future internet

Article

A Comparative Analysis of High Availability for Linux Container Infrastructures

Marek Šimon *, Ladislav Huraj and Nicolas Búčik

Institute of Computer Technologies and Informatics, University of SS. Cyril and Methodius, 917 01 Trnava, Slovakia; ladislav.huraj@ucm.sk (L.H.); bucik1@ucm.sk (N.B.)
* Correspondence: marek.simon@ucm.sk

Abstract: In the current era of prevailing information technology, the requirement for high availability and reliability of various types of services is critical. This paper focusses on the comparison and analysis of different high-availability solutions for Linux container environments. The objective was to identify the strengths and weaknesses of each solution and to determine the optimal container approach for common use cases. Through a series of structured experiments, basic performance metrics were collected, including average service recovery time, average transfer rate, and total number of failed calls. The container platforms tested included Docker, Kubernetes, and Proxmox. On the basis of a comprehensive evaluation, it can be concluded that Docker with Docker Swarm is generally the most effective high-availability solution for commonly used Linux containers. Nevertheless, there are specific scenarios in which Proxmox stands out, for example, when fast data transfer is a priority or when load balancing is not a critical requirement.

Keywords: high availability; containers; Linux; Kubernetes; Proxmox; Docker; Docker Swarm

check for updates

Citation: Šimon, M.; Huraj, L.; Búčik, N. A Comparative Analysis of High Availability for Linux Container Infrastructures. *Future Internet* **2023**, *15*, 253. https://doi.org/10.3390/fi15080253

Academic Editors: Jerry Chou and Wu-Chun Chung

Received: 8 July 2023
Revised: 25 July 2023
Accepted: 27 July 2023
Published: 28 July 2023

1. Introduction

Today, containers are highly popular in cloud technology and also in other IT industries, which have become one of the most sought after technologies. Moving to a container platform within cloud computing is becoming a trend not only for companies but also for governments. With the ability to package and run applications and related components in an isolated environment called a container, application deployment can be accomplished quickly and easily.

Various leading cloud service providers have invested in container technologies. These include Amazon Web Services (AWS), which provide Elastic Compute Cloud (EC2) for compute instances, and, since 2017, has been using Elastic Container Service (ECS) to create Kubernetes (EKS) clusters. Google Cloud Platform (GCP) uses Google Compute Engine (GCE) to create and run large numbers of virtual machines and leverages Google Cloud Storage to store big data objects with an increased focus on consistency, durability, high availability, and scalability. GCP also offers high-performance networking capabilities with automatic load balancing solutions. Another major player is Microsoft Azure, which provides Hyper-V containers, with each container running inside a dedicated virtual machine. Red Hat's OpenShift is a recognized leader and active creator of Kubernetes, and offers additional tools for automated deployment, building, and updating of container infrastructure. It also provides monitoring and security features to manage clusters and allocated resources. These major platforms in the cloud services space are guarantors of the future use and development of container technologies on the Internet [1–3].

There are many high-availability solutions within Linux containers, but it is often not entirely obvious which one offers better performance characteristics. Given the constant growth of active users of various services or microservices, it is important that these services are always active and accessible to their users. For this reason, the environments for providing and achieving high availability are constantly expanding and evolving.

High availability ensures that the service remains accessible even when there is a hardware failure or a software upgrade; high availability can be described as a combination of concepts to determine whether an information system is available and has optimal performance, thus it does not have frequent outages and is available most of the time. The goal of high availability is to find and eliminate possible single points of failure, i.e., such points that when these points fail, the service stops working or works only in a limited way. Properly configured high availability can also deal with possible software and hardware failures. Systems and services that use this concept should be fully automated, that is, no human intervention is required for the high-availability concept to work properly, and they should be able to independently manage performance, anticipate some types of failures, and intervene in the event of a catastrophic situation [4,5]. The issue of security of container systems and protection against possible threats of various types, ranging from unauthorized access to DDoS attacks [6–8], additionally, contributes significantly to achieving high availability.

In the case of high availability, several metrics can be considered [9]:

- Mean time between failures—how long the environment works between system failures,
- Mean downtime—how long a system is unavailable before it is restored or replaced by another in the topology,
- Recovery time objective—the total time it takes to recover a given target until it is available again,
- Recovery point objective—the period of time during which data needs to be recovered. For example, if data are backed up every 24 h, the total amount of data that can be lost is in a 24 h period.

In this context, it is very important to evaluate the performance characteristics of different high-availability solutions within Linux containers to determine which offers optimal results. As technology continues to advance and user demands increase, evaluating the performance of high-availability solutions within Linux containers becomes even more important. Achieving continuous availability is no longer a luxury, but a necessity in today's dynamic IT environment. Organizations must carefully evaluate and compare the performance characteristics of different high-availability solutions to determine the most appropriate option. The growing user base of various services and microservices requires a production environment that is always active and available. In addition, high availability must address hardware and software failures, minimize single points of failure, and include automated systems capable of managing performance and intervening during critical situations.

This paper explores several containerization platform solutions such as Docker using Docker Swarm, which uses Containerd technology, Kubernetes uses CRI-O container interface, and Proxmox, which uses LXC technology. The focus is on the essential characteristics of the systems, such as the recovery time of service availability, the rate of data to the user, the number of failed calls, and the average call time. The paper proposes test scenarios based on which testing of these services is carried out, as well as an analysis of the test results.

1.1. Contribution

The originality of the paper lies in the comparative analysis and performance evaluation of high-availability solutions in Linux container infrastructures. While other studies address various aspects of container technologies such as containerization, orchestration, security, and virtualization, this paper specifically focuses on evaluating the performance characteristics of different high-availability solutions. It fills a gap in the existing literature by providing detailed information on service recovery time, failed calls, data transfer rate, and average call time for popular container platforms Docker, Kubernetes, and Proxmox through experiments and analysis of results. This new contribution enables readers to make informed decisions when selecting the most appropriate high-availability solution for their specific use cases.

1.2. Organization

The article is organized as follows: Section 2 discusses the literature closely related to the research of the paper. Section 3 gives a brief overview of the container technologies under investigation. Section 3 describes the experimental test environment and the proposed test scenarios. Section 4 presents the results of the experiments conducted, their analysis, and discussion. Finally, a brief summary of the results and future research directions in this area is presented in Section 5.

2. Related Work

There are diverse studies and research on the issue of analyzing the performance of virtualization and containerization technologies from different perspectives.

For example, a comparison between virtualization and containers is addressed in a study by Moravcik et al. [10]. In addition to virtualization, it also compares two leading container platforms, Linux containers (LXC) and Docker, and highlights their advantages and disadvantages. Linux containers are more OS-centric and present an efficient method of creating and managing multiple OSs on a single device; Docker enables containerization through application containers, making it a suitable choice for systems hosting numerous applications due to its robust orchestration support. The study states that from a networking perspective, Linux containers and Docker are equivalent, with each allowing different network configurations. In an experiment, the authors found that LXC achieves lower latency than Docker, which could be important for choosing a more suitable container platform based on specific needs and requirements.

The results of the experimental analysis in the study by Kaur et al. [11] focus on comparing the performance of containerization and virtual machines in data centers. Although cloud computing may be slightly slower compared to on-premises environments, its benefits are unquestionable. Researchers focus on the performance and security of applications in containerized and virtual environments, testing parameters such as CPU performance, file I/O, and memory utilization. In addition, they examine the behavior of applications loaded with HTTP data from different sites to compare performance on both these types of environments. Overall, the findings confirm that the Linux-based KVM virtualization hypervisor provides a better performance than Docker containers in these tested areas. Virtualization and containerization have become key tools within datacenters, contributing to more efficient resource utilization and bringing many other benefits. With the growing trend of microservices and DevOps, containerization is starting to gain popularity, and it is important to understand and compare the performance of these technologies.

Putri et al. in [12] focused on evaluating the impact of different container platforms (Docker, LXC, and LXD) when running different TCP services. They analyzed the system performance of each container and compared it with a native system without container solutions. The purpose of the research was to provide comprehensive performance metrics to compare container platforms. The results indicate that the performance between containers can vary. The authors found that LXD performs better in terms of system performance. They also observed that server performance results vary depending on what specific service was being evaluated.

Li et al. in [13] addressed a comprehensive comparison of different convergence architectures. They investigated the characteristics of container solutions such as Docker, QEMU/KVM, Firecracker, gVisor, and Kata to identify limitations and the most suitable use cases. As part of the measurements, they analyzed the performance of CPU, RAM, system calls, network, disk storage, and startup time. They also evaluated their performance when running the Nginx web server and the MySQL database management system. On the basis of the benchmark tests, they analyzed the results, discussed the advantages and disadvantages of each architecture, and identified potential limitations. The results of the analysis showed various trade-offs and performance barriers to virtualization, providing insights for making decisions about the use of each technology in specific cases. They also outlined a direction for future research, which should focus on optimizing the memory footprint of lightweight hypervisor containers and on leveraging other technologies such as Xen with customized unikernels and the Kubernetes KubeVirt superstructure.

Choosing the right container orchestration tool is a complex task for organizations that manage a large number of containers. Each tool has its advantages and limitations. Authors Malviya and Dwivedi in [14] performed a comparative analysis of four commonly used orchestration platforms: Redhat OpenShift, Mesos, Docker Swarm, and Kubernetes. They evaluated each tool based on parameters such as security, deployment, cluster installation, scalability, stability, and learning curve. They found that Kubernetes has the best scheduling features and Docker Swarm is easy to use. Mesos, on the other hand, is suitable for scalable deployments, and OpenShift is an excellent tool from a security perspective. Kubernetes is the most popular tool among the platforms mentioned above. It is important for organizations to choose tools based on their specific needs and the complexity of the application.

Although many other performance comparisons of containerization environments and orchestration methods have been performed, e.g., [15–17], still the high availability of applications hosted on container platforms remains largely unexplored. This study seeks to fill this gap and show which of the containerization technologies under investigation can be used to achieve the best level of high availability.

3. Materials and Methods

Linux containers provide isolated environments that enable the execution of applications and processes independently of the underlying host operating system. Containerization is a form of operating system virtualization that allows applications to run in isolated environments without the need to fully virtualize the entire operating system. By encapsulating all the necessary dependencies and application files, containers offer application portability, compatibility, and independence from specific operating systems. Containers, including those deployed in cloud environments, are characterized by their compact size and enhanced portability. This means that applications developed for an operating system can be seamlessly executed on different operating systems without requiring modifications to the application code. In the context of achieving high availability in containerized solutions, operating system-level virtualization is used, allowing multiple Linux systems to run concurrently. Unlike technologies that rely on a separate virtualization hypervisor, containers operate without the need for an additional hypervisor [18–20].

Container virtualization works by not emulating hardware layers, but creating a virtual environment with its own processor and memory using native Linux elements. These elements include the Linux kernel and features such as cgroups (control groups), namespace isolation, and SELinux (Security-Enhanced Linux), which are used to isolate applications and processes in containers. Cgroups control container access to resources such as CPU time, memory, and network resources. Namespace isolation allows to separate the container environment from the host system, which means that processes in the container can only access resources in their own isolated environment. SELinux provides an additional level of security by allowing one to define rules for accessing resources in containers. These features make containers easy and efficient to deploy applications. It is generally recommended that a single container manages only one application or part of an application, which reduces system overhead and improves the isolation of applications from each other [21,22].

Next, the article describes three popular container platforms, Docker, Kubernetes, and Proxmox. First, three specific examples are given for each of these platforms in which they are commonly used.

Docker: (i) Microservices architecture deployment—Docker is commonly used in microservices-based architecture deployments where applications are divided into smaller, independent services. Each service can be encapsulated in a Docker container, allowing easy scaling, isolation, and management of individual components. This enables organizations to build and deploy highly scalable and modular applications. (ii) Continuous Integration and Deployment (CI/CD)—Docker plays a key role in CI/CD workflows, where software development teams automate the process of building, testing, and deploying applications. By wrapping an application and its dependencies in Docker containers, developers can ensure consistent and reproducible builds across environments. This facilitates the seamless integration of code changes, automated testing, and efficient deployment, ultimately speeding up the software delivery process. (iii) Development Environments—Docker is widely used by developers to create portable and self-contained development environments. With Docker, developers can define the exact software package, libraries, and dependencies needed for their applications within a container. This ensures consistency between different development engines, solving the problem of inconsistencies in compatibility between different environments. Docker also allows developers to easily share their development environments, enabling collaboration and streamlining development processes.

Kubernetes: (i) Cloud-native applications—Kubernetes is often used as a foundation for building cloud-native applications. It provides the infrastructure and capabilities necessary to develop and run containerized applications in cloud environments, thus promoting portability, scalability, and resilience. (ii) Big Data and Analytics—Kubernetes is used in big data and analytics environments to organize and manage distributed data processing frameworks such as Apache Spark and Hadoop. It enables the deployment of containerized data processing tasks and simplifies the management of complex data feeds. (iii) Machine learning and AI—Kubernetes is used in machine learning and AI workflows to manage the deployment and scaling of training and inference workloads. It enables organizations to efficiently run containerized machine learning models, perform distributed training, and manage resource-intensive AI workloads.

Proxmox: (i) development and test environments—Proxmox can be used in development and test scenarios to create isolated virtual environments for software development, debugging, and testing. It allows developers to easily replicate different settings and configurations, enabling efficient software development and testing workflows. (ii) Web hosting and service providers—Proxmox is used by web hosting companies and service providers to offer hosting and cloud-based virtual private server (VPS) services to their customers. It provides the infrastructure and management capabilities necessary to host multiple virtual environments and provide scalable and reliable services to clients. (iii) Media streaming and content delivery—Proxmox can be used in media streaming and content delivery applications. It enables the creation of virtual environments for media servers, content caching, and load balancing, facilitating the efficient distribution of multimedia content over networks.

3.1. LXC and Proxmox as a Container Orchestrator

Tools used in container virtualization include LXC. This is an operating system-level virtualization that allows multiple isolated containers to run and virtualize simultaneously. LXC uses the features of the Linux system to limit and prioritize system resources without the need to start a virtual machine. Namespaces are used to isolate the access of a given container within the operating system, allowing the ownership of the container files to be separated from the ownership of the operating system files themselves.

LXC is able to create a virtual environment that isolates applications from other running applications on the host system and ensures that the applications do not share resources. In this way, applications can run in the container independently of other applications and system resources. LXC offers fast creation and the launch of virtual environments, simplifies application development and testing, and reduces server management costs. In addition, it provides the ability to create layers for containers, allowing multiple containers to be created that share the same base but differ in specific applications or dependencies. This leads to savings in disk space and allows easier and faster updates [23–25].

Proxmox or Proxmox Virtual Environment (Proxmox VE) is a converged open source environment used in container virtualization. Its web-based environment enables the management and orchestration of operating systems and virtualized environments. Promox subsupports the virtualization of LXC containers and their use in high availability, Figure 1. To manage and orchestrate containers in Proxmox, a cluster manager is used to enable interconnection and communication between Proxmox instances. The manager provides redundancy and error detection through a method called fencing. Proxmox has several fencing options, such as external power switches, network-level node isolation, and the use of timers to recover failed nodes. If a node cannot be automatically recovered, a manual intervention by an administrator is required [26–28].

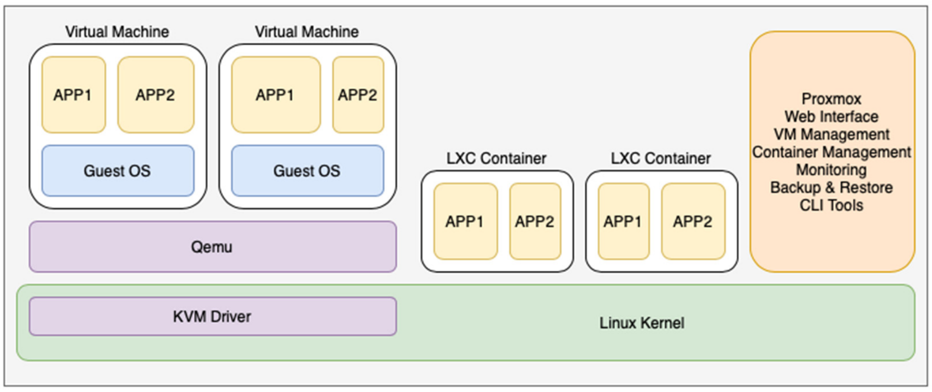


Figure 1. Proxmox Virtual Environment Architecture.

3.2. *Containerd, Docker, and Docker Swarm as a Container Orchestrator*

Containerd is an abstraction layer that separates the OS-specific system calls and functions needed to run and virtualize containers. It provides simplified and abstract functions so that users do not have to work directly with system calls. Containerd does not provide management of network elements and settings, which is the responsibility of the superstructures above it. Containerd is based on an API that allows access and management of containers [29,30].

Docker is a tool and a superstructure on top of Containerd that provides application automation in containers. The Docker engine is the application deployment layer of containers. Docker makes it easy to test and deploy applications and is based on existing container virtualization methods such as Containerd and libvirt. Docker provides isolation of containers from the host system but requires additional steps to secure the installation. Docker consists of a client and server, Docker containers, Docker registries, and Docker images. A container is an isolated environment where applications run on the basis of Docker images that define instructions for the container. Docker registries are used to store finished Docker images [13,31,32], see Figure 2.

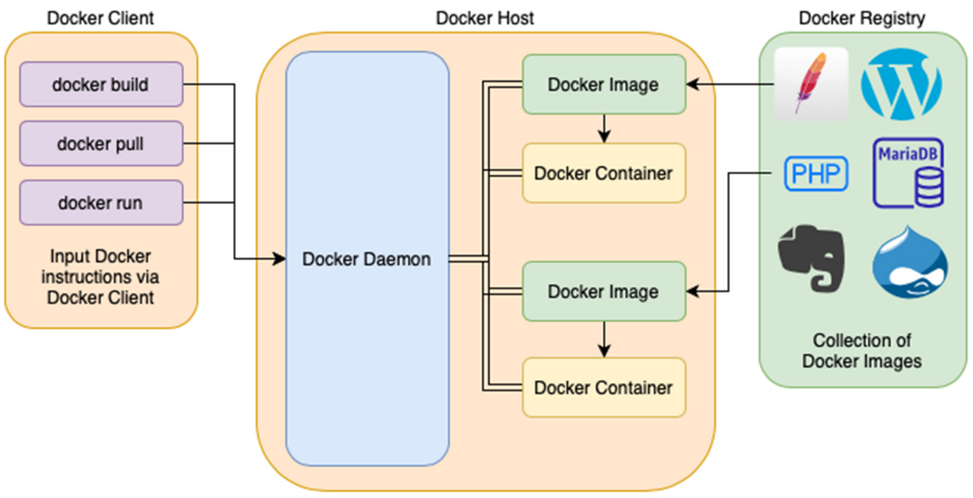


Figure 2. Docker Architecture—Host, Client, and Registry.

Docker Swarm is a tool for distributing containers and services, clustering them, and managing resources. It is often used for high-availability solutions where multiple instances of the same service are run and the user load is balanced. Docker Swarm's main features encompass cluster management, tightly integrated with the Docker engine, decentralized design, declarative service models, scalability, service health monitoring, multihost networking, load balancing, and easy application updates. Docker Compose also offers inter-container networking capabilities and the definition of external data sources [33,34].

Security in Docker is an important topic because Docker is not a fully isolated system like a virtual machine. The applications in the container can run under the main admin user, which poses a risk of attacks. To secure Docker, there are security profiles and namespaces to define access rights and isolate containers from the main operating system [35–37].

3.3. CRI-O and Kubernetes as a Container Orchestrator

CRI-O was developed as a standardized plugin for the Kubernetes environment, to run and monitor the functionality of containers. It serves as a link and layer between containers and the Kubernetes CRI (Container Runtime Interface) and is based on an older version of the Docker architecture. Compared to Docker, it is less demanding on system resources; its primary benefit lies in the reduction of resource consumption. CRI-O can directly run containers created by Kubernetes without the need for additional code or tools. When Kubernetes needs to launch a container, it simply calls a command on CRI-O, and a specially crafted daemon for CRI-O works with any Open Container Initiative (OCI) compliant program to launch and create that container. If Kubernetes needs to interact with CRI-O, a request is made to its Container Request Interface (CRI), which in turn makes a request to the CRI-O service. This service executes instructions based on the specification and provides Kubernetes with information about the new container, such as its ID and status [38–40].

Kubernetes is an open-source container orchestration platform that enables automation of the deployment, extension, and management of container applets. Using Kubernetes, developers can improve the responsiveness and stability of an application by distributing container clusters to service zones. However, one of its limitations is the lack of decoupling for single individual users, requiring the creation of dedicated clusters for each of them. Kubernetes uses service replication across multiple nodes in a cluster and maintains information about the states of these containers, see Figure 3. In a high-availability environment, Kubernetes uses methods such as automatic replacement of failed nodes, self-healing, and load balancing. Automatic failed node replacement moves applications to other nodes if a failure occurs. Self-healing attempts to restart failed containers automatically. The load balance ensures a fair load distribution among the nodes in the cluster [39,41].

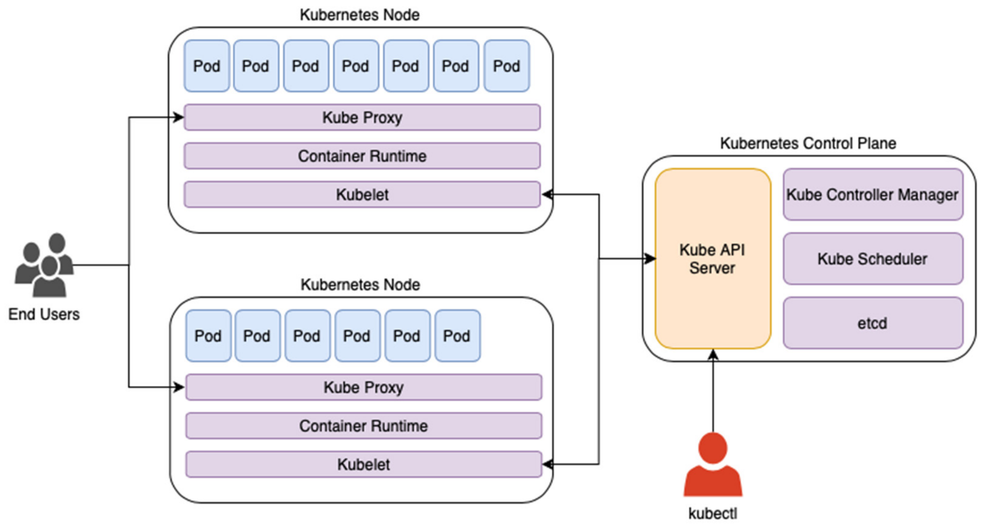


Figure 3. Kubernetes Architecture.

3.4. High Availability

Reliability and high availability are important factors in Linux containers and container orchestration. High availability means the ability of the system to continue operating despite node failures in the cluster or hardware and software updates. Fault tolerance is a key technique to achieve high availability. Replication management is one way to implement fault tolerance. In this way, the required number of containers can be maintained, and continuous operation can be ensured. Health checking is used to identify faulty containers and then remove them, deploying additional containers to ensure that the desired number of replicas is maintained. High availability is essential for Linux containers because it provides continuous operation of applications even if parts of the system fail. Replica management and high-availability controllers are tools that allow one to achieve high availability and guarantee stable operation of containerized applications. High-availability controllers provide the ability to configure multiple orchestration administrators to ensure continuous control of the application even if an orchestrator node fails or becomes overloaded. Their implementation is important in the design and implementation of container systems to minimize failures and ensure reliable operation [42–44].

4. Experiment Test Environment

The operating system used for testing was Debian 11, codenamed "Bullseye", on which all solutions were tested. To make the results comparable, three systems with the same specifications were used, on which Debian was installed. The Debian system was installed three times on the machines in separate partitions; only one platform was installed during testing.

For comparability of results, the same performance constraint was set for all containers, namely the maximum usage of one CPU core and two gigabytes of RAM. This ensured that the tested containers were always performance equal and thus it was possible to make a comparison between them.

The three computers that were part of the test infrastructure were connected to a single switch on a single router on the same network; the devices had to communicate with each other, so they had to be on the same network. The computer from which the testing was performed had the same specifications as the actual computers in the cluster and was also connected to the same network, see Figure 4. The configuration that the computers used was as follows: an Acer Veriton E430G, 4 GB DDR3 1600 MHz RAM, a Pentium Dual Core G2030 processor, and a 500 GB hard drive with a 7200 RPM spinning hard drive.

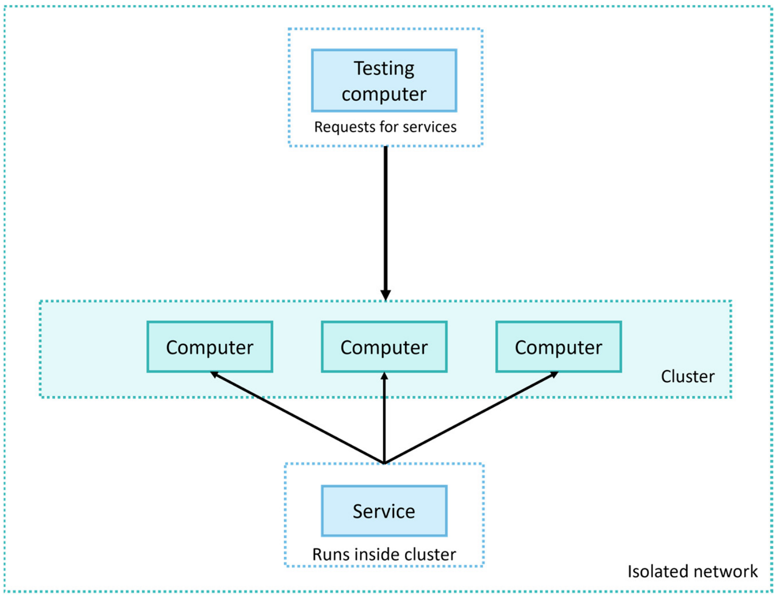


Figure 4. Connection diagram of test computers.

The tests were conducted in an isolated network, which included the testing computer itself, from which the tests were conducted; the network cards of the computers and their switch (all ports) had a transfer rate of 1000 Megabits per second. All computers were directly connected to the switch to reduce the possibility that other factors, such as other network communication, could affect the results. The switch model was a Tenda TEG1024D, 24 gigabit ports. Individual environments were booted independently on the test nodes without being affected by other environments. The testing was performed in April 2023.

The **Proxmox** environment was installed using the official automatic installer, which installed the environment to the disk. After successful installation, a highly available cluster was created. The cluster was then connected to the other two Proxmox instances installed on the other two computers. After the standard installation of Proxmox, the first step was to set up the shared storage on which container replication was performed; this storage required a separate disk on each machine with a Proxmox instance. To use replication, the storage had to be set up as a ZFS file system. Then, it was necessary to create a cluster and connect to the cluster of each Proxmox instance. Then, a container image was downloaded to the first node in the cluster; the node's disk size was twenty gigabytes. The number of processors was limited to one and the operating memory was also limited to two gigabytes; the network settings were chosen by DHCP. Then, the replication was set to achieve high availability. Replication was performed on an instance in the cluster, and the replication time was set to every minute.

The **Docker** environment with Docker Swarm was installed along with the Docker Compose add-on. After Docker was installed, the so-called "Swarm" was created first, which is the cluster to which other Docker instances connect. Once the cluster was created, a token was requested to add the other Docker instances to the cluster. Then, a Dockerbuild file was created with instructions for installing the apache2 service and running it. Lastly, a Docker compose file had to be created to run the instance, which had instructions to create a compiled container from the Dockerfile for execution. As with Proxmox, limits were set, namely: the number of processors were limited to only one and the maximum operating memory usage to 2 GB. Container nodes and one container instance were then started, as well as the increase of the number of instances to three.

When installing the **Kubernetes** environment with CRI-O, the first thing that needed to be installed was Buildah, which is the software for creating containers in the OCI standard. Subsequently, the image was created from the Dockerfile using Buildah. Then the cluster was initialized, other Kubernetes instances were connected to the cluster and worker type nodes were also connected. In the same way, CPU and RAM usage was also limited here.

When designing tests to test performance on applications, the possibility of comparing results with each other in general was taken into account; for this reason, applications that are not specifically designed for any of the test platforms but are general-purpose were selected for testing.

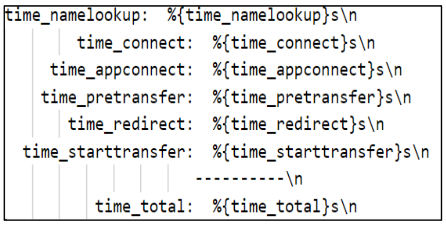
The first application selected was CURL, which can measure the response time since the request was made; this makes it possible to measure the total time it takes to migrate a dropped node to another machine in the cluster.

The second application was the ApacheBench tool, which allows you to perform a simulation of the load on individual nodes; with such a simulation, it is possible to measure and also obtain information from users of the service on how each technology handles the load. ApacheBench is a tool that is often used to benchmark web servers. When performing this test, the ApacheBench tool returned information about the error rate of requests under load, what the response time of the request was from sending it to receiving the response, and also what the average data rate was during the tests.

In testing, the focus was on measurable aspects such as service recovery speed, average transfer rate, and number of failed calls; for Kubernetes and Docker platforms, the parameters of transfer rate and number of failed calls were also tested when native load balancing was enabled, so three service replicas were running simultaneously on three machines. Proxmox was excluded from this load balancing test because it does not support native load balancing, and the experiments were primarily focused on directly supported features by the platforms.

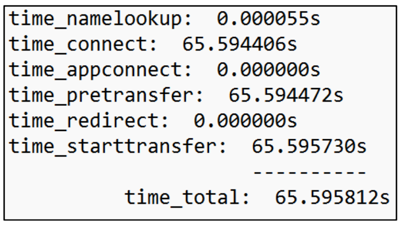
Test Scenarios

Several test scenarios have been proposed in the experiments performed, which are reproducible on the above infrastructure as well as on any other layout. The **first test scenario** was to measure the service restoration rate using the CURL tool. A query was run on the server for a given service; this query was continuously repeated, and while the query was running, the machine running the service was disconnected from the computer network. Thus, the response returned to the query was only from a restored or migrated service from another computer in the cluster. The numeric result returned the time of service restoration. The measurements were repeated ten times, the times were averaged, and an average service recovery time was produced. The testing used a text format that the CURL tool accepts as an input parameter, and on this basis, CURL compiles a response, as shown in Figure 5a.



```
time_namelookup:  %{time_namelookup}s\n
       time_connect:  %{time_connect}s\n
    time_appconnect:  %{time_appconnect}s\n
   time_pretransfer:  %{time_pretransfer}s\n
      time_redirect:  %{time_redirect}s\n
 time_starttransfer:  %{time_starttransfer}s\n
                    ----------\n
         time_total:  %{time_total}s\n
```

(**a**)



```
time_namelookup:  0.000055s
time_connect:  65.594406s
time_appconnect:  0.000000s
time_pretransfer:  65.594472s
time_redirect:  0.000000s
time_starttransfer:  65.595730s
                    ----------
          time_total:  65.595812s
```

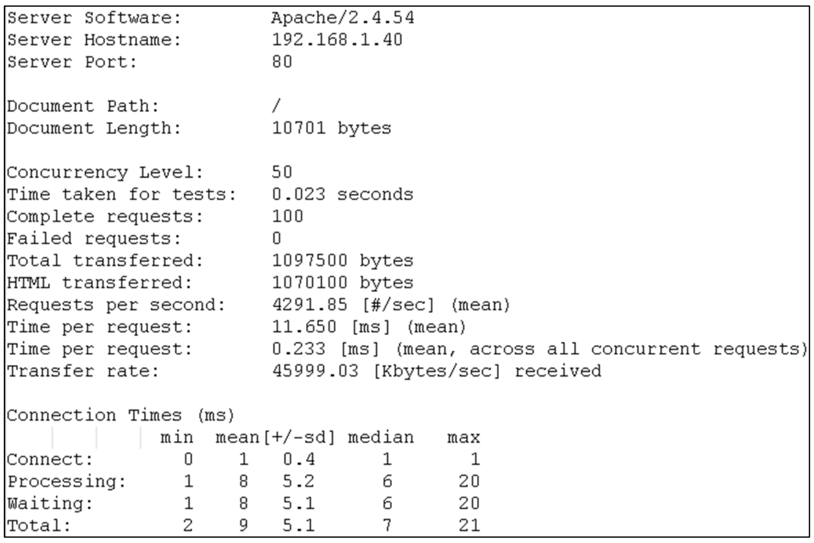
(**b**)

Figure 5. CURL Input parameter for formatting (**a**) text format (**b**) information from the output.

The information that is received from the output is given in seconds and is namely: Time to discover the actual IP address from the DNS server (Time namelookup), Time to establish a TCP connection to the server (Time connect), Time to establish an SSL connection (Time appconnect), Time to start the pretransfer call (Time pretransfer), Time redirect (Time redirect), Time when the server is ready to send the data (Time starttransfer), and Time total call (Time total), as shown in Figure 5b.

In the **second test scenario**, ApacheBench was used to execute queries on a particular instance of the service. Subsequently, after the tests were executed the returned responses were analyzed. The requests were monitored for: the average time per request (Time per request), the number of successful requests (Completed requests) and failed requests (Failed requests), and the transfer rate (Transfer rate), as shown in Figure 6. The number of requests and the number of concurrent calls were increased during the testing to increase the load on the services and to gain insight into the load handled by the systems. Testing was performed using the built-in command "ab -n NUMBER_REQUEST -c NUMBER_ CONCUR-RENT_CALLS IP_ADDRESS<".

The average value provides a concise and informative summary of the typical response time, capturing the overall performance of the system being evaluated and is sufficient for comparison purposes in this case. It provides a robust and concise summary of response time, facilitating informed decision-making about the performance of the system under evaluation.



```
Server Software:        Apache/2.4.54
Server Hostname:        192.168.1.40
Server Port:            80

Document Path:          /
Document Length:        10701 bytes

Concurrency Level:      50
Time taken for tests:   0.023 seconds
Complete requests:      100
Failed requests:        0
Total transferred:      1097500 bytes
HTML transferred:       1070100 bytes
Requests per second:    4291.85 [#/sec] (mean)
Time per request:       11.650 [ms] (mean)
Time per request:       0.233 [ms] (mean, across all concurrent requests)
Transfer rate:          45999.03 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   0.4      1       1
Processing:     1    8   5.2      6      20
Waiting:        1    8   5.1      6      20
Total:          2    9   5.1      7      21
```

Figure 6. Measurement result using ApacheBench.

5. Results

Based on the analysis of the results of the experiments carried out, the results of the service recovery time were obtained in the first scenario; after repeating this test ten times for each of the installed services and configurations, it can be concluded that in general the Proxmox VE platform takes the longest time to recover a node. The likely reason for this slowdown is due to the method that the Proxmox platform uses to recover the service, the fencing method, which attempts to recover the dropped node first and then transfer it to another machine on that node. Overall, the fastest service recovery was in the Docker environment along with its high-availability manager Docker Swarm, see Figure 7. Kubernetes only achieved second place in service recovery speed, likely due to its higher overhead services and its use of the CRI-O intermediate. Using CRI-O as an intermediate layer may introduce additional communication and processing, which may affect the service recovery rate by default.

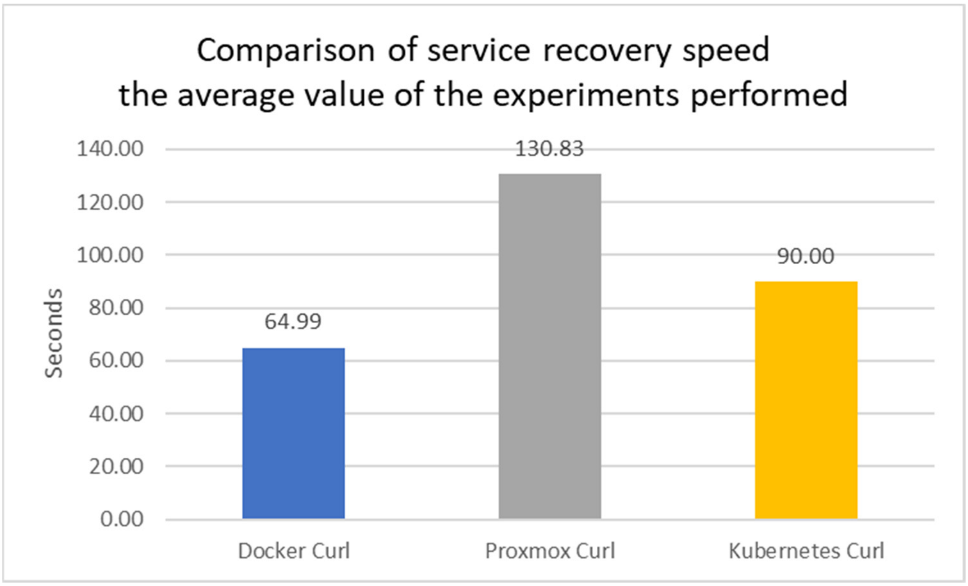


Figure 7. Measuring the service recovery rate.

Based on the analysis of the results from the experiments performed using the second scenario focusing on the total number of failed calls, it can be concluded that the Kubernetes platform using a single node had the highest number of failed calls; the total number of failed calls increased linearly on this service. In contrast, the Docker platform with its high-availability manager, Docker Swarm, had the lowest number of failures, but using up to three nodes. When using only one node, the Docker platform ranked slightly worse than the Proxmox platform; Proxmox had an average overall failure rate in the middle of all platform measurements, see Figure 8.

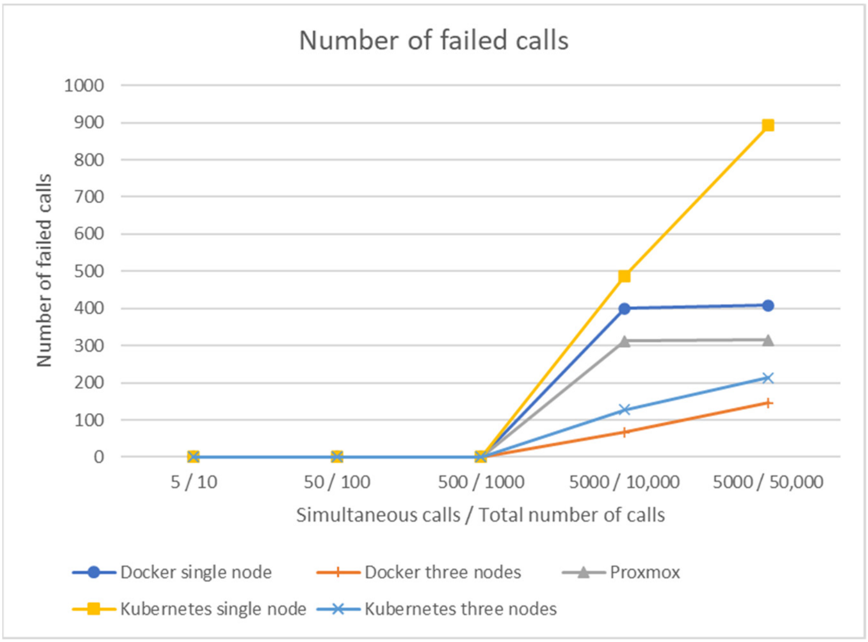


Figure 8. Measurement of the average number of failed calls.

Another analysis based on the second scenario using ApacheBench was the data rate analysis. From the results of the given analysis, it can be stated that the Proxmox platform obtained the highest average data transfer rate, i.e., it was able to transfer the largest amount of data in a certain time. Overall, the Kubernetes platform obtained the worst result using a single node, which had the slowest average data transfer rate. However, the services, regardless of their total number of nodes, were relatively balanced. The results are interpreted in the table, which is divided into columns with an increasing number of concurrent calls and total calls for each platform. The last column represents the mean of the results from all columns in a row. Individual results are shown for data rates in kilobytes per second, see Table 1.

Table 1. Measurement of transfer rate in kilobytes per second.

Simultaneous Calls/Total Number of Calls	5/10	50/100	500/1000	5000/10,000	5000/50,000	Mean
Docker single node	455.57	783.14	975.55	85.62	270.84	514.14
Docker three nodes	476.17	1337.50	694.94	132.23	350.11	598.19
Proxmox	33,724.90	45,999.03	47,790.24	1712.38	8137.98	27,472.90
Kubernetes single node	428.22	634.27	852.74	472.25	128.30	503.16
Kubernetes three nodes	455.29	827.31	610.27	572.13	214.12	535.82

The last analysis of the results from the second scenario experiments using ApacheBench was an analysis of the average call time to a given service. Based on the results of the analysis, it can be concluded that the Proxmox platform had the longest average call time just for the impact call of 5000 concurrent calls and 10,000 total calls. In general, the impact calls of 5000 concurrent calls and 10,000 total calls appeared to be the most challenging calls to process for all services. The presumed reason why the Proxmox platform had the worst average call time result is that it does not support native load balancing, i.e., all calls go directly to a given service and have no load balancing manager. This statement is confirmed by the fact that the platforms that provided such functionality in the experiments had much better results. In summary, the Docker platform using three nodes had the best results and thus was the fastest at answering queries; the Kubernetes platform differed only slightly from the best in its speed when using three nodes but ranked as the second slowest when using only one node, see Figure 9.

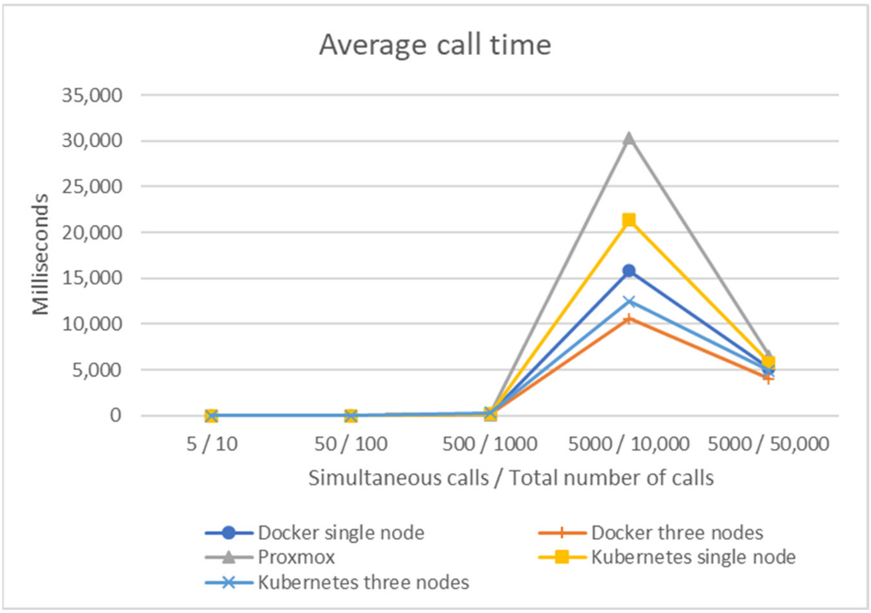


Figure 9. Average call time results.

Based on the analysis of the results obtained from all the experiments, it can be concluded that the most suitable high-availability solution in terms of performance in commonly used Linux containers is Docker using the Docker Swarm high-availability manager. If the most important criterion is the transfer rate, the most suitable platform is Proxmox. However, it should be noted that if one does not plan to use a load balancing service and it is important to have as few total failed calls as possible, then Proxmox is again the most suitable choice, although one has to take into account its results from average call times, which are significantly worse than in the other platforms. While Kubernetes may not have consistently achieved the absolute best results, its competitive performance in key metrics, such as average call time, average failed calls, and average recovery rate, highlights its potential as a robust and reliable high-availability solution in specific scenarios. Summary Table 2 presents the collective results of the individual experiments.

Table 2. Summary results from individual experiments.

	KB/s	**Milliseconds**	**Number**	**Seconds**
Technologies	Average transfer rate	Average call time	Average failed calls	Average recovery rate
Docker	514.14	4231.38	161.60	64.99 [1]
Proxmox	27,472.90 [1]	7405.50 [2]	125.40	130.83 [2]
Kubernetes	503.16 [2]	5486.94	276.00 [2]	90.00
Dockers 3 nodes	598.19	2967.18 [1]	42.60 [1]	-X
Kubernetes 3 nodes	535.82	3541.69	68.20	-

[1] The best result in the category (in the column). [2] The worst result in the category (in the column).

6. Discussion

This study provides valuable information on the performance characteristics of different high-availability solutions for Linux containers. Comparison of different services based on service recovery time, total number of failed calls, data rate, and average call time enables informed decision making when selecting the appropriate high-availability solution for specific use cases. Understanding the strengths and weaknesses of each service can lead to optimal selection, ensuring a reliable and efficient container-based infrastructure.

In addition, the study highlights the importance of taking into account not only performance indicators but also the specific requirements and characteristics of each application or service deployed. Different applications may have different resource, scalability, and fault tolerance requirements. Therefore, it is critical for organizations to carefully assess their individual needs before selecting a container platform.

As mentioned in Section 2, various studies have attempted to compare container platforms from different perspectives. The scope of the present study was specifically focused on high availability within Linux container infrastructures, which narrowed the parameters evaluated; other comparative studies have investigated various aspects of virtualization, containerization, system performance, and container orchestration. The selection of platforms depends on what a company prioritizes when implementing a container environment.

A study by Moravcik et al. [10] compares virtualization and containerization, specifically the Linux containers LXC and Docker. From a network perspective, Linux and Docker containers are considered equivalent, but allow for different network configurations. The study found that LXC achieves lower latency than Docker, which may be important for specific use cases. Kaur et al. in [11] compare the performance of containerization and virtual machines in data centers. They evaluate parameters such as CPU performance, file I/O, memory usage, and application behavior in different environments. The findings show that the Linux-based KVM virtualization hypervisor performs better than Docker containers in the tested domains. The study by Putri et al. [12] evaluates the impact of different container platforms (Docker, LXC, and LXD) when running TCP services. It analyzes the system performance and compares the containers with the native system. The results indicate that performance between containers can vary, and LXD performs better in terms of system performance. The study by Li et al. [13] provides a comprehensive comparison of different convergence architectures (Docker, QEMU/KVM, Firecracker, gVisor, and Kata). It analyzes performance metrics such as CPU, RAM, system calls, network, disk storage, and boot time. The results highlight virtualization trade-offs and performance barriers and provide insights for case-specific decision making.

In [14], the authors Malviya and Dwivedi focused on comparing container orchestration platforms (Redhat OpenShift, Mesos, Docker Swarm, and Kubernetes). Evaluation considers parameters such as security, deployment, cluster installation, scalability, stability, and learning curve. The findings suggest that Kubernetes has the best scheduling features, Docker Swarm is easy to use, Mesos is suitable for scalable deployments, and OpenShift excels in security. Kubernetes is identified as the most popular tool among the platforms discussed.

Each of the above studies provided valuable insight and recommendations based on their specific areas of focus and benchmarks. Likewise, the present study aimed to analyze the performance of different high-availability solutions within a given experimental setup and provide insights into their strengths and weaknesses based on specific performance metrics.

The following are the recommendations that have emerged from the measured results. The recommendations will help cloud professionals and decision makers select the most appropriate high availability solution based on specific use cases and requirements.

High performance and data transfer: In scenarios where high data transfer speeds are preferred, Proxmox appears to be the most suitable option. Proxmox's ability to transfer large amounts of data within a certain time frame makes it ideal for data-intensive workloads such as big data processing and large-scale data analysis. Organizations that work with significant volumes of data can benefit from Proxmox's high-throughput capabilities that ensure efficient and timely data processing.

Load balancing and failover: If load balancing and failover features are essential, Docker with Docker Swarm is recommended. The Docker Swarm load balancing manager efficiently distributes requests across containers, optimizes resource utilization, and ensures high availability. For applications that require balanced traffic distribution and redundancy for failover scenarios, Docker Swarm's load balancing capabilities prove to be highly valuable.

Recovery from node failure: For environments that require fast and reliable recovery from node failures, Docker with Docker Swarm proves to be the right choice. Docker Swarm's fast recovery time minimizes downtime and ensures seamless continuity of critical services. Docker Swarm's robust node failover features contribute to increased service reliability and reduced service interruptions. The Kubernetes platform trails closely behind in the ranking hierarchy.

Minimum number of failed calls and no load balancing: If minimizing the number of failed calls without the use of load balancing is a priority, Proxmox stands out as the most suitable option. Proxmox's stability in such scenarios ensures reliable service delivery with fewer interruptions and minimal call drops. In situations where load balancing is not necessary or relevant, Proxmox's consistent performance proves advantageous for providing uninterrupted services.

Comprehensive fault tolerance: Organizations with a variety of applications requiring comprehensive fault tolerance should consider Docker with Docker Swarm. The combination of Docker Swarm's high service recovery speed, low call failure rate, and load-balancing features ensures a robust and reliable infrastructure. Docker Swarm's ability to maintain service availability and deliver reliable performance makes it an excellent choice for critical and high-availability applications.

Automatic resource scaling: For large-scale applications whose resource demands change dynamically over time, the Kubernetes platform is an appropriate choice. Autoscaling allows a Kubernetes cluster to automatically adjust the number of running pods (groups of one or more containers that run together on a single physical or virtual machine within a clustered environment; pods share resources and communicate with each other) based on workload demands. It ensures efficient resource allocation, preventing under- or over-utilization of resources. This makes it possible to optimize costs based on actual demand. When workload is low, Kubernetes can reduce the number of pods to reduce resource consumption and cost. On the contrary, during periods of high demand, the number of pods can be increased to satisfy increased workload without manual intervention. The application can handle increased traffic or peak workloads, providing end users with better performance and responsiveness.

These guidelines and recommendations are based on specific performance characteristics observed in experiments. However, it is important to note that these recommendations are not exhaustive and that each organization's decision may be influenced by its unique requirements and specific use cases. Careful evaluation of workload characteristics, scalability needs, and resource requirements will further help to select the optimal high-availability solution.

One limitation of the study may be that the evaluation focused on a limited set of container platforms, namely Docker, Kubernetes, and Proxmox. However, these platforms represent widely used tools in the industry and form the basis of many container infrastructures. Comparisons between them provide important information for practitioners and organizations to decide on the appropriate container platform for their needs. Focusing directly on these three popular container platforms allowed for relevant and comparable results. However, the container environment is constantly evolving, and new platforms and orchestration tools may emerge over time that will find widespread adoption among users.

Future investigations could extend the research to encompass a wider range of container technologies and take into account the latest advances in the industry.

In addition, it is important to note that the performance of container platforms can be affected by multiple factors including hardware configurations, network environments, and workload characteristics. In the experiments carried out, an attempt was made to provide the same conditions for all container platforms, focusing on minimizing differences between the tested tools to better compare their capabilities. However, in practice, it can be difficult to achieve a complete environment identity for each container platform, as there are multiple variables that can affect performance. As a result, performing tests in different scenarios and settings can provide a more comprehensive understanding of the capabilities and limitations of the platforms. The variety of conditions under which testing platforms can be performed can provide an understanding of how different platforms behave in different situations and which factors may have the greatest impact on their performance. In this way, research can get even closer to real-world conditions and better understand their behavior in practical scenarios. Moreover, realistic operating environments can have unpredictable impacts on the performance and availability of container platforms.

Another limitation is that direct node failure was used in the testing. In addition to direct node failure, it would be useful to address realistic failure scenarios, such as simulating hardware errors or network problems. These scenarios would help to evaluate how quickly and efficiently container platforms can respond to unforeseen events and restore applications to full functionality.

In addition, considering the limited number of nodes used in the testing phase, it is important to recognize the possible impact on generalizability of the results to larger and more complex deployments. Further investigations with a broader range of nodes would provide a more comprehensive understanding of the capabilities and performance solutions in various scenarios.

7. Conclusions

The article focused on the comparison and analysis of different high-availability solutions for Linux container infrastructures. Based on the experiments performed, data on average service recovery time, average number of data transferred, and total number of failed calls were collected. These data allowed one to identify the strengths and weaknesses of each solution and to determine the optimal container solution for common use.

Analysis of the tests found that Docker, using its high-availability manager Docker Swarm, performed best in terms of fastest service recovery, with Kubernetes coming in second place. On the other hand, the Proxmox platform had the worst results in the service recovery speed test.

For the number of failed calls, the Docker platform using its high-availability manager, Docker Swarm, was again in the best position; however, when the load balancing feature of Docker and Kubernetes was not used, the Proxmox platform emerged as the best service.

When the average call time results were tested and analyzed, the best performing service was Docker using its high-availability manager, Docker Swarm; in contrast, the worst performing service was Proxmox, which had the worst average call time across all test cases.

However, when analyzing the results of the average data transfer rate test, the Proxmox platform was the definitive winner, as it was able to transfer data the fastest.

Based on these results, it can be concluded that, in general, the best high-availability solution within commonly used Linux containers is Docker using its high-availability manager Docker Swarm. The exceptions are cases where the transfer speed of the service is more important, or where it is not a priority to use load balancing; in this case, the Proxmox platform leads the way.

However, it is important to remember that these technologies are dynamically evolving and constantly improving, and it is essential to stay up to date with the latest developments and innovations in this field. Future research will focus on investigating the latest security practices and mechanisms to protect container environments and on comparing different types of persistent storage in terms of performance, availability, scalability, and security. An important goal is also to minimize costs while improving the reliability and availability of container applications. Research should focus on innovative approaches to enable effective management of high availability in challenging and dynamic container environments. In addition, future studies could also incorporate artificial intelligence and clustering methods to improve the performance, management, and security of container solutions [45–47].

Author Contributions: Conceptualization, M.Š. and N.B.; methodology, M.Š. and N.B.; validation, M.Š. and N.B.; formal analysis, M.Š., N.B. and L.H.; investigation, N.B.; resources, M.Š., N.B. and L.H.; writing—original draft preparation, M.Š. and N.B.; writing—review and editing, M.Š. and L.H.; supervision, M.Š.; funding acquisition, L.H. All authors have read and agreed to the published version of the manuscript.

Funding: The work was funded by the Cultural and Educational Grant Agency MŠVVaŠ SR, grant number KEGA 012UCM-4/2021.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available in this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bentaleb, O.; Belloum, A.S.Z.; Sebaa, A.; El-Maouhab, A. Containerization technologies: Taxonomies, applications and challenges. *J. Supercomput.* **2022**, *78*, 1144–1181. [CrossRef]
2. Chouliaras, S.; Sotiriadis, S. Auto-scaling containerized cloud applications: A workload-driven approach. *Simul. Model. Pract. Theory* **2022**, *121*, 102654. [CrossRef]
3. Seisa, A.S.; Damigos, G.; Satpute, S.G.; Koval, A.; Nikolakopoulos, G. Edge Computing Architectures for Enabling the Realisation of the Next Generation Robotic Systems. In Proceedings of the 30th Mediterranean Conference on Control and Automation, Athens, Greece, 28 June–1 July 2022; pp. 487–493. [CrossRef]
4. Gao, X.; Xu, P. Distributed storage system for small files. In Proceedings of the 2021 2nd International Conference on Artificial Intelligence and Computer Engineering (ICAICE), Hangzhou, China, 5–7 November 2021; pp. 736–739. [CrossRef]
5. Somasekaram, P.; Calinescu, R.; Buyya, R. High-availability clusters: A taxonomy, survey, and future directions. *J. Syst. Softw.* **2022**, *187*, 111208. [CrossRef]
6. Sierra-Arriaga, F.; Branco, R.; Lee, B. Security Issues and Challenges for Virtualization Technologies. *ACM Comput. Surv.* **2020**, *53*, 1–37. [CrossRef]
7. Šimon, M.; Huraj, L.; Čerňanský, M. Performance Evaluations of IPTables Firewall Solutions under DDoS attacks. *J. Appl. Math. Stat. Inform.* **2015**, *11*, 35–45. [CrossRef]
8. Wang, C.; Xie, X.; Zheng, L.; Zhang, D. Design and Implementation of Container-based Elastic High-Availability File Preview Service Cluster. In Proceedings of the 2023 5th International Conference on Communications, Information System and Computer Engineering (CISCE), Guangzhou, China, 14–16 June 2023; pp. 57–60. [CrossRef]
9. Red Hat, Inc. What Is High Availability? 2022. Available online: https://www.redhat.com/en/topics/linux/what-is-high-availability (accessed on 9 June 2023).
10. Moravcik, M.; Segec, P.; Kontsek, M.; Uramova, J.; Papan, J. Comparison of LXC and Docker Technologies. In Proceedings of the 2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA), Košice, Slovakia, 12–13 November 2020; pp. 481–486. [CrossRef]
11. Kaur, P.; Josan, J.K.; Neeru, N. Performance analysis of docker containerization and virtualization. In Proceedings of the Third International Conference on Communication, Computing and Electronics Systems: ICCCES 2021, Tamil Nadu, India, 28–29 October 2021; Springer: Singapore, 2022; pp. 863–877. [CrossRef]
12. Putri, A.R.; Munadi, R.; Negara, R.M. Performance analysis of multi services on container Docker, LXC, and LXD. *Bull. Electr. Eng. Inform.* **2020**, *9*, 2008–2016. [CrossRef]
13. Li, G.; Takahashi, K.; Ichikawa, K.; Iida, H.; Nakasan, C.; Leelaprute, P.; Thiengburanathum, P.; Phannachitta, P. The Convergence of Container and Traditional Virtualization: Strengths Limitations. *SN Comput. Sci.* **2023**, *4*, 387. [CrossRef]
14. Malviya, A.; Dwivedi, R.K. A Comparative Analysis of Container Orchestration Tools in Cloud Computing. In Proceedings of the 2022 9th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi, India, 23–25 March 2022; pp. 698–703. [CrossRef]

15. Koziolek, H.; Eskandani, N. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, Coimbra, Portugal, 15–19 April 2023; pp. 17–29. [CrossRef]
16. Kaiser, S.; Haq, M.S.; Tosun, A.S.; Korkmaz, T. Container technologies for ARM architecture: A comprehensive survey of the state-of-the-art. *IEEE Access* **2022**, *10*, 84853–84881. [CrossRef]
17. Chengeta, K. Comparing the performance between Virtual Machines and Containers using deep learning credit models. In Proceedings of the International Conference on Artificial Intelligence and its Applications, Virtual, 9–10 December 2021; pp. 1–8. [CrossRef]
18. Bhardwaj, A.; Krishna, C.R. Virtualization in Cloud Computing: Moving from Hypervisor to Containerization—A Survey. *Arab. J. Sci. Eng.* **2021**, *46*, 8585–8601. [CrossRef]
19. Tapia, F.; Mora, M.Á.; Fuertes, W.; Aules, H.; Flores, E.; Toulkeridis, T. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Appl. Sci.* **2020**, *10*, 5797. [CrossRef]
20. Zhou, N.; Zhou, H.; Hoppe, D. Containerization for High Performance Computing Systems: Survey and Prospects. *IEEE Trans. Softw. Eng.* **2022**, *49*, 2722–2740. [CrossRef]
21. Li, W.; Kanso, A.; Gherbi, A. Leveraging Linux Containers to Achieve High Availability for Cloud Services. In Proceedings of the IEEE International Conference on Cloud Engineering, Tempe, AZ, USA, 9–13 March 2015; pp. 76–83. [CrossRef]
22. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
23. Mishra, P.; Bhatnagar, S.; Katal, A. Cloud Container Placement Policies: A Study and Comparison. In Proceedings of the Second International Conference on Computer Networks and Communication Technologies: ICCNCT 2019, Alghero, Italy, 29 September–2 October 2019; Springer International Publishing: Cham, Switzerland, 2020; pp. 513–524. [CrossRef]
24. Flauzac, O.; Mauhourat, F.; Nolot, F. A review of native container security for running applications. *Procedia Comput. Sci.* **2020**, *175*, 157–164. [CrossRef]
25. Casalicchio, E.; Iannucci, S. The state-of-the-art in container technologies: Application, orchestration and security. *Concurr. Comput. Pract. Exp.* **2020**, *32*, e5668. [CrossRef]
26. Viejo-Cortés, J.; Ruiz-De-Clavijo-Vázquez, P.; Ostúa-Arangüena, E.; Cano-Quiveu, G.; Juan-Chico, J. Virtualization environment for IT labs development and assessment. In Proceedings of the 2022 Congreso de Tecnología, Aprendizaje y Enseñanza de la Electrónica (XV Technologies Applied to Electronics Teaching Conference), Teruel, Spain, 29 June–1 July 2022; pp. 1–5. [CrossRef]
27. Ljubojević, M.; Bajić, A.; Mijić, D. Implementation of High-Availability Server Cluster by Using Fencing Concept. In Proceedings of the 2019 18th International Symposium INFOTEH-JAHORINA (INFOTEH), East Sarajevo, Bosnia and Herzegovina, 20–22 March 2019; pp. 1–5. [CrossRef]
28. Đorđević, B.; Timčenko, V.; Kraljević, N.; Jovičić, N. Performance comparison of KVM and Proxmox Type-1 Hypervisors. In Proceedings of the 2022 30th Telecommunications Forum (TELFOR), Belgrade, Serbia, 15–16 November 2022; pp. 1–4. [CrossRef]
29. Cochak, H.Z.; Koslovski, G.P.; Pillon, M.A.; Miers, C.C. RunC and Kata runtime using Docker: A network perspective comparison. In Proceedings of the 2021 IEEE Latin-American Conference on Communications (LATINCOM), Santo Domingo, Dominican Republic, 17–19 November 2021; pp. 1–6. [CrossRef]
30. Mavridis, I.; Karatza, H. Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurr. Comput. Pract. Exp.* **2023**, *35*, e6365. [CrossRef]
31. Boettiger, C. An introduction to Docker for reproducible research. *ACM SIGOPS Oper. Syst. Rev.* **2015**, *49*, 71–79. [CrossRef]
32. Škrinárová, J.; Vesel, E. Model of education and training strategy for the high performance computing. In Proceedings of the 2016 International Conference on Emerging eLearning Technologies and Applications (ICETA), Stary Smokovec, Slovakia, 24–25 November 2016; pp. 315–320. [CrossRef]
33. Nguyen, N.; Bein, D. Distributed mpi cluster with docker swarm mode. In Proceedings of the 2017 IEEE 7th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 9–11 January 2017; pp. 1–7. [CrossRef]
34. Moravcik, M.; Kontsek, M. Overview of Docker container orchestration tools. In Proceedings of the 2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA), Kosice, Slovenia, 12–13 November 2020; pp. 475–480. [CrossRef]
35. Combe, T.; Martin, A.; di Pietro, R. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Comput.* **2016**, *3*, 54–62. [CrossRef]
36. Fouda, E. Security. In *A Complete Guide to Docker for Operations and Development*; Apress: Berkeley, CA, USA, 2022. [CrossRef]
37. Alyas, T.; Ali, S.; Khan, H.U.; Samad, A.; Alissa, K.; Saleem, M.A. Container Performance and Vulnerability Management for Container Security Using Docker Engine. *Secur. Commun. Netw.* **2022**, *2022*, 6819002. [CrossRef]
38. Kim, T.; Yoo, S.E.; Kim, Y. Edge/Fog Computing Technologies for IoT Infrastructure. *Sensors* **2021**, *21*, 3001. [CrossRef] [PubMed]
39. Nguyen, T.-T.; Yeom, Y.-J.; Kim, T.; Park, D.-H.; Kim, S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration. *Sensors* **2020**, *20*, 4621. [CrossRef] [PubMed]
40. Vaño, R.; Lacalle, I.; Sowiński, P.; S-Julián, R.; Palau, C.E. Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions. *Sensors* **2023**, *23*, 2215. [CrossRef] [PubMed]

41. De, S.; Singh, R.P. Selective Analogy of Mechanisms and Tools in Kubernetes Lifecycle for Disaster Recovery. In Proceedings of the 2022 IEEE 2nd International Conference on Mobile Networks and Wireless Communications (ICMNWC), Tumkur, Karnataka, India, 2–3 December 2022; pp. 1–6. [CrossRef]
42. Robberechts, J.; Sinaeepourfard, A.; Goethals, T.; Volckaert, B. A Novel Edge-To-Cloud-As-A-Service (E2CaaS) Model for Building Software Services in Smart Cities. In Proceedings of the IEEE International Conference on Mobile Data Management, Versailles, France, 30 June–3 July 2020; Institute of Electrical and Electronics Engineers Inc.: New York, NY, USA, 2020; pp. 365–370. [CrossRef]
43. Le, D.N.; Pal, S.; Pattnaik, P.K. Cloud Database. *Cloud Comput. Solut. Archit. Data Storage Implement. Secur.* **2022**, *8*, 123–142. [CrossRef]
44. Casalicchio, E. Container orchestration: A survey. In *Systems Modeling: Methodologies and Tools*; Springer: Cham, Switzerland, 2019; pp. 221–235. [CrossRef]
45. Rani, V.; Kumar, S. Dissimilarity measure between intuitionistic Fuzzy sets and its applications in pattern recognition and clustering analysis. *J. Appl. Math. Stat. Inform.* **2023**, *19*, 61–77. [CrossRef]
46. Chiang, R.C. Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms. *Clust. Comput.* **2023**, *26*, 13–23. [CrossRef]
47. Ramos, F.; Viegas, E.; Santin, A.; Horchulhack, P.; dos Santos, R.R.; Espindola, A. A machine learning model for detection of docker-based APP overbooking on kubernetes. In Proceedings of the ICC 2021-IEEE International Conference on Communications, Montreal, QC, Canada, 14–23 June 2021; pp. 1–6. [CrossRef]