2.4 shows an overview of the architecture of a Kubernetes cluster.

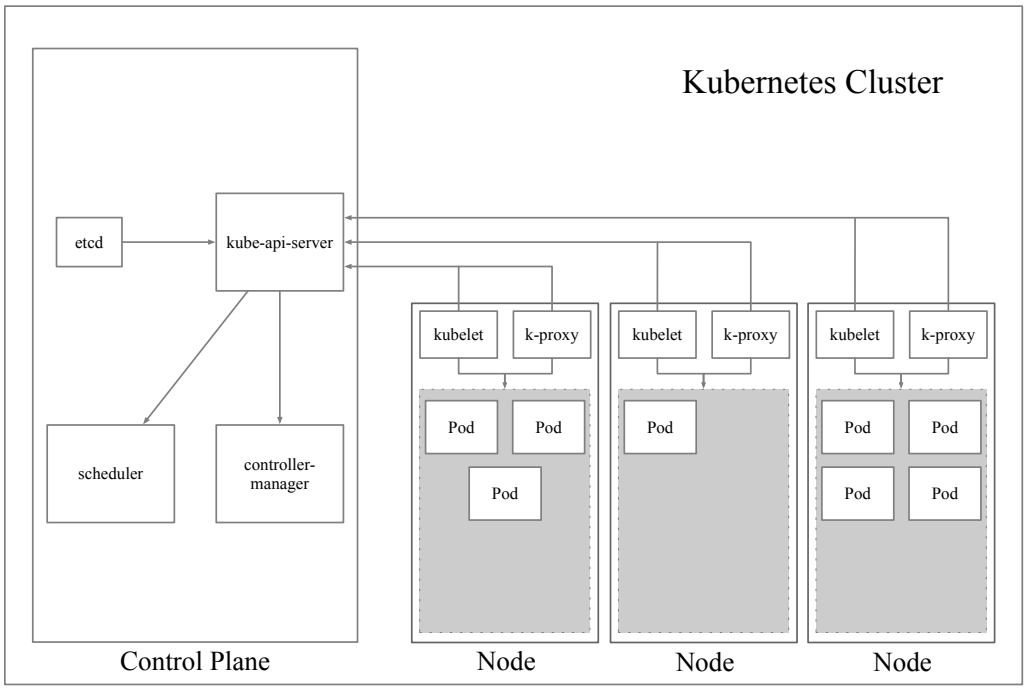

Figure 2.4: Kubernetes Cluster Architecture

2.3 The Continuum Framework

Different tools and platforms may have different architectures, configurations, and workloads, which makes it difficult to compare multi-machine workloads fairly and objectively. A standardised framework for benchmarking distributed systems is required to objectively and accurately assess the trade-offs and bottlenecks of different isolation mechanisms. The Continuum Framework fulfils the above requirement by providing an automated way of deploying and benchmarking of infrastructure. With companies like AWS and GCP, the cloud provides extensive computing, storage, and resource management services along with large-scale infrastructure (10). Additionally, Continuum automates the software stack installation.

Below is an example of a Continuum configuration file (the runtime option is part of the framework's development as part of this thesis).

```
1 [infrastructure]
2 provider = qemu
3
4 cloud_nodes = 2
5 cloud_cores = 8
```

```
cloud_memory = 60
cloud_quota = 1.0

[benchmark]
resource_manager = kubecontrol
runtime = kata-fc

application = empty
```

The infrastructure section defines QEMU as the virtual machine technology representing the 2 physical machines - one of them is the Kubernetes control plane on the node), each one with 8 cores and 60 GiB of memory. The benchmark section defines that the "empty" application should be deployed on the node, with a Firecracker `Kata` Containers runtime (isolation mechanism).

3

Design

Evaluating the impact of a container orchestration system on the performance of an isolation mechanism presents challenges, primarily because of the inherently distributed nature of such environments. When dealing with such distributed systems, there are numerous variables and factors that can affect the performance. Given these complexities, it's crucial to establish a setup that can be consistently replicated, with well-defined parameters. This approach ensures that external variables are minimized, paving the way for more accurate and as objective as possible outcomes.

We have split the Design of the project into two distinct parts. The first part consists of the requirements of the infrastructure and the container orchestration mechanism. The second part is regarding the design of the benchmarks/tests that will help evaluate the differences between the various execution runtimes.

3.1 Infrastructure and Orchestration Requirements

To address the research questions we have posed, it is essential to establish clear requirements for the system in question. We begin by defining the requirements of the infrastructure, which will serve as the deployment target for our experiments.

3.1.1 Infrastructure Requirements

R1. Consistent performance: Important, as it enables increased confidence in the results and minimizes unpredictability.

R2. Ability to configure system resources: Allows for testing different performance scenarios leading to more informed conclusions.

R3. Ability to configure deployment resources: This enables analyzing the scaling behaviour of the system, as well as understanding its performance thresholds and resource efficiency under varied configurations.

3.1.2 Orchestration Mechanism Requirements

To ensure the container orchestration mechanism can support the subsequent benchmarks, we have defined the following requirements. These criteria are designed to not only support our benchmarks but to also enable detailed insights into the orchestration process itself.

R4. Support for various types of execution runtimes (isolation mechanisms) It is important to be able to compare different runtimes with various degrees of performance and isolation. This will aid in determining the most appropriate runtime based on security requirements.

R5. Capability to identify various execution stages of orchestration: Having the capability to examine the execution flow through the orchestration mechanism will assist in pinpointing its related bottlenecks, comprehending system performance, and distinguishing it from runtime constraints.

R6. Capability to identify various execution stages of runtime: Being able to trace the distinct stages of runtime execution facilitates a deeper understanding of the runtimes' behaviour, enables comparing different runtimes, and offers insights into potential areas for optimization.

3.2 Design of Benchmarks

Below, we outline the benchmarks designed to thoroughly evaluate the various isolation mechanisms from diverse perspectives. For every test type, consistently deploying the same application across all scenarios is crucial, aligning with **R1: Constant performance**.

3.2.1 Design of Startup Performance Benchmarks

In order to get a good understanding of the startup performance penalty of a deployment on a distributed environment enabled by a container orchestration mechanism, we followed the paradigm below, for each isolation mechanism:

B1: Measure the startup performance of a fixed amount of identical deployments over different numbers of machines.

B2: Measure the startup performance of an equal number of deployments per machine, with the total number of machines varying for each benchmark.

Each one of the points above refers to a collection of benchmarks. The workload in all cases is deployed in parallel.

B1, essentially, tests for strong scalability of the startup times of the setup. The ideal outcome is that doubling the amount of machines results in half the total aggregated time of launching a set amount of pods. The importance of the benchmarks lies in testing the container orchestration mechanism's ability to reduce the total execution time with the addition of more resources.

The benchmarks in the collection **B2**, test for weak scalability of the startup times of the setup; the problem size and machines increase proportionally. There, we test the system's ability to keep a constant execution time, given double the resources and deployments. The ideal scenario is that no additional overhead is added and all independent machines finish their deployments at the same time.

By testing for both strong and weak scalability, we can comprehensively assess the system's startup performance across varying workloads and resource allocations in diverse scenarios with different isolation mechanisms.

3.2.2 Design of Scalability Benchmarks

Here, we assess the scalability of the amount of deployments the system can support (in comparison to the scalability of startup performance above). The scalability tests are important for the following reasons:

1. It is unrealistic to anticipate that a machine will deploy the same number of instances manually as it would using a container orchestration mechanism. Thus, it is important to focus on the potential limitations imposed by the orchestration mechanism itself.

2. It should not be presumed that the system scales in a linear fashion. For instance, doubling the worker nodes from two to four might not necessarily double the deployments from 20 to 40. While this is our hypothesis, empirical data is essential to confirm it.

The primary factors influencing the system's capacity to support the maximum number of deployments are the quantity of worker nodes and the memory allocated to them. Consequently, the benchmarks should include a diverse range of combinations derived from

these variables, allowing for a comprehensive evaluation of the system's scalability. Such a benchmark can be as broad as time and resources allow.

After calculating each combination of the value of the variables of choice mentioned above, we end up with a scalability per worker per memory unit metric.

3.2.3 Design of Resource Usage Tests

Here, we outline the design of tests aimed at understanding the effect of resource usage of an orchestration mechanism on the runtime it targets.

Design of Memory Usage Tests.
To effectively compare the effect of an orchestration mechanism on different isolation mechanisms, it is important to assess the memory overhead of the system. The following blueprint can be followed for each of the different isolation mechanisms of the comparison.

- **Step 1:** Measure the total free memory of the system per worker node before and after the deployments take place.
- **Step 2:** Divide the difference between the two by the number of deployments yielding average deployment memory overhead per deployment.

Design of CPU Performance Tests.
Here we measure the scalability of the performance of the system. To do that, we now switch to a deployment that is computing-intensive. With different combinations of the CPU and number of worker nodes variables, we measure the total execution time of the whole deployment (sum of deployment units). Thus, we can test:

- **B3: Strong Scalability of Performance**
- **B4: Weak Scalability of Performance**

In **B3**, for each isolation mechanism to be tested, we begin by setting a fixed number of deployments and setting a baseline by measuring the total time of the whole deployment execution. Then, we increase the resources (CPU and worker nodes) available to the system. This enables us to test for efficiency under increased resource conditions and determine how those additional resources impact the overall system throughput.

In **B4**, for each isolation mechanism under consideration, we start with a set number of deployments relative to a given resource allocation, establishing a performance baseline.

As we incrementally increase both the deployments and the associated resources (CPU and worker nodes), we gauge how consistently the system maintains its performance. This method helps us assess whether the system can effectively manage a balanced growth in workload and resources while maintaining stable execution capacity. Both of the benchmarks can be used to optimize the resource of time.

4

Implementation

4.1 Runtime Implementation

In this section, we focus on setting up and utilizing Kata Containers (our selected alternative isolation mechanism) and integrating the complete software stack into the Continuum Framework. We opted for Kata Containers as it streamlines the typical process of deploying containers within VMs, supports a wide range of VM implementations, and seamlessly integrates with containerd.

4.1.1 Automating the installation of Kata Containers

Installing the Kata Containers runtime in every worker node is the first step in using it as an alternative Kubernetes runtime. To automatically manage the infrastructure installation, we used `Ansible` files which allow us to define, configure, and orchestrate system configurations in a declarative way. Utilizing alternative container runtimes with Kubernetes is feasible, provided that the chosen runtime is compatible with the Kubernetes CRI (Container Runtime Interface). Notably, our alternative container runtime of choice, Kata Containers, is compatible with both the CRI-O and containerd Shim API. To ensure consistency between the default Kubernetes runtime, runc, and various versions of Kata Containers like QEMU, we chose to use containerd. This decision helps minimize discrepancies across these platforms.

We begin with the installation of the default kata-runtime package which uses QEMU as the wrapping VM. First, we extract the latest Kata Containers release (3.1.3 at the time of writing this) and add a symbolic link to the kata-runtime in the `/usr/local/bin` directory. Then, we create a custom (not mentioned in official documentation) executable file (`/usr/local/bin/containerd-shim-kata-qemu-v2` - naming important) that when

executed, will run the `containerd-shim-kata-v2` binary with the necessary QEMU configuration file:

```
#!/bin/bash
KATA_CONF_FILE=/opt/kata/share/defaults/kata-containers/configuration-qemu.toml
↪ /opt/kata/bin/containerd-shim-kata-v2 $@
```

Finally, we add the following section in the containerd configuration file (`runtimes` section) to make it aware of the newly installed runtime:

```
[plugins."io.containerd.grpc.v1.cri".containerd.runtimes.kata-qemu]
    runtime_type = "io.containerd.kata-qemu.v2"
```

The installation of Kata Containers with Firecracker as the wrapping VM requires additional steps. Firecracker does not support the Overlay Filesystem (overlayfs), which is implemented directly in the Linux kernel and requires the device mapper framework. Thus, we begin by setting up the necessary infrastructure for the devmapper snapshotter to be used with containerd, with the following `bash` script:

```
mkdir -p /var/lib/containerd/io.containerd.snapshotter.v1.devmapper

touch /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/data
truncate -s 20G /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/data

touch /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/meta
truncate -s 10G /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/meta

DATA_DEV=$(sudo losetup --find --show
↪ /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/data)
META_DEV=$(sudo losetup --find --show
↪ /var/lib/containerd/io.containerd.snapshotter.v1.devmapper/meta)

DATA_SIZE="$(sudo blockdev --getsize64 -q ${DATA_DEV})"
LENGTH_IN_SECTORS=$((DATA_SIZE / 512))

dmsetup create devpool --table "0 ${LENGTH_IN_SECTORS} thin-pool ${META_DEV}
↪ ${DATA_DEV} 128 32768
```

Here, we need to explicitly specify the size of the block device that will be used for storing the actual container data. This allows containerd to store both container images and runtime data using the devmapper snapshotter. Additionally, we need to configure the containerd configuration file:

```
[plugins."io.containerd.snapshotter.v1.devmapper"]
    discard_blocks = true
    base_image_size = "10GB" # As defined above
    pool_name = "devpool"
    root_path = "/var/lib/containerd/io.containerd.snapshotter.v1.devmapper"

# change default snapshotter - necessary for Kubernetes deployment
[plugins."io.containerd.grpc.v1.cri".containerd]
    snapshotter = "devmapper"
```

Finally, we need to create the executable file that defines the kata runtimes firecracker configuration file variable (`/usr/local/bin/containerd-shim-kata-fc-v2`):

```
#!/bin/bash
KATA_CONF_FILE=/opt/kata/share/defaults/kata-containers/configuration-fc.toml
↪ /opt/kata/bin/containerd-shim-kata-v2 $@
```

4.1.2 Kata containers as Kubernetes runtime

After verifying that the Kata containers runtime is operational in the worker nodes, we can add them as Kubernetes `RuntimeClass` targets with:

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: kata-fc
handler: kata-fc
```

and

```
apiVersion: node.k8s.io/v1
kind: RuntimeClass
metadata:
  name: kata-qemu
handler: kata-qemu
```

Creating and applying (`kubectl apply -f <file>`) the above files enables Kubernetes to specify a target runtime for a deployment, like the following example of a deployment:

```
parallelism: 1
template:
  metadata:
```

```
4     name: empty-kata-fc
5   spec:
6     runtimeClassName: kata-fc
7     containers:
8     - name: empty
9       image: empty
10      imagePullPolicy: Never
11      resources:
12        requests:
13          memory: "500Mi"
14          cpu: 1
15      env:
16      - name: SLEEP_TIME
17        value: "600"
18    restartPolicy: Never
```

4.2 Implementing kata-runtime Value Retrieval

For Kata Containers, the startup time to launch a container is anticipated to be longer compared to traditional containerization solutions such as runc. This is because a virtual machine needs to be initiated for each container in Kata Containers.

We are not solely focused on the startup time of the container. Instead, we are keen on understanding the intermediate phases involved in launching a VM, initiating a container within that VM, and deploying the workload. This detailed analysis will allow us to compare various VM implementations effectively.

The Kata runtime can generate traces, which can be utilized to evaluate the code's execution path, object relationships, and timing information.

```
1  func (k *kataAgent) startSandbox(ctx context.Context, sandbox *Sandbox) error {
2          span, ctx := katatrace.Trace(ctx, k.Logger(), "StartVM", kataAgentTracingTags)
3          defer span.End()
4  ...
5  }
```

An `OpenTracing` endpoint, like `Jaeger` can receive and export a graphical frontend. The Jaeger frontend was extensively used to understand the call path and different phases of execution of the Kata Containers runtime.

We can subsequently retrieve all the traces and process the unrefined data. This allows us to deduce various execution stages, such as initiating the `kata-runtime`, establishing the

virtual machine (VM), connecting to the VM, and so on. To achieve this, we utilize Python, which is the programming language used in the Continuum Framework's implementation.

```python
# curl request to jaeger endpoint
jaeger_api_url = f"http://{ip}:{port}/api/traces?service=kata&operation=rootSpan&limit=10000"
response_data = requests.get(jaeger_api_url).json()

# Sort each trace's spans based on starTime and sort traces based on startTime
traces = sorted(
    [sorted(trace["spans"], key=lambda x: x["startTime"]) for trace in traces],
    key=lambda x: x[0]["startTime"],
)
```

4.3 Benchmarks implementation

The Continuum Framework automates not only the infrastructure but the deployment of workloads as well. This automation is facilitated through `cfg` files, which act as input parameters for the framework, allowing users to define both the infrastructure and benchmark settings.

To illustrate, consider the example provided below. Here, we're tweaking various parameters like the number of worker nodes, the runtime, and the deployments per worker to experiment with diverse scenarios. In the leftmost `cfg` file, we deploy 25 `empty` applications on 4 different worker nodes running on the Kata Containers runtime with Firecracker as the encapsulating VM.

```
[infrastructure]
...
cloud_nodes = 5
...
[benchmark]
runtime = kata-fc
application = empty
applications_per_worker = 25
...
```

```
[infrastructure]
...
cloud_nodes = 1
...
[benchmark]
runtime = runc
application = empty
applications_per_worker = 100
...
```

5

Evaluation

In this section, we evaluate our system as implemented based on the previous two sections. We deploy our workloads on a Kubernetes cluster and compare different runtimes. The evaluation is focused on three main dimensions: startup performance, scalability and resource usage. The latter is further subdivided into the subsections of memory and CPU usage:

Startup: How do QEMU and Firecracker-based Kata containers' startup times compare with default containerd (runc)?

Scalability: How do QEMU and Firecracker-based Kata containers' scalability compare with default containerd (runc)?

Resource usage: How do QEMU and Firecracker-based Kata containers' memory and CPU usage compare with default containerd (runc)?

5.1 Benchmark results and evaluation

5.1.1 Evaluation of Startup Performance

We begin by determining the startup performance of the three different runtimes. First, we look into the **strong scalability of the startup performance**. The results in Figure 5.1 represent a deployment of a total of 100 applications across one, two and four worker nodes and measure the total time all 100 applications take so that all reach a running state. The horizontal axis represents duration in seconds, and lower (faster) values are preferred. The topmost 3 rows of the vertical axis represent the default Kubernetes runtime (runc), the following two are QEMU-based and Firecracker-based Kata containers.

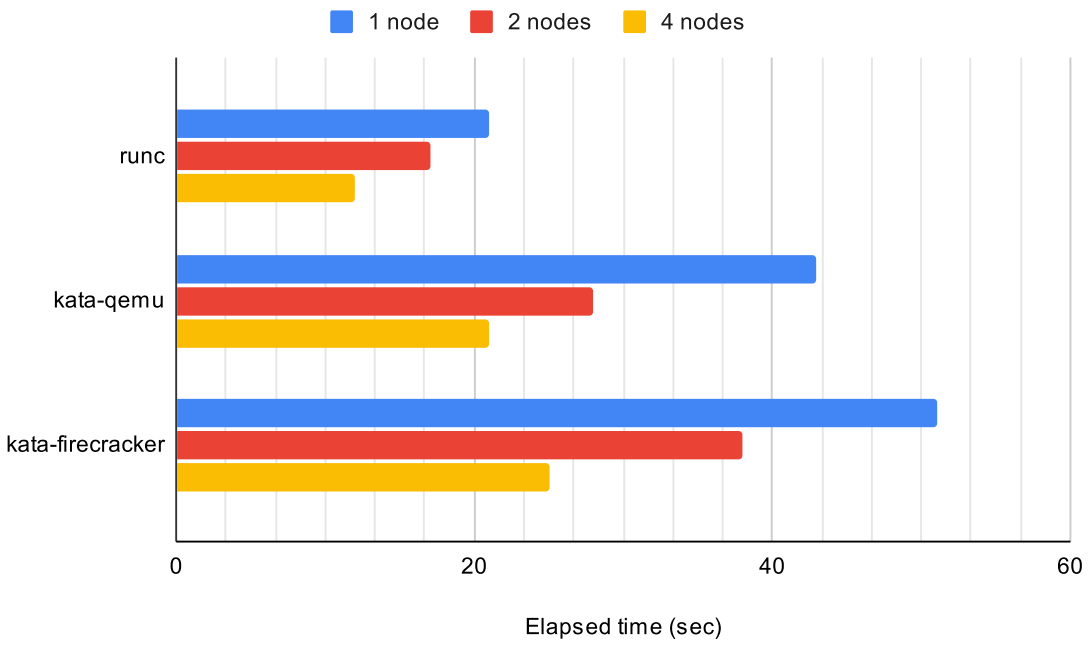

Figure 5.1: Launching a deployment of 100 replicas across 1, 2 and 4 worker nodes

From this result, two main points can be derived. First, we see a scalability of sublinear nature. While adding more resources increases the performance, the performance gains do not increase linearly with the number of resources. Second, Kata Containers' startup time is significantly slower with Firecracker compared to with QEMU. That is unexpected since tests across the board have shown Firecracker to display state-of-the-art boot times, in contrast to QEMU (31), (13).

We thus proceed to investigate this discrepancy. To do that, we continue by plotting the total execution time of each one of the above 100 deployments across two worker nodes, split into different phases of execution - Figure 5.2 and Figure 5.3.

Comparing the two graphs, we see a significant difference: the execution of the Firecracker-based Kata containers follows a linear pattern. Additionally, in the same graph, we can notice that the last container (topmost row) is created around the time the first (bottom-most row) is finished (phase `kata_create_container_and_launch (s)`). This behaviour is not displayed by the QEMU-based Kata Containers Figure 5.2.

Investigating this issue, we came up with the following explanation. Kata Containers with QEMU and Firecrackers differ in the filesystem they use: Firecracker-based Kata

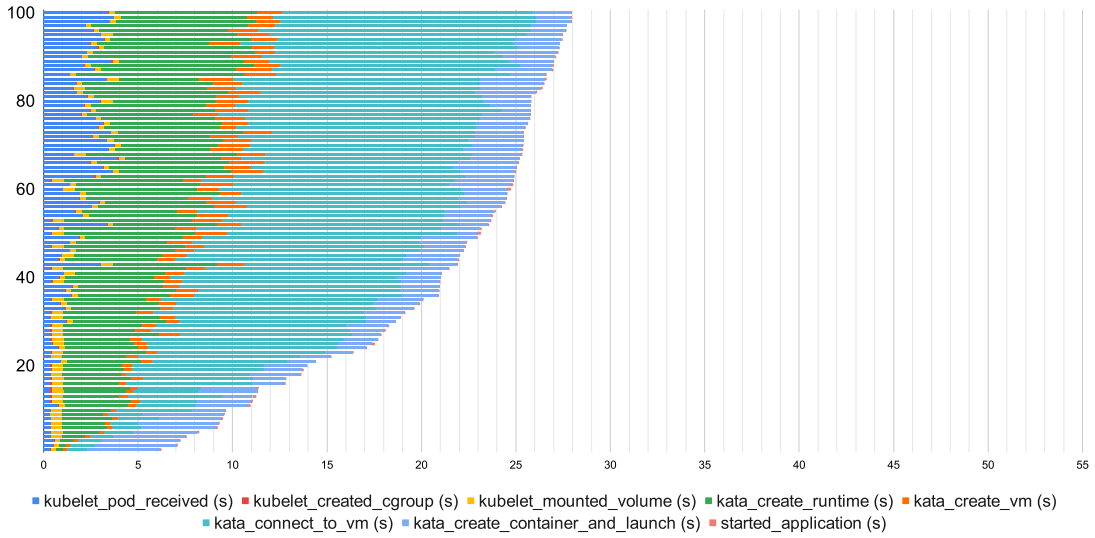

Figure 5.2: 100 applications across 2 worker machines, Kata Containers (QEMU)

containers require the use of the devmapper framework, while the QEMU-based ones use overlayfs by default. Firecracker's use of the devmapper framework stems from its limitation of only supporting block-based storage drivers. To strengthen our case, we proceed by running the same experiment on QEMU-based Kata containers with the use of the devmapper framework. Plotting the different execution phases in this case produces Figure 5.5.

Here we notice a similar behaviour to Figure 5.3, verifying our finding that the devmapper framework was the cause of the Firecracker-based Kata Containers' slower boot performance. Additionally, the last phase of execution (`start_application (s)`) takes significantly longer than the previous two cases. Finally, the final boot time reaches a duration of 53 seconds (up from 28 when using overlayfs). This is significantly slower compared to Firecracker-based Kata containers' 38 seconds and is aligned with performance displayed at isolated tests.

We are thus led to a significant finding: using Kata containers with Firecracker forces the use of the devmapper framework. Compared to the default Kata containers (with QEMU), counter-intuitively, this affects significantly and negatively the boot performance of the containers.

Next, we look into the **weak scalability of the startup performance**. To do that,

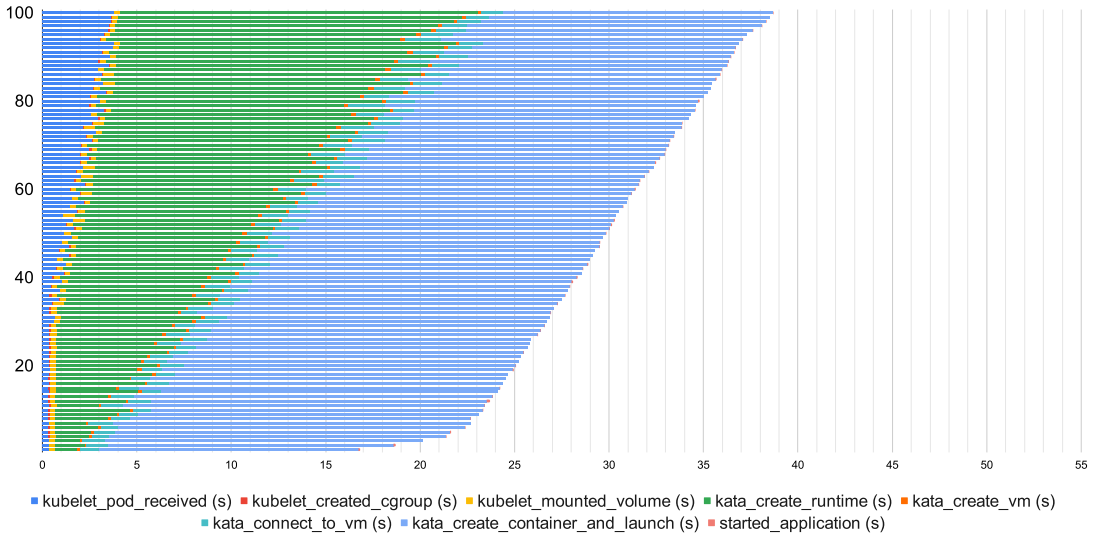

Figure 5.3: 100 applications across 2 worker machines, Kata Containers (Firecracker)

we increase the workload (Kubernetes pods to launch) and worker nodes proportionally. More specifically, for this benchmark we deploy 25 replicas of the workload per working machine (node) and measure the total boot time for a workload of one, two and four worker nodes.

Table 5.6 shows the total time it took until all pods reached a "Running" state, per configuration of an example deployment. Interestingly, in this particular test, we see that runc does not scale well, and in fact, has a 100% decrease in boot time when quadrupling the workload and resources proportionally. Firecracker-based Kata containers show an increase of 10% and 36% boot time speed when doubling and quadrupling the workload and resources proportionally. The best boot time scaling is seen with QEMU-based Kata containers, where we see an 11% and 17% increase in double and fourfold workload and resources.

5.1.2 Evaluation of memory usage

In this section, we measure the average Kubernetes pod memory usage per runtime. To do that, the free memory of a worker machine is measured. Then, we start by deploying 100 replicas of our "simple" application on that machine. After the pods reach a "Running" state, we measure the free memory of the system. We divide the difference of the two by

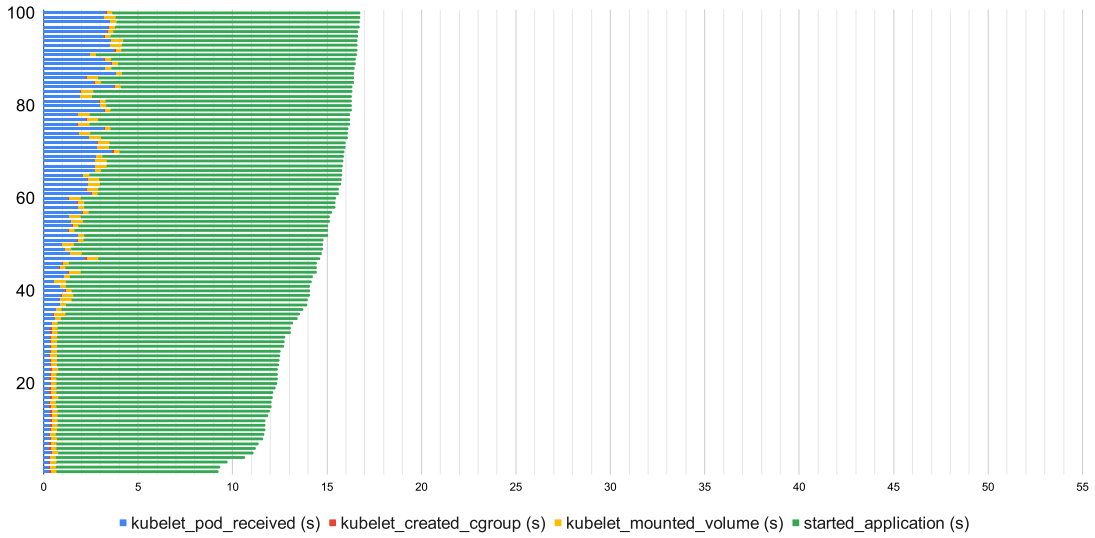

Figure 5.4: 100 applications across 2 worker machines, runc

100 which leads us to the average memory footprint. We the proceed to repeat the process for our three different runtimes.

	average memory footprint (MiB)
runc	10
kata-qemu	152
kata-firecracker	135

Table 5.1: Average pod memory footprint

In Table 5.1, we can see that Firecracker-based Kata containers have a memory footprint of around 10% smaller than those based on QEMU. Moreover, in both cases, the memory footprint is one order of magnitude greater than that of runc (containerd).

It is important to note that simply measuring the memory footprint of each pod's process on the worker node itself is insufficient. The kata-containers runtime spawns other memory-adamant processes and reserves memory during the execution.

5.1.3 Evaluation of CPU usage

For this test, we deployed a 60-second CPU-intensive workload, for different combinations of workload replicas, worker nodes and CPU cores. Code 5.7 shows the Dockerfile that was

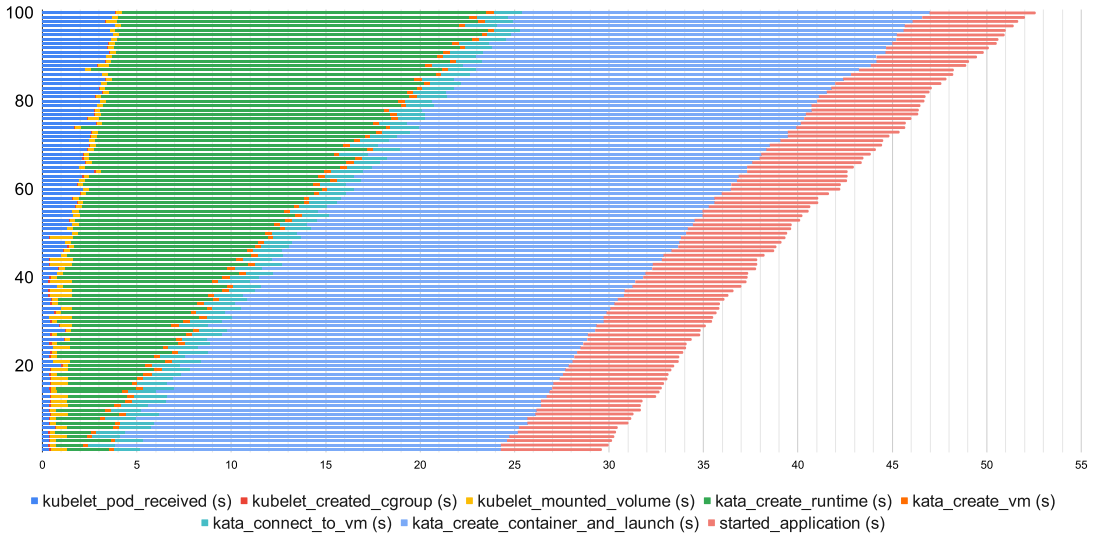

Figure 5.5: 100 applications across 2 worker machines, QEMU-based Kata Containers (devmapper)

used to create the stress test.

In the following figures, we compare different runtimes to understand how they affect CPU-intensive workloads. The horizontal axis represents the duration in seconds - in this specific case, 60 seconds is the theoretical minimum.

First, we evaluate the strong scalability of the CPU. In Figure 5.8, we deploy a constant total of 8 workloads with a 60-second duration across 1,2 and 4 worker nodes. From this graph, we can see that both versions of Kata containers perform in a similar manner and are both slower (4-15%) than the baseline (runc).

In Figure 5.9 we test for the weak scalability of the CPU overhead. There we can see that for each runtime, after doubling the workload and resources, the time stays constant (within a margin of error). The same happens when we quadruple the resources and workload. Again, we see an increased overhead with Kata containers compared to the baseline (runc). These results reflect the expected behaviour of weak scaling.

5.2 Reporting Negative Results

To evaluate **scalability**, our original plan was to measure the maximum number of concurrent pods each worker node can handle, based on a set configuration, for all different runtimes. Our understanding was that the limiting factor in that case would be each worker

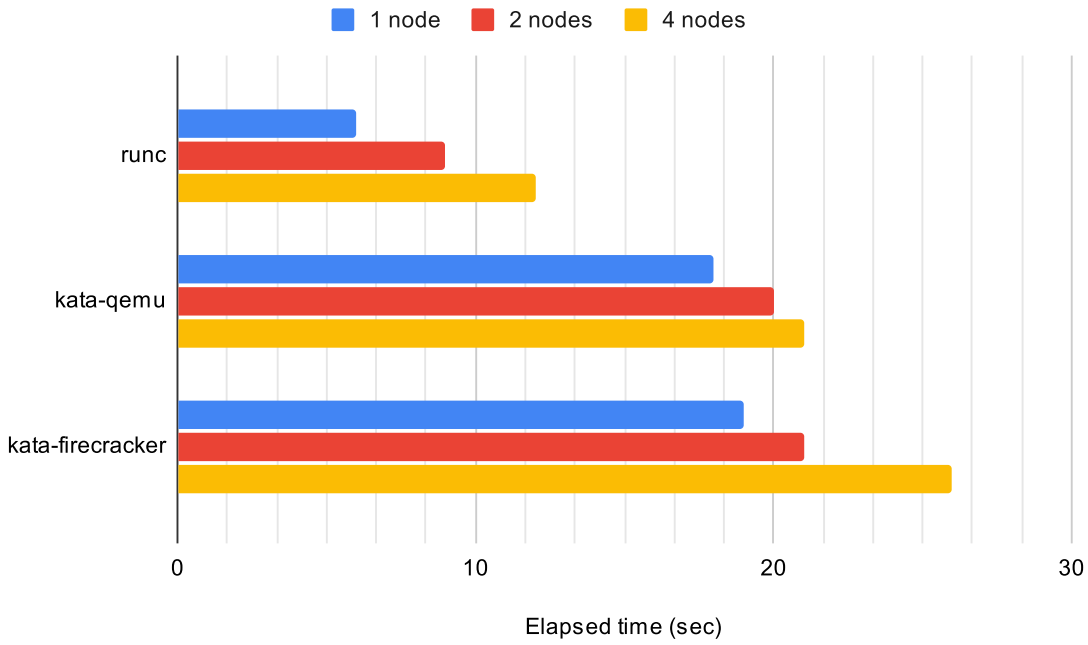

Figure 5.6: Launching a deployment of 25 replicas per node across 1, 2 and 4 worker nodes (weak scaling of boot time)

node's available memory. We measured the memory overhead of each pod in a different experiment, and we expected it to be in the range of 150 MiB in either case of Kata Containers implementations. After making sure the worker had sufficient memory available (128 GiB) and the pods did not request a minimum amount of memory, we launched a workload of 1000 replicas as a first attempt at gauging the scalability. To our surprise, the number of pods that would reach a "Running" phase would max out at 109. We investigated the running pods and proceeded by deleting a pod of a different deployment that was responsible for gathering resource usage metrics. After making sure to delete all running pods and re-deploying the test, we found ourselves in a similar situation: the number of running pods would not go past 110.

After some research, an unexpected limitation from Kubernetes itself came up: In a Kubernetes cluster, the maximum number of running pods per worker node is in fact limited to 110 (32). This led to a decision crossroad. We could either try to implement a modified Kubernetes version or answer scalability indirectly. The former would require a significant time investment due to our lack of understanding of the reasons for the imposed limitation, with limited potential for success. As a result, we chose the latter.

```
1  FROM alpine:3.18.4
2
3  RUN apk add --no-cache stress-ng
4
5  # Default timeout value
6  ENV TIMEOUT=60s
7
8  CMD ["sh", "-c", "stress-ng --cpu 1 --timeout $TIMEOUT"]
```

Figure 5.7: Simple stress image Dockerfile

5.3 Limitations and Threat to Validity

The tests mentioned above were run on a virtual Kubernetes cluster, deployed using the Continuum Framework. Each worker node was a virtual machine itself with a sufficient amount of available memory for each test, and cores pinned to the physical cores of the machine. The latter means that the number of worker nodes was limited by the number of cores (in our case 20) of the physical machine. We opted to not use virtual cores as the results would be misleading and inconsistent, especially in the case of CPU stress tests. However, we expect the virtual nature of our testing environment did not alter the relative differences between the different runtimes.

It would be interesting to test the boot scalability on more worker nodes and find a point of diminishing returns in boot time performance. Additionally, we expect to see different values when deployed on a physical Kubernetes cluster.

The finding that Firecracker-based Kata containers perform worse than QEMU-based ones led to the implementation of a filesystem option in Continuum. This would allow the users to choose a filesystem from the configuration file they use continuum with (assuming it is compatible with their runtime of choice). Unfortunately, in the case of QEMU-based kata containers, choosing the devmapper framework proved to be unstable: there would be currently unidentified reasons some tests would fail. Due to time constraints, that was not examined properly. Having the option to compare Firecracker and QEMU-based Kata Containers with the devmapper framework in the tests above would potentially lead to more findings and a more complete comparison of the two runtimes.

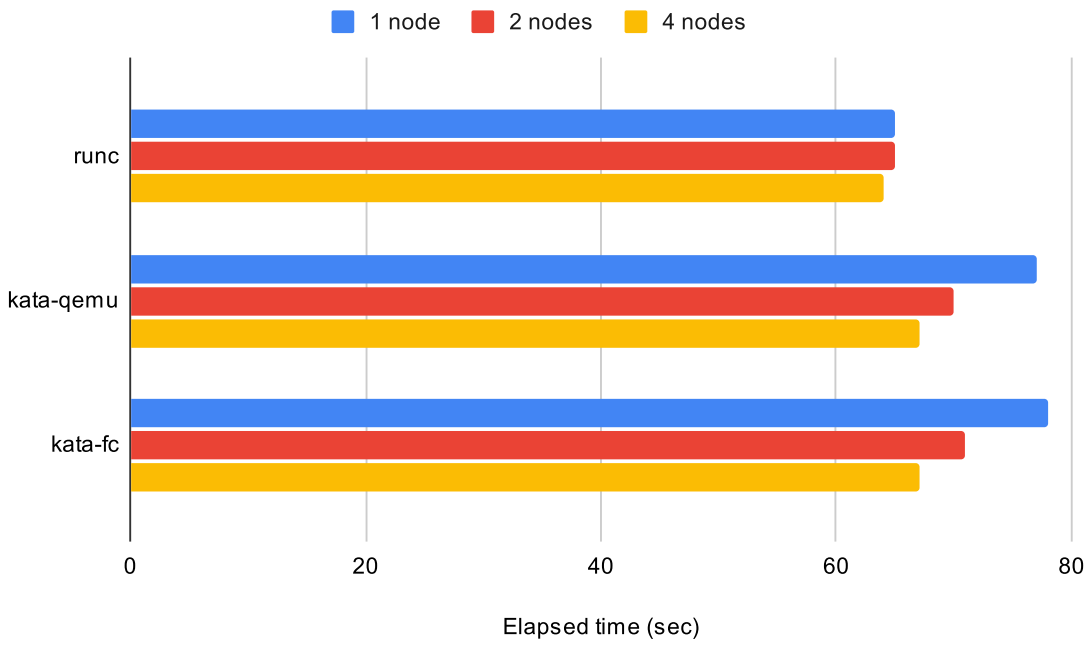

Figure 5.8: Strong scalability of CPU

5.4 Summary

To summarise, in this section, we evaluated the boot time performance and resource usage of QEMU and Firecracker-based Kata Containers and compared them with the default Kubernetes runtime, runc. We found the Firecracker-based Kata Containers reaching a running state in a slower time in comparison to QEMU-based ones. This unexpected finding does not correspond to the expected behaviour of isolated runtimes, where the hyper-specialised Firecracker VM outperforms QEMU in the realm of boot times. We showed that Kata Containers offer significantly slower boot times in comparison to runc. Additionally, we showed that the memory footprint of Kata Containers is one order of magnitude larger than that of runc, in the context of Kubernetes. Moreover, the CPU overhead was in the worst case 15% worse than baseline (runc).

In consideration of the constraints imposed by Firecracker, it is advised to adhere to the default configuration for Kata Containers, which utilizes QEMU.

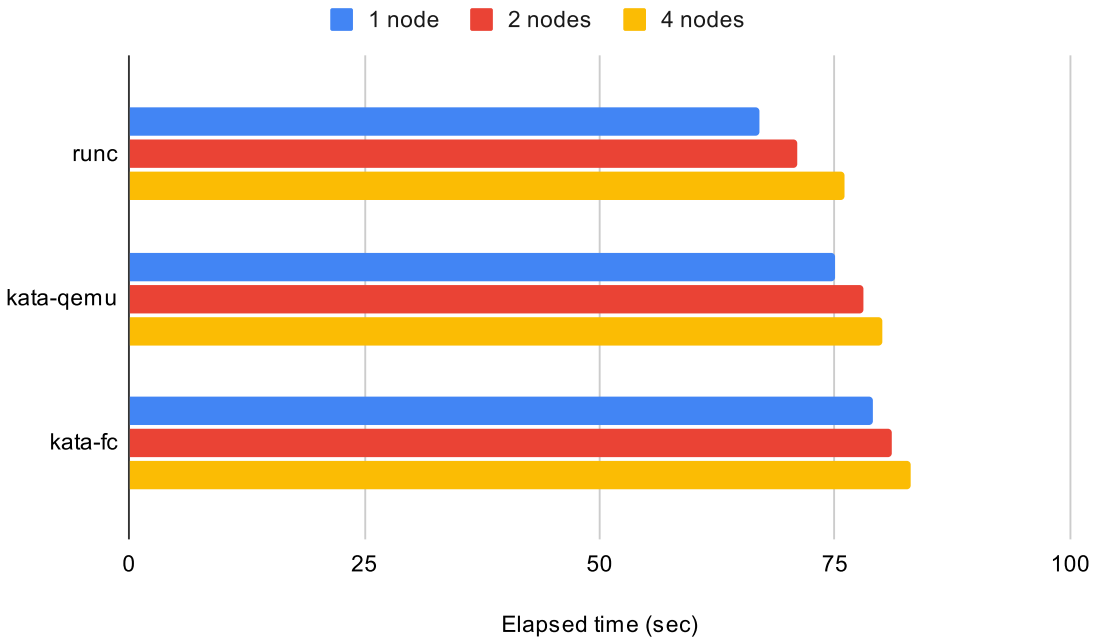

Figure 5.9: Weak Scalability of CPU

6

Related Work

This section is dedicated to examining surveys and publications that concentrate on comparing various container runtimes. None are comparing the different isolation solutions in the context of a container orchestration system but in an isolated environment. Their analysis remains useful to determine whether the relative performance relations translate to the context of Kubernetes.

An Updated Performance Comparison of Virtual Machines and Linux Containers (14). Here, virtual machines are compared with Linux containers in isolated scenarios. It is concluded that the unavoidable virtual machine hypervisor overheads cannot be removed by a subsequent higher layer, but those overheads are mainly affecting I/O performance. The additional layers of abstraction in virtualization lead to diminished workload performance, resulting in customers experiencing a less favourable price-to-performance ratio. Remarkably, this paper questions the common practice of deploying containers inside virtual machines, which is employed by the runtime we worked with extensively in our project, Kata Containers. This practice is questioned as "it imposes the performance overheads of virtual machines".

Hypervisors vs. Lightweight Virtualization: A Performance Comparison (15). In this study, a significant observation is highlighted: Container-based solutions, with their shared kernel and operating system libraries, offer the benefit of a higher density of instances and smaller disk images compared to hypervisor-based approaches. It is concluded that for certain application types, disk I/O efficiency may continue to be a limiting factor. Based on that, we can focus on disk I/O performance in future work. Moreover, the conclusion is drawn that containers exhibit strong performance, though their flexibility and simplified management come at the cost of security. In our work, we verified the higher density capabilities of runc containers.

Performance Overhead Comparison between Hypervisor and Container Based Virtualization (16). In this work, the following conclusions are drawn. First, compared to VMs, containers are more resource and time-efficient, as they bypass the need for running a hypervisor and guest OS, and eliminate the booting and shutdown of a full OS. Second, while container-based solutions are certainly more lightweight, hypervisor-based technology does not always result in higher performance overhead. Last, the container's average performance is generally better than the VM's and is even comparable to that of the physical machine with regard to many features.

Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments (21). In comparison to the rest of the related work, this work focuses on High Performance Computing, which combines computational resources to solve advanced problems. A noteworthy contribution of this study is the presentation of a comprehensive isolation comparison of various systems. However, the runtimes explored are somewhat outdated.

My VM is Lighter (and Safer) than your Container (23). This paper asserts that security remains an ongoing challenge in container environments. As a better alternative, it recommends Unikernels as a faster and safer alternative. Applicable to a cloud environment is the recognition that compute services for various tenants require robust isolation to minimize the risk of sensitive information leakage.

7

Conclusion

In this work, we compared the performance of Kubernetes-Deployed Containers. To do that we designed, deployed, and evaluated experiments with various configurations and goals. Those experiments were run on a virtual Kubernetes cluster, with the help of the Continuum framework. The Continuum framework automates the deployment of infrastructure and benchmarks. It was necessary to extend the Continuum Framework so that it supports different runtimes. We provide the implementation of Kata Containers a runtime that encapsulates containers in virtual machines that run standard OCI Image Format images. Kata Containers use QEMU as the encapsulating virtual machine by default. In addition to that, we enabled the operation of Firecracker-based Kata containers in Continuum.

Through the process of evaluating the startup performance of different containers, we came across an important finding. The startup performance of Firecracker-based Kata containers was worse than the QEMU-based ones. That is unexpected since Firecracker is a serverless-optimised virtual machine monitor where startup time is of high importance. Moreover, it has been shown to outperform QEMU in this regard in various works. We concluded that the reason for this result is Firecracker's limitation of only supporting block-based storage drivers. QEMU-based Kata containers make use of the overlayfs filesystem, which in our tests outperforms Firecracker's devmapper framework.

Our experimental results suggest that the default runtime of Kubernetes, runc, remains the preferable option when performance is a high priority. In the case where security is a high priority, QEMU-based (default) Kata containers present a viable alternative.

7.1 Answering Research Questions

In this section, we present our answers to the research questions proposed in the Introduction section. As mentioned in the section of Evaluation 5.2, we failed to answer **RQ2**, due to unexpected limitations of the Kubernetes system.

RQ1: How does the startup time of different containerised platforms compare in Kubernetes?

The values in Figure 7.1 show the overhead of QEMU and Firecracker-based Kata containers. This overhead is observed when deploying a total of 100 pods across varying configurations of 1, 2, and 4 worker nodes, and is expressed as a percentage. The observed increase is notably significant, particularly in the context of Firecracker-based Kata Containers.

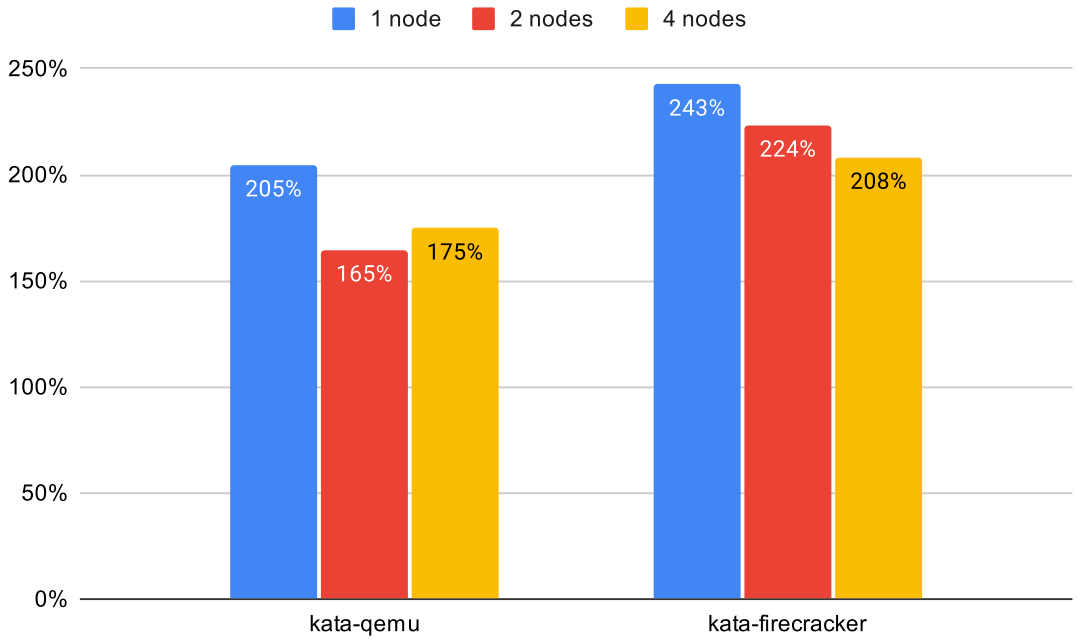

Figure 7.1: Startup time overhead as a percentage from baseline (runc)

	average memory footprint (MiB)	as a percentage of baseline
runc	10	
kata-qemu	152	1520 %
kata-firecracker	135	1350 %

Table 7.1: Average pod memory footprint

RQ2: How does the scalability of different containerised platforms compare in Kubernetes??

We can answer this question indirectly, based on the values of **RQ3**.

RQ3: How does the resource usage of different containerised platforms compare in Kubernetes?
Resource usage was quantified from two distinct aspects: memory consumption and CPU overhead. Table 7.1 provides a comparative overview of the average memory footprint for the various runtimes evaluated. Additionally, it details the percentage increase from the baseline for each runtime. Subsequently, we examine the CPU overhead associated with QEMU and Firecracker-based Kata Containers relative to runc, which serves as the baseline. Figure 7.2 illustrates the results of a 60-second stress test conducted on configurations of 1, 2, and 4 worker nodes.

7.2 Limitations and Future Work

The evaluation of this work was done in an emulated environment, on a single machine, with the help of the Continuum Framework. Emulating a Kubernetes cluster on a single machine can cause performance to differ from real-world scenarios.

In future work, it would be beneficial to incorporate additional runtime environments into these comparative analyses. Candidates include Cloud Hypervisor-based Kata containers and gVisor.

Additionally, based on work from Section 6 - Related Work, we can investigate the I/O performance of Kubernetes deployed containers in future work.

Moreover, it would be interesting to evaluate the performance of Kubernetes containers on edge computing scenarios.

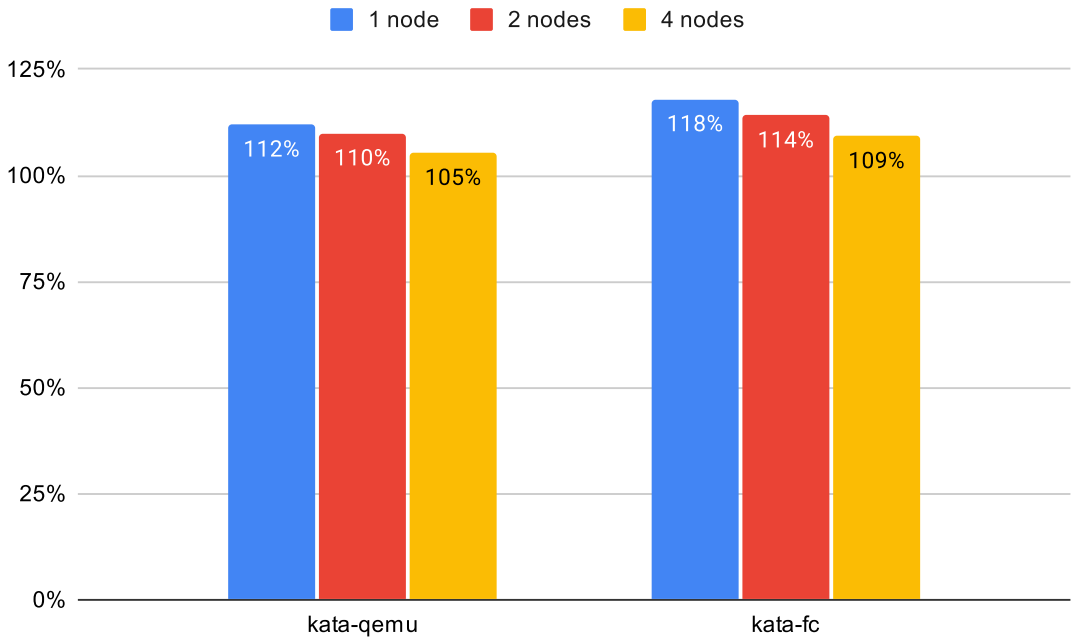

Figure 7.2: CPU overhead as a percentage from baseline (runc)

References

[1] Peter Mell, Tim Grance, et al. **The NIST definition of cloud computing**. 2011. 1

[2] Inc. Gartner. **Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $600 Billion in 2023**, 2022. 1

[3] Google Cloud Band Pulse Survey. **Wave 5**, 2022. 1, 3

[4] Cloud Native Computing Foundation. **With Kubernetes, the U.S. Department of Defense is enabling DevSecOps on F-16s and battleships**. `https://www.cncf.io/case-studies/dod/`. 1

[5] Animesh Trivedi Antonios Sklavos, Matthijs Jansen. **Exploring the Performance-Isolation Trade-off for Isolation Mechanisms**. Technical report, 2023. 1, 4, 7, 8

[6] Ilias Mavridis and Helen Karatza. **Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing**. *Future Generation Computer Systems*, **94**:674–696, 2019. 1, 2, 9, 10

[7] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. **Borg, omega, and kubernetes**. *Communications of the ACM*, **59**(5):50–57, 2016. 2, 11

[8] Cloud Native Computing Foundation. **CNCF Annual Survey 2022**, 2022. 2, 11

[9] Red Hat. **Enterprise Open Source Report**, 2021. 2

[10] Matthijs Jansen, Linus Wagner, Animesh Trivedi, and Alexandru Iosup. **Continuum: Automate Infrastructure Deployment and Benchmarking in**

the Compute Continuum. In *Proceedings of the First FastContinuum Workshop, in conjuncrtion with ICPE, Coimbra, Portugal, April, 2023*, 2023. 2, 12

[11] Michael Pearce, Sherali Zeadally, and Ray Hunt. **Virtualization: Issues, security threats, and solutions**. *ACM Computing Surveys (CSUR)*, **45**(2):1–39, 2013. 2, 8, 10

[12] Tal Garfinkel, Mendel Rosenblum, et al. **A virtual machine introspection based architecture for intrusion detection.** In *Ndss*, **3**, pages 191–206. San Diega, CA, 2003. 2, 8

[13] Tyler Caraza-Harter and Michael M Swift. **Blending containers and virtual machines: a study of firecracker and gVisor**. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 101–113, 2020. 2, 10, 28

[14] 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. 3, 37

[15] Roberto Morabito, Jimmy Kjällman, and Miika Komu. **Hypervisors vs. lightweight virtualization: a performance comparison**. In *2015 IEEE International Conference on cloud engineering*, pages 386–393. IEEE, 2015. 3, 9, 10, 37

[16] 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. 3, 8, 10, 38

[17] Federico Sierra-Arriaga, Rodrigo Branco, and Ben Lee. **Security issues and challenges for virtualization technologies**. *ACM Computing Surveys (CSUR)*, **53**(2):1–37, 2020. 7

[18] James E Smith and Ravi Nair. **The architecture of virtual machines**. *Computer*, **38**(5):32–38, 2005. 8

[19] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. **Container security: Issues, challenges, and the road ahead**. *IEEE access*, **7**:52976–52996, 2019. 8, 9

[20] Antony Martin, Simone Raponi, Théo Combe, and Roberto Di Pietro. **Docker ecosystem–vulnerability analysis**. *Computer Communications*, **122**:30–43, 2018. 9

[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. 9, 38

[22] Aditya Bhardwaj and C Rama Krishna. **Virtualization in cloud computing: Moving from hypervisor to containerization—a survey**. *Arabian Journal for Science and Engineering*, **46**(9):8585–8601, 2021. 9

[23] 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. 10, 38

[24] Vincent van Rijn and Jan S Rellermeyer. **A fresh look at the architecture and performance of contemporary isolation platforms**. In *Proceedings of the 22nd International Middleware Conference*, pages 323–335, 2021. 10

[25] Alessandro Randazzo and Ilenia Tinnirello. **Kata containers: An emerging architecture for enabling mec services in fast and secure way**. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019. 10

[26] Olivier Flauzac, Fabien Mauhourat, and Florent Nolot. **A review of native container security for running applications**. *Procedia Computer Science*, **175**:157–164, 2020. 10

[27] Intel. **Kata containers**. `https://www.intel.com/content/www/us/en/developer/articles/technical/kata-containers.html`. 10

[28] Xingyu Wang, Junzhao Du, and Hui Liu. **Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes**. *Cluster Computing*, **25**(2):1497–1513, 2022. 10

[29] Angel M Beltre, Pankaj Saha, Madhusudhan Govindaraju, Andrew Younge, and Ryan E Grant. **Enabling HPC workloads on cloud infrastructure using Kubernetes container orchestration mechanisms**. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, pages 11–20. IEEE, 2019. 11

[30] Md Shazibul Islam Shamim, Farzana Ahamed Bhuiyan, and Akond Rahman. **Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices**. *2020 IEEE Secure Development (SecDev)*, pages 58–64, 2020. 11

[31] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. **Firecracker: Lightweight Virtualization for Serverless Applications.** In *NSDI*, **20**, pages 419–434, 2020. 28

[32] Kubernetes. **Best practices**. `https://kubernetes.io/docs/setup/best-practices/cluster-large`. 33

Appendix A

Reproducibility

A.1 Abstract

This project uses a modified version of the Continuum Framework to benchmark alternative Kubernetes runtimes. The code is publicly available. It has been tested on a machine with a 20-core Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz. OS was Ubuntu 20.04.5 LTS.

A.2 Artifact check-list (meta-information)

- **Program:** Continuum Framework
- **Run-time environment:** Google Cloud, bare-metal.
- **How much disk space required (approximately)?:** 10 GiB.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour.
- **How much time is needed to complete experiments (approximately)?:** 2-3 hours per experiment.
- **Publicly available?:** Yes.

A.3 Description

A.3.1 How to access

The code is available on GitHub: `https://github.com/anskl/continuum`.

A.3.2 Software dependencies

- QEMU 6.1.0
- Libvirt 6.0.0

- Docker 20.10.12
- Python 3.8.10
- Ansible 2.13.2

A.4 Installation

Download the code from GitHub, and run a `.cfg` file from the `configuration` folder:

```
python3 continuum.py <configuration.cfg>
```

See the configuration template at `https://github.com/anskl/continuum/blob/main/configuration/template.cfg` for inspiration.

A.5 Experiment workflow

All tests from the evaluation are in this folder: `https://github.com/anskl/continuum/tree/antonis/configuration/antonis_thesis`. Make sure to alter `base_path` variable - remove it altogether if the home folder has enough space available (10GiB).

In the home folder of the project, there are bash scripts that run all tests per category, one after another. E.g. `https://github.com/anskl/continuum/blob/antonis/run_all_antonis_1_startup.sh`.

A.6 Evaluation and expected results

Numbers similar to evaluation section

A.7 Experiment customization

All experiments can be modified. For this work, `https://github.com/anskl/continuum/tree/antonis/configuration/antonis_thesis` was used.

A.8 Notes

I am happy to help with any issues related to the project. Feel free to contact me at `antonis@sklavos.io`.

Appendix B

Self Reflection

I wish I spent time on the design earlier on, and not after having almost finished with an original version of the implementation. After designing the system in a more formal manner, I saw many shortcomings and errors in judgment on my part.

I realised soon that I could not estimate the time cost of different parts of the projects accurately. As a representative example, I was making absolutely no progress for 3 days trying to run Kata containers inside a QEMU VM. The solution was a simple XML line in the KVM creation file, allowing access to all CPU features of the host:

```
<cpu mode='host-passthrough'/>
```

Another example is how time-consuming it was to automate the installation of the Kata containers and dependencies with Ansible after having done it manually. Thankfully, in the end, I understood and got familiar with Ansible. Moreover, when I thought I was about 90% done with the project, there was still around 50% left somehow.

I am very grateful for doing this project, not only for sharpening my Linux toolkit but also for working with and understanding Kubernetes in depth. I consider the latter a big milestone and I expect it to help me in my future career.