Oskari Heino

EFFICIENCY IN THE CLOUD: CONTAINERS VS. VMS

School of Technology
2025

VAASAN AMMATTIKORKEAKOULU
Cloud-Based Software Engineering

TIIVISTELMÄ

Tekijä	Oskari Heino
Opinnäytetyön nimi	Resurssitehokkuus pilvipalveluissa: Kontit vs. Virtuaalikoneet
Vuosi	2025
Kieli	Englanti
Sivumäärä	63 + 10 liitettä
Ohjaaja	Johan Dams

Tämän tutkimuksen tavoitteena on vertailla kontitettujen ja natiivien sovellusten eroavaisuuksia resurssitehokkuuden ja suorituskyvyn muodossa. Yritysten siirtyessä etenevissä määrin pilvipohjaisiin ympäristöihin, konttien ja virtuaalikoneiden suorituskyvyn ymmärtämisestä on tullut yhä tärkeämpää. Tutkimuksen tavoitteena oli selvittää miten nämä kaksi erilaista toteutustapaa eroavat toisistaan suorituskyvyn suhteen ja millaisia vaikutuksia tällä voi olla sovellus- ja infrastruktuurivalinnoissa.

Teoreettisessa osuudessa perehdytään virtualisoinnin ja konttiteknologioiden perusteisiin sekä suorituskyvyn mittaamisen keskeisiin käsitteisiin ja metriikoihin. Tutkimus toteutettiin kokonaisuudessaan hyödyntäen VPS-palvelinta, johon valitut sovellukset asennettiin mahdollisimman yhtenevin konfiguraatioin – kumpaakin toteutustapaa hyödyntäen. Tutkimukseen valitut sovellukset olivat Nginx, MySQL ja Redis.

Keskeisenä havaintona tutkimuksessa ilmeni, että natiivisti asennetut sovellukset saavuttivat huomattavasti paremman suorituskyvyn lähes jokaisessa suoritetussa testissä. Konttien asennus ja yleinen hallittavuus osoittautuivat kuitenkin huomattavasti helpommiksi ja joustavammiksi.

Johtopäätöksenä voidaankin todeta, että valinta konttien ja natiivien sovellustoteutusten välillä on pitkälti riippuvainen käyttötarkoituksesta ja prioriteeteista, kuten suorituskyvystä, skaalautuvuudesta ja yleisestä käytön helppoudesta.

Avainsanat	konttiteknologiat, virtualisointi, pilvipalvelut, pilvilaskenta

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Cloud-Based Software Engineering

ABSTRACT

Author	Oskari Heino
Title	EFFICIENCY IN THE CLOUD: CONTAINERS VS. VMS
Year	2025
Language	English
Pages	63 + 10 Appendices
Name of Supervisor	Johan Dams

This thesis explores the performance differences in containerized and natively installed applications. Nowadays, companies are increasingly transitioning to cloud-based environments where such solutions are omnipresent. The goal of the study was to determine how these two implementations differ from each other performance-wise and what kind of ramifications could these differences have on application and infrastructure decisions.

The theoretical section dives deeper into the basics of virtualization and containerization and explains the key metrics examined. The whole research was conducted using experimental methods, utilizing a virtual private server with comparable application configurations. Both container-based and native. The selected applications were Nginx, MySQL and Redis.

The central finding of the thesis was that native implementations achieved significantly better performance throughout the testing process. However, the deployment and overall management of containerized applications proved out to be much simpler and flexible.

As a conclusion, it can be stated that the choice between a containerized or native installations are highly dependent on usage and priorities, such as performance, scalability and ease-of-use.

Keywords containerization, virtualization, cloud-based technologies, cloud computing

CONTENTS

LIST OF FIGURES AND TABLES

APPENDICES

APPENDIX A. Nginx configuration files

APPENDIX B. MySQL configuration files

APPENDIX C. Redis configuration files

APPENDIX D. Benchmarking scripts

APPENDIX E. Full benchmark result tables

1 INTRODUCTION

With the growing popularity of microservices, the need for efficient computing power and ever-changing technological advances raises the question of which technology offers the most balanced performance and reliability.

The two isolation-offering information technology advancements being compared are virtualization and containerization. Both are fundamental and perhaps even cornerstones in many a section of modern-day information technology. These technologies offer scalability, flexibility and most importantly improved productivity.

Virtualization and containerization as technologies are essential in cloud computing, development, testing and production environments, however, choosing between the two often depends on various compatibility and performance factors.

The purpose of this thesis is to have a deeper look at these technologies, what they have to offer, how are they capitalized and how will they compare to each other in a detailed performance and reliability analysis under various workloads.

1.1 Background

Traditionally, a business operates via running applications on servers. Usually one application per server. This model was never sustainable as often these servers would only be capitalizing a small portion of the available resources by running a single application or service. Thus why, virtualization and containerization were created to maximize the resource utilization and cost-effectiveness of computing power. (IBM, 2023)

These technologies, at their core, offer better resource utilization and can be helpful in simplifying and creating scalable, efficient computing environments. As organizations continue to migrate towards more cloud-based architectures, the choice between containerization and virtualization becomes more relevant.

1.2 Research Questions

The primary objective of this thesis is to compare two technologies: containerization and virtualization. Specifically, in terms of performance and reliability. How do these technologies compare under stressful conditions in modern testing and production environments.

Containers and virtual machines are two distinct approaches to creating isolated IT environments by combining various IT components. The key difference lies in the specific components that are isolated. (Red Hat, 2023)

To be able to answer this hypothesis comprehensively, the study addresses the following sub-questions and hypotheses:

1. How do the performance metrics compare in terms of CPU and memory utilization under high load?
2. How stable are containers and virtual machines when running network-intensive applications?

1.3 Scope

The thesis focuses on identifying the key differences between containerization and virtualization, with performance and reliability as the primary areas of comparison. The study focuses on commonly used applications in the field of information technology worldwide. The study will be conducted with a virtual private server (VPS) that is leased from Contabo, a global cloud provider based in Germany.

The selected applications undergo compute-intensive tasks to determine the impact on the system resources. The key performance metrics being compared are CPU usage, memory utilization, disk I/O and network performance. Cost-effectiveness and security considerations are outside the scope of this study.

While the results of the research may not produce a single definitive answer, they will provide valuable insight for IT professionals and DevOps engineers in choosing the appropriate technology for their workloads.

1.4 Structure of the Thesis

The thesis consists of five chapters and follows a structured order of execution:

- Chapter 1 begins with the introduction, explaining the background and motivation for the research.
- Chapter 2 focuses on the theoretical side of the thesis, such as the compared metrics and used technologies providing an overview of them.
- Chapter 3 examines through the practical side of the thesis – the methodology and how the research is conducted.
- Chapter 4 presents the analytical findings, discussing the results obtained from the research.
- Chapter 5 summarizes the comparisons and discusses the broader implications of the findings.

2 THEORETICAL FRAMEWORK

This chapter provides the theoretical foundation for the thesis and works as an introduction focused on the key technologies, concepts and collected performance metrics.

2.1 Virtualization

Virtualization is a technology in which access to a piece of hardware is coordinated in a way that multiple guest operating systems can share the same resource, in isolation. It allows the creation of multiple virtual, simulated environments from a single physical machine. There are multiple different virtualization types. Including server, desktop, data, storage, application and network virtualization. However, this thesis will mostly focus on server virtualization. (Red Hat, 2024)

Server virtualization, often referred to as the most common type of virtualization, can be thought of as inserting another encapsulated layer on top of the original layer. The added layer will capitalize the same hardware and resources as the original layer. These layers might have completely different operating systems, configurations and use cases. The underlying hypervisor handles the communication between the used hardware and the operating systems, meaning that the encapsulated layers can focus on existing and believing that they are the sole owner of these resources. Figure 1 below illustrates how a typical VM set up looks like. (Golden, 2007)

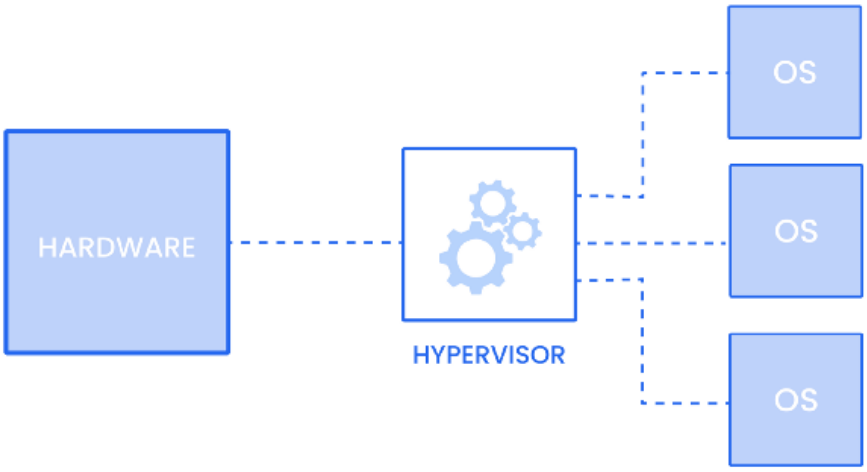

Figure 1. High-level architecture of a typical hypervisor enabled virtual machine (ByteHide, 2023).

Hypervisor-based virtualization has been a crucial part of the development of cloud computing for decades. Hypervisor enables the distribution of physical hardware resources for virtualized machines to be used separately from other machines in the same environment. According to Shirinbab et al. (2020), the additional abstraction layer added by the hypervisor could introduce performance degradation. This could cause limitations within performance-critical applications. The hypervisor introduced overhead has increased the popularity of container-based solutions in environments where resource efficiency and swiftness are critical.

Virtualization as a concept date back all the way to the early 1960s, meaning it has been a part of the computing field for over half a century. During the late 1960s and early 1970s IBM developed a time-sharing operating system called Control Program/Cambridge Monitor System (CP/CMS). In modern terminology, CP is the hypervisor and the VM, whereas CMS is a lightweight operating system running on the VM created by the CP. (“CP/CMS,” 2024)

These systems together allowed separate users to run an isolated system within a single time-shared computing environment. Basically, providing each user with a simulated, stand-alone computer. (“CP/CMS,” 2024)

Virtualization has come a long way since the early days. As a technology, it took off on a larger scale in the late 1990s and early 2000s as the demand for computing power rapidly increased. Today it is standard practice in global IT infrastructure.

According to Gartner Inc., the global end-user spending on public cloud services is forecasted to reach a total of 723,4 billion in 2025, a significant increase from 595,7 billion in 2024. This growth highlights the need for computing power and increasing reliance on virtualized infrastructures to meet the demands of modern IT environments. (Gartner, 2024)

2.1.1 Concepts

Virtualization enables the sharing of physical computing resources, allowing several virtual machines to operate independently on a physical server under the management of a hypervisor. The hypervisor acts as an intermediary between the physical server and the VM. It coordinates the creation of virtual environments and manages access to the required resources while ensuring the isolated nature of VMs. (Red Hat, 2024)

Generally, there are two types of hypervisors:

- Type 1 hypervisor runs directly on the host hardware without requiring an underlying operating system. Type 1 hypervisor is often referred to as a native or bare-metal hypervisor. Examples include VMware ESXi and Microsoft Hyper-V. (IBM, 2023)
- Type 2 hypervisor runs on top of an existing operating system as a software or application, such as VirtualBox. This type of hypervisor is often used for development and testing environments. (IBM, 2023)

A fundamental advantage of virtualization is its ability to isolate workloads – while ensuring that failures or security vulnerabilities in one VM do not affect other VMs running on the same physical hardware. The isolated nature is especially critical in

multi-tenant environments, where multiple applications or customers might operate under a shared infrastructure. (Red Hat, 2024)

Another key benefit of virtualization is resource efficiency. In a traditional non-virtualized environment, when an application is idle on a server, a significant portion of the computing resources remain unused. However, by virtualizing the server and slicing it into multiple VMs, workloads can be allocated automatically, reducing the unused computing power and increasing overall efficiency. (Shamir, 2021)

In addition to improved efficiency and isolation, virtualization offers:

- Scalability – virtualized environments support horizontal scaling, meaning that additional VMs can be provisioned quickly according to demand.
- Minimal downtime – application and OS crashes might cause significant downtime and disrupt productivity. With virtualization, running multiple redundant VMs alongside each other will increase high availability and reduce possible downtime.
- Optimized maintenance – VMs support migrating, cloning and snapshotting, allowing them to be restored to a previous state reducing the complex upkeep and management of physical infrastructure.

By integrating these capabilities, virtualization can be a powerful tool that enhances performance, reliability, and resource efficiency – factors that are crucial for any modern IT environment. (Shamir, 2021)

2.1.2 Use Cases

Virtualization has been a widely adopted technology in various IT domains for decades due to its ability to optimize hardware utilization. Companies are able to improve resource efficiency by virtualizing physical servers to multiple virtual ones, this process is called server consolidation. This approach ensures that hardware resources are utilized efficiently, and the infrastructure costs are reduced.

In cloud computing, virtualization enables Infrastructure as a Service (IaaS) by allocating resources based on demand. The ability to scale infrastructure without significant investment in physical hardware is beneficial for many businesses regardless of the size.

Another critical use case is software development and testing. Virtualization allows developers to have their own isolated IT environments across different operating systems. This reduces compatibility issues, streamlines the development process and enables effective debugging without the need for multiple physical machines. (Red Hat, 2024)

2.2 Containerization

The development of containerization began with resource efficiency in mind. Traditional virtualization allocates a set amount of resources for the virtualized entity, which in many cases leads to overallocation with excess resources as virtual machines do not fully capitalize on all reserved resources. In addition, hypervisor-based virtualization has a limit for how many instances can run on a host machine without significant overload. These issues led to the development of a more lightweight and resource efficient solution: containerization. By using the resources needed without excess resource reservation, containers enable more instances on a single host due to improved distribution of resources compared to traditional virtualization. (Gupta, 2015)

Containerization is a lightweight virtualization method that packages all necessary software components into an isolated container – these components include libraries, configuration files and other dependencies required for the container's successful execution. Unlike VMs, that virtualize an entire machine from the OS down to hardware layers, containers run on top of a container runtime platform that shares the underlying host operating system's kernel. This approach minimizes overhead and reduces common compatibility issues associated with traditional VMs. (Susnjara & Smalley, 2024)

Although the roots of containerization trace back several decades, the lightweight virtualization technology had its breakthrough in 2013 with the release of Docker. Docker, considered one of the most popular container platforms, was first released in 2013 by Docker, Inc. Docker made containerization more user-friendly by simplifying the containerization process, and offering a platform for running the containers in different environments. Docker's ease-of-use, clear standardization and strong compatibility facilitated the speedy wide-spread adoption of it. (Susnjara & Smalley, 2024)

Following Docker's release, containerization technologies gained significant popularity, which in turn accelerated the technological development in the field. Especially after Google released an open-source container orchestration system Kubernetes in 2014, the development of containerization technologies escalated. The orchestration, scalability and management of containers experienced rapid advancements which strengthened the foothold of containerization in modern IT infrastructure. (Red Hat, 2021)

Today, containerization is one of the most central solutions for deploying and maintaining applications in DevOps, software development and IT architecture globally. It is present on practically all major cloud platforms such as Amazon ECS, Google Kubernetes Engine and Microsoft Azure Kubernetes Service. In cloud platforms, containerization enables swift and resource efficient software development and a flexible, effortlessly scalable infrastructure. (Susnjara & Smalley, 2024)

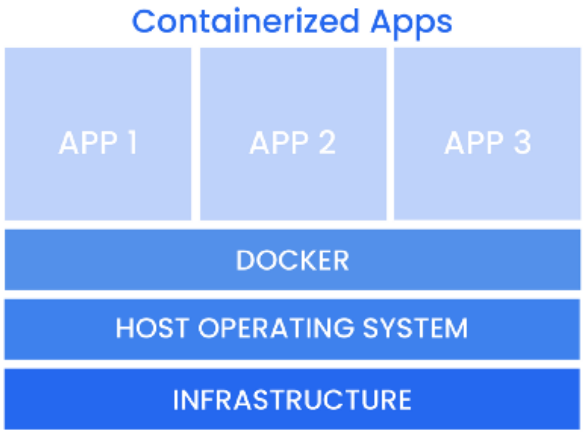

Figure 2. High-level architecture of a typical container-based environment (ByteHide, 2023).

2.2.1 Concepts

Containerization enables the packaging of software components and their dependencies into a small, isolated container. A container runtime acts as the core component managing the container lifecycle, resource allocation and isolation. Some of the most common container runtimes are Docker Engine, Podman, CRI-O and containerd. These runtimes are responsible for, among other things, starting and stopping containers and allocating resources between them. (Ehrman, 2024)

A key concept in containerization is the container image. Container images are immutable static files that include the application and its dependencies such as the code libraries and configuration files. Container images are usually small and effortlessly shareable, which makes the application deployment process speedy and flexible. (Powell & Smalley, 2024)

Another key concept with containers is isolation. Despite sharing the host machine's kernel, containers achieve isolation using separate namespaces, allowing each container to maintain independent file systems, processes and networks configurations. This enables multiple simultaneous container instances on the same virtual or physical host without any unwanted interference between them.

Containers utilize resource limiting, meaning that each container's CPU, memory and other resource usage can be assigned a ceiling, which will not be surpassed. This enables smooth and even resource distribution, ensuring stability and mitigating the possibility for a crash caused by resource overloads in individual containers.

Container orchestration is a central part of containerization technology. Orchestration tools such as Kubernetes enable automation, scalability and troubleshooting of multiple containers simultaneously. These tools automate many aspects of container management, aiding in application scaling and assist in recovering from potential issues with minimal downtime. (Powell & Smalley, 2024)

With these features containerization offers a powerful, resource efficient, modern and agile environment for scalable software deployments.

2.2.2 Use Cases

In recent years, containerization usage in modern IT architectures has grown tremendously due to the versatility and resource efficiency of containers. The ability for rapid deployment, their lightweight nature and scalability has elevated containerization as an excellent solution for managing modern applications.

One of the most significant uses cases is the microservices architecture, where containerization allows splitting applications into smaller, independent services. Each service can be developed, updated and maintained individually, which helps with the management and troubleshooting of the architecture. (Scale Computing, 2024)

Another major use case includes cloud-based application platforms and services. Leveraging containerization, major cloud providers such as AWS, Google Cloud and Microsoft Azure can offer end-users rapidly deployable and scalable applica-

tion environments. Containers enable fast and flexible resource adjustments according to demand, minimizing the costs in infrastructure for businesses of any size. (Susnjara & Smalley, 2024)

One of the benefits of containerization is strong compatibility, independent of the host machine's operating system. For example, software development and testing processes benefit from this. By using Docker containers during the application development, developers can create identical environments that are not dependent on the hardware nor OS – reducing the compatibility issues and simplifying troubleshooting and debugging. (Scale Computing, 2024)

The lightweight, modular structure of containers effectively supports continuous integration and continuous delivery (CI/CD), which are central practices in modern DevOps. CI/CD pipelines consist of standardized procedures that developers follow to automate software releases. These established procedures simplify the automation of software development and significantly improve software quality and delivery speed. (Red Hat, 2025)

The versatility of containerization has secured a solid role for it in the ever-growing modern IT infrastructure. Its flexibility, resource efficiency, and support for agile practices make it indispensable for organizations aiming for scalable, reliable, and easily maintainable application deployments.

2.3 Key Metrics for Evaluating Performance

Selecting appropriate metrics is critical when comparing the performance between containers and virtual machines. For the purposes of this thesis, I have chosen the following key metrics: CPU usage, memory utilization, disk I/O and network performance.

These metrics are expected to provide comprehensive results and accurately describe the behavior of these systems under different workloads as realistically as

possible. Thoroughly comparing these metrics will give an idea of which technology outperforms the other under different workloads. These metrics are crucial to the detailed analysis presented in Chapter 4, where the collected data is examined and compared in depth.

2.3.1 CPU

CPU usage is a central performance metric as it directly reflects the system's processing power and efficiency under various workloads. When looking at CPU usage in terms of containerization and virtualization, the efficiency and how these technologies are able to utilize the computing power is crucial.

The structural differences between these technologies notably influence CPU utilization. Virtual machines work on top of hypervisors, which will add computing overhead in the equation. The hypervisor must coordinate and schedule CPU usage across VMs, which might cause performance throttling, especially under high workload situations. Containers, on the other hand, share the host's kernel, eliminating the hypervisor layer, which in theory should enable superior performance and significantly faster startup times.

CPU performance evaluations typically consider analyzing metrics such as percentage CPU usage and peak CPU consumption during stress testing. These evaluations will provide a better understanding of how efficiently and reliably containers and VMs manage processing power.

CPU performance is a particularly critical metric in cloud-based environments and applications, where workloads are volatile and the capacity to handle load spikes is essential. When choosing the appropriate technology – for example, for maintaining micro or web services, CPU efficiency and optimal utilization of processing power become more important factors.

2.3.2 Memory

Memory utilization is a fundamental metric for performance evaluation, it indicates resource efficiency and reliability management under varying workloads. Efficient memory management is essential for system stability and service availability.

The structural differences of containers and VMs considerably impact their memory consumption and management. While VMs allocate dedicated memory resources for their individual operating systems, containers share the host system's OS kernel, mitigating the need for separate memory-consuming OS instances. Containers typically reserve memory exclusively for separate applications and their dependencies, which improves the overall resource efficiency. This memory-efficiency is particularly beneficial in environments where resources are limited or in scenarios where rapid scalability is required. VMs generally consume more memory due to deeper isolation and structural differences in virtualization level.

Memory utilization evaluations typically analyze metrics such as memory usage and peak memory consumption under stress testing. These analytics will provide better understanding of container and VM memory management.

2.3.3 DISK I/O

Disk input and output measures the system's ability to read and write data to storage devices. Efficient, competent disk management is directly affecting system latency, application response times and overall user experience. The efficiency of disk operations is particularly emphasized in tasks where a substantial amount of data is handled, such operations could be database related or moving large files.

When evaluating disk I/O performance, one of the key metrics is Input/Output Operations per Second (IOPS), which measures the number of disk operations com-

pleted per second. Another fundamental metric is latency, which refers to the average time required to complete a disk operation. Low latency is especially critical for applications or services that require minimal response times.

The third important metric is throughput. Throughput indicates the maximum amount of data possible to be read from or written to storage devices within a specific period. High throughput is especially important when processing large files or data streams, such as in media services or big data solutions.

Analyzing these metrics provides valuable insight for performance comparison, as disk operations and I/O efficiency carries a pivotal role in the overall system performance.

2.3.4 Network

Network performance, like the previous metrics, is integral when evaluating the reliability and efficiency of a system. Application latency, data transfer efficiency and pleasant user experience are all dependent on reliable network resource management. The importance of network performance notably occurs in network-intensive environments and applications, where continuous network communication and data transmission are common.

Central metrics include network throughput, latency and packet loss. Network throughput indicates the amount of network packets a system can process in a given amount of time. Latency, on the other hand, refers to how much time elapsed during the traveling of a data packet from point A to point B. The third key metric is packet loss, which depicts the reliability and quality of the network. Packet loss occurs when one or more transmitted data packets are lost during transmission, potentially causing disruptions and noticeable performance degradation.

Analyzing these metrics helps with identifying how effectively containers and virtual machines utilize and manage network resources. Assessing network performance in specific scenarios supports determining the appropriate technology for different applications and environments.

3 RESEARCH DESIGN AND METHODOLOGY

This chapter describes the key aspects related to the implementation and methods of the study. The objective is to produce a comprehensive, practical comparison of performance and reliability between virtual machines and containers under varying workloads. To achieve this, a structured quantitative approach is applied – where data is collected systematically with controlled testing.

This chapter further elaborates the applications and industry-standard tools used in data collection, monitoring and visualization. More detailed research questions, experimental setups and tested applications are presented in the subsequent sections.

3.1 Research Design

In this thesis, the research is approached using quantitative, experimental methods, which enable objective, metric-based and comparative data collection for further analysis.

The testing is conducted on a controlled virtual private server (VPS) provided by an international cloud service provider Contabo. This VPS serves as the platform for data and performance collection platform for both containerized and native (VM-level) application deployments. The testing will be performed on each technology respectively, with a set of real-world applications and services – all commonly used in IT environments globally.

The objective is to analyze and compare performance related metrics to identify how applications running on containers and virtual machines differ under various workloads.

Selected applications include web servers, databases and other services universally utilized by both technologies.

3.2 Experimental Design

The experimental study was carried out in a controlled test environment, where performance measurements were done respectively for each technology. Both technologies utilized the same environment, ensuring comparable results.

The following subchapters will provide a detailed description of the test environment used, monitoring tools, container environment specifications and the deployment of native applications.

3.2.1 VPS Configuration

The virtual private server used in the thesis is leased via Contabo. Contabo offers a decently powerful VPS for less than seven euros a month. The current setup includes 4 vCPUs, 6 GB of RAM, 100 GB of NVMe storage and a possibility for one snapshot. I chose Contabo as the VPS provider because of the cost-effectiveness, known reliability and recommendations I received and read online.

The OS is Debian 12 and hosting server is located in Dusseldorf, Germany to allow minimal latency. The bandwidth is limited to 32 TB of egress data set to 200 Mbit/s per month. Ingress data is unmetered and unlimited. Data exceeding the limits will be throttled with a limited bandwidth, however, I believe there will not be any issues with exceeding the monthly limits.

The capability for a single snapshot could be helpful in reverting the VPS environment to a baseline configuration between tests to ensure consistency. The 4 vCPUs and 6 GB of RAM are sufficient for running containerized and virtualized workload respectively, ensuring a reliable and well-performing environment under moderate stress testing.

Debian 12 was selected for great compatibility between the technologies and tools used in the thesis. The OS in its lightweight nature offers minimal overhead on resources and allows for efficient performance testing.

One limitation to the usage of a virtual private server is the inability to have a nested virtualization. Meaning that the applications must be run directly on the VPS rather than within a virtualized environment, however, this downside could be beneficial with lowering latency as unnecessary virtualization layer is absent. Nonetheless, such environment style is regularly used in testing and even production.

3.2.2 Monitoring Tools

The monitoring tools utilized in this research are widely adopted, open-source solutions that enable efficient collection and visualization of the respective data. The selected tools include:

- Prometheus, an open-source systems monitoring tool initially deployed in 2012, collects metrics data by scraping configured HTTP endpoints and stores it in a time series database.
- Grafana acts as the visualization platform, where the collected data metrics are displayed in clear, easy-to-interpret graphs.
- Node Exporter, an extension of Prometheus, introduces detailed host-level metrics such as CPU, memory, disk and network usage.
- Process Exporter, which focuses more on scraping process-level metrics.
- cAdvisor, on the other hand, is a tool designed to scrape and provide detailed metrics specifically from running containers.

Together, these tools collectively offer a comprehensive, detailed view of the performance and resource management of both the entire system as well as individual containers. The gathered data provides an accurate and comparable basis for performance comparison between VMs and containers.

All monitoring tools are running as separate containers during the performance testing phase, for both virtualization and containerization. Each tool is deployed

using Docker Compose, ensuring consistency and simplified management during workload testing.

3.2.3 Docker Environment Setup

Docker was selected as the containerization platform due to its familiarity, wide adoption and flexible configurability. Docker is an open-source platform for automating the deployment, management and testing of applications and services by packaging software into standardized units called containers.

In this study, containers are managed using Docker Compose, which is a tool for managing multi-container applications. Docker Compose uses YAML files to define the application environment, which allows the presentation of an entire application in a single file. Each application running as a container has their respective docker-compose.yml -file, which works as a blueprint for the container it sparks up.

Testing is conducted using Docker version 20.10.24 and Docker Compose version 1.29.2. Detailed Docker Compose configurations are provided in the appendices section.

3.2.4 Native Environment Setup

The applications running directly on the VPS without any containerization were installed using APT package manager. Also, the utilized stress and benchmarking tools were installed directly on the VPS.

The application configuration files were modified to match the configurations of the containerized applications, to ensure balanced and fair testing. Services are started and managed using the existing system tools, such as systemctl or directly via command line.

The operating system used is Debian 12. Detailed application configurations are provided in the appendices section.

3.3 Applications and Workloads for Testing

This section presents a brief overview of the applications and workloads involved in the testing phase. The selected applications represent commonly used IT services in both testing and production environments. These applications provide a comprehensive picture of the efficiency of containerization and virtualization under different scenarios, including CPU-, memory-, disk- and network-intensive workloads. Applications and workload scenarios are described in detail in the following subchapters.

3.3.1 Web Server – Nginx

Nginx is an open-source web server with multiple capabilities, such as operating as a load balancer, reverse proxy or HTTP caching server. It can also function as a proxy server for several mail protocols, like SMTP, IMAP and POP3. According to W3Techs (2025), Nginx has been the most used web server technology since 2022, when it surpassed Apache as the long-time market leader. (Solo.io, n.d.)

Nginx is an ideal choice for both static and dynamic web pages, which has made it the most used option in various IT environments, including testing, development and production scenarios.

In the testing setup, Nginx instances are configured with identical configurations for both Docker and application running directly on the virtual machine (VPS). Stress testing is conducted using a modern HTTP benchmarking tool called wrk. With wrk, it is possible to simulate large number of connections and concurrent requests, which allows flexible and realistic network traffic load testing. Wrk's advantages are its lightweight nature and ease of use, making it an ideal tool for the purposes of this thesis.

The key performance metrics are CPU usage, memory utilization and network performance such as latency, throughput and error rate. These metrics are visualized

and analyzed using Prometheus- and Grafana, providing a clear and comprehensive overview of performance differences between containerized and virtualized environments.

3.3.2 Database – MySQL

MySQL was chosen because of its widespread usage. Benchmarking is done using Sysbench, which is a scriptable benchmarking tool that offers ready templates for MySQL benchmarking purposes. Sysbench is used to test each technology respectively: a containerized MySQL instance and MySQL database running directly on the VPS. Sysbench produces realistic load for the database by simulating concurrent connections and creating read and write queries.

Testing is conducted with two separate instances of MySQL, one deployed directly on the virtual machine (VPS) and the other deployed as a Docker container. Testing will measure key performance metrics, such as CPU and memory utilization, disk input and output and network performance under various stressful loads.

The testing is performed using Sysbench benchmarking tool, where concurrent connections and randomly generated read and write queries are increased after every iteration. This approach will enable a comprehensive and realistic comparison between the application's behavior on each platform.

The results are visualized and analyzed using Prometheus and Grafana as monitoring tools.

3.3.3 Redis

Redis is a fast and widely adopted in-memory database. Among other things, Redis is often used as a caching server, NoSQL database or lightweight session management tool. Unlike traditional databases, Redis stores data primarily in the host sys-

tem's memory (RAM), making it significantly faster than traditional disk-based database solutions. For this reason, Redis is popular in applications where latency is a critical factor for functionality. (Redis, n.d.)

Redis was chosen as one of the applications for stress testing due to its popularity and memory-intensive nature, offering more variety in the test application pool. Redis includes a built-in performance testing tool called redis-benchmark, which can simulate realistic and highly intensive concurrent load.

In the testing environment, Redis is configured to run directly on the VPS and inside a Docker container, respectively. The key metrics being monitored and analyzed are memory utilization, CPU load, latency and the number of queries processed per second (throughput). These metrics provide a comprehensive picture of the differences between containers and virtual machines, particularly under memory-intensive workloads.

The obtained results are visualized and analyzed in using Prometheus and Grafana as monitoring tools.

4 RESULTS AND DISCUSSION

This chapter presents the benchmarking of the selected applications and their results. The fundamental element for these tests was to compare the behavior of native and containerized applications under varying workloads.

Each subchapter introduces selected metrics, such as latency, throughput, CPU load and memory utilization. The results are analyzed application-specific and towards the end summarized between the two technologies.

The main goal is to recognize performance differences and consider the implications in real-life scenarios.

4.1 Nginx Performance Analysis

The performance testing of Nginx web servers was conducted using an HTTP-benchmarking tool called wrk. The testing was done using an automated script running a wrk command with gradually increasing concurrency levels, from 50 to 25 000 concurrent connections. While analyzing the results, it became apparent that a concurrency level of 5000 and above distorted and degraded the data. Thus, the final analysis was done using a concurrency spread of 50 to 5000 for clarity. The tests ran for three minutes for each concurrency level with a two-minute break between cycles to allow for the system and resources to reset. The long duration ensures sufficient sample size for latency and throughput metrics, while avoiding unwanted noise from fluctuating short-term inconsistencies.

Both Nginx instances were running with identical configurations (v.1.22.1) for a balanced and fair comparison. Each web server instance was configured with a static HTML web page with text and a high definition .jpeg picture. All tests were executed on the same VPS environment, alternating between native and containerized instances to eliminate hardware-related performance altercations.

The results are presented and analyzed in the subsequent chapters using clear charts portraying the key metrics, such as latency, request throughput, data throughput and error rate. In addition, CPU and memory usage trends collected with Grafana will complete the overall picture of the compared environments.

The command for stress testing explained:

```
wrk -t4 -c<CONCURRENCY> -d180s http://<vps_ip_address>:80
```

- Parameter -t defines the number of utilized threads.
- Parameter -c specifies the number of concurrent connections.
- Parameter -d states the test's duration.
- The target website is at the end of the command.

The detailed configurations and test results for all compared metrics are presented in the appendices.

4.1.1 Latency Analysis

The figure below shows the average latency difference between a native and a containerized Nginx applications under various concurrent workloads (Figure 3). These tests at changing concurrency levels simulate real-world scenarios, such as when a service experiences increasing user traffic or stress during peak hours.

The results show a clear performance advantage with the native Nginx instance up to 4000 concurrent connections. In the native Nginx instance, the average latency is significantly lower, whereas containerized Nginx runs with slightly higher latency. The added virtualization layer with the container-based Nginx web server could explain the higher latency.

Interestingly, after 4000 concurrent connections, native Nginx servers' latency spikes to 82,6 milliseconds with 5000 concurrent connections, while containerized Nginx server increases more moderately to 71,6 milliseconds. This may happen because the native Nginx instance reaches its resource limitations faster, whereas

Docker's internal resource management divides the load more evenly, although with lower overall performance.

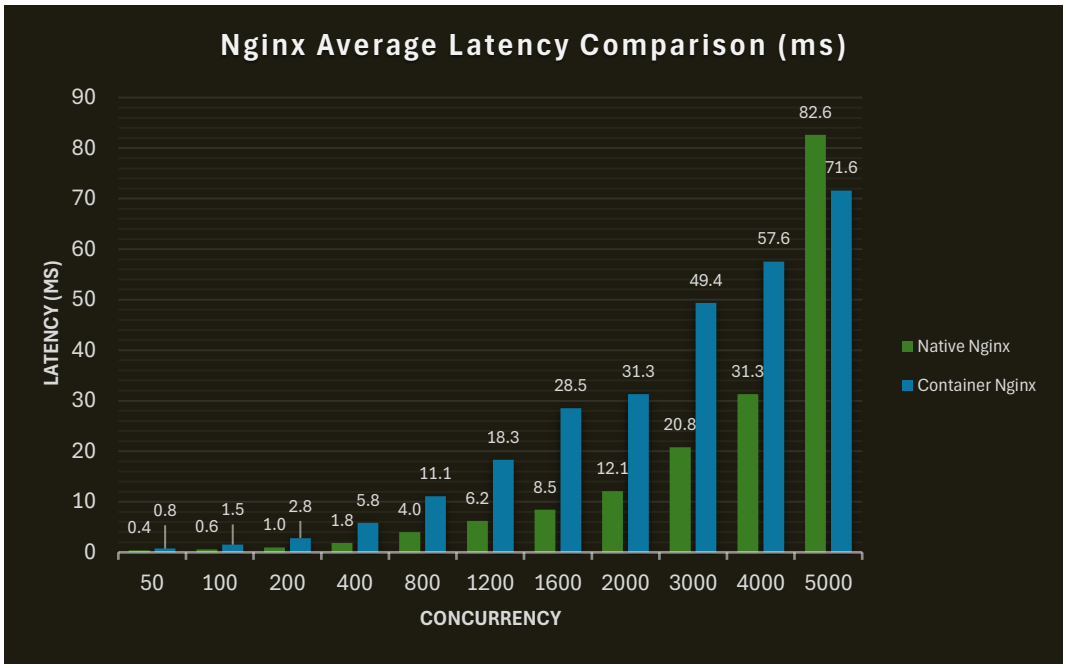

Figure 3. Average latency comparison between a containerized and a native Nginx.

Overall, the results suggest that native Nginx web server offers better raw performance for latency-sensitive workloads, while container-based Nginx web server might provide more consistent latency under spiking, highly stressful workloads.

Key findings:

- Native Nginx's average latency is significantly lower overall.
- With 4000+ concurrent connections, container-based Nginx remains more stable.
- The native instances' maximum latency efficiency is achieved with sub-4000 concurrent connections.
- The results portray the added overhead of additional isolation offered by containerized applications. Whether this is advantageous or disadvantageous depends solely on the workload.

4.1.2 Request Throughput Analysis

The figure below portrays the Nginx web servers' ability to handle HTTP requests per second under various concurrency levels (Figure 4).

The results imply that a native Nginx instance offers significantly higher request throughput throughout benchmarking. It reaches the peak throughput at approximately 400 concurrent connections (200771 req/s), after which the performance begins a slow decrease under an increasing workload.

Containerized Nginx web servers' throughput is on average 40% of the native web servers' throughput. However, its performance remains stable and balanced throughout the tests. This may be due to the added overhead of Docker's isolation layer, including resource management and virtualized networking, which evenly distributes the workload and helps avoid sharp performance drops.

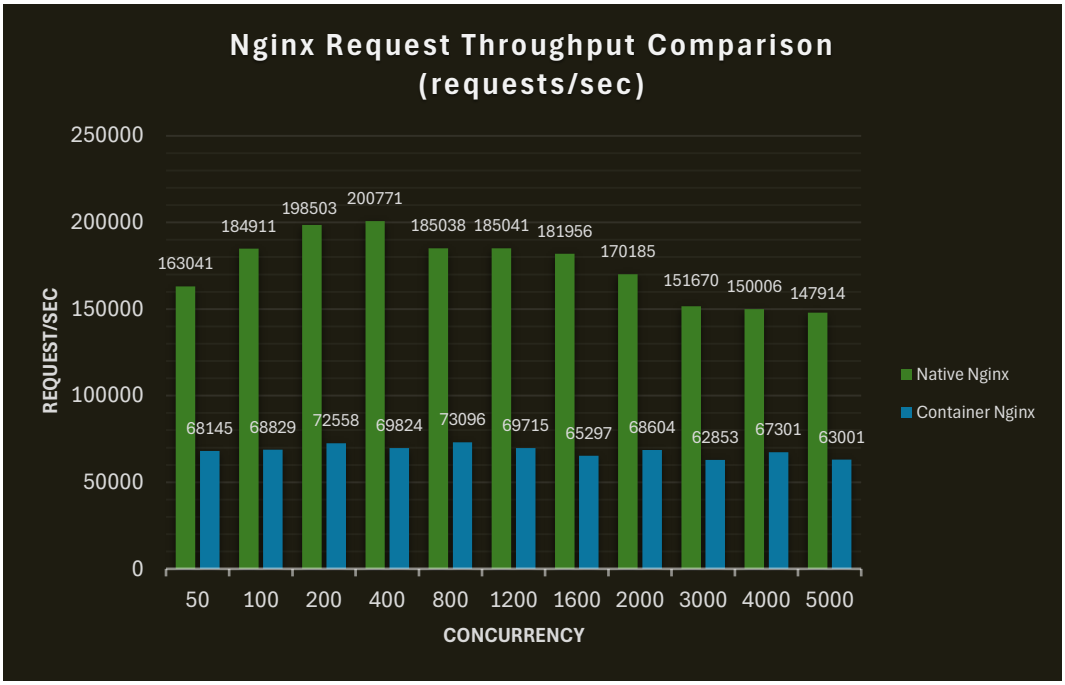

Figure 4. HTTP request throughput comparison.

Key findings:

- Native Nginx reaches its maximum throughput at approximately 400 concurrent connections.
- Container-based Nginx's throughput is, on average, around 40% of its native counterpart.
- The performance of both instances starts to decline after 1600-2000 concurrent connections, indicating resource constraints on the test machine.
- The lesser performing, yet stable nature of the containerized Nginx could be beneficial in certain production environments, where predictability is more important than raw performance.

4.1.3 Data Throughput Analysis

The figure below depicts the differences in data throughput between configured Nginx web servers with varying concurrency levels (Figure 5). Data transfer throughput (MB/s) refers to the amount of data a server can transfer per second when responding to HTTP requests.

The results show a clear performance advantage for the native instance at all load levels. The maximum throughput happens at approximately 400 concurrent connections (102 MB/s), after which throughput starts a slow decline, however, remaining at a high level. This may indicate host system resource starvation towards higher concurrency levels.

Containerized Nginx displays significantly lower throughput. The performance remains consistent with little fluctuation, which may indicate that Docker's abstract layer, offering isolation and resource management, may degrade the data transfer speed compared to the native installation.

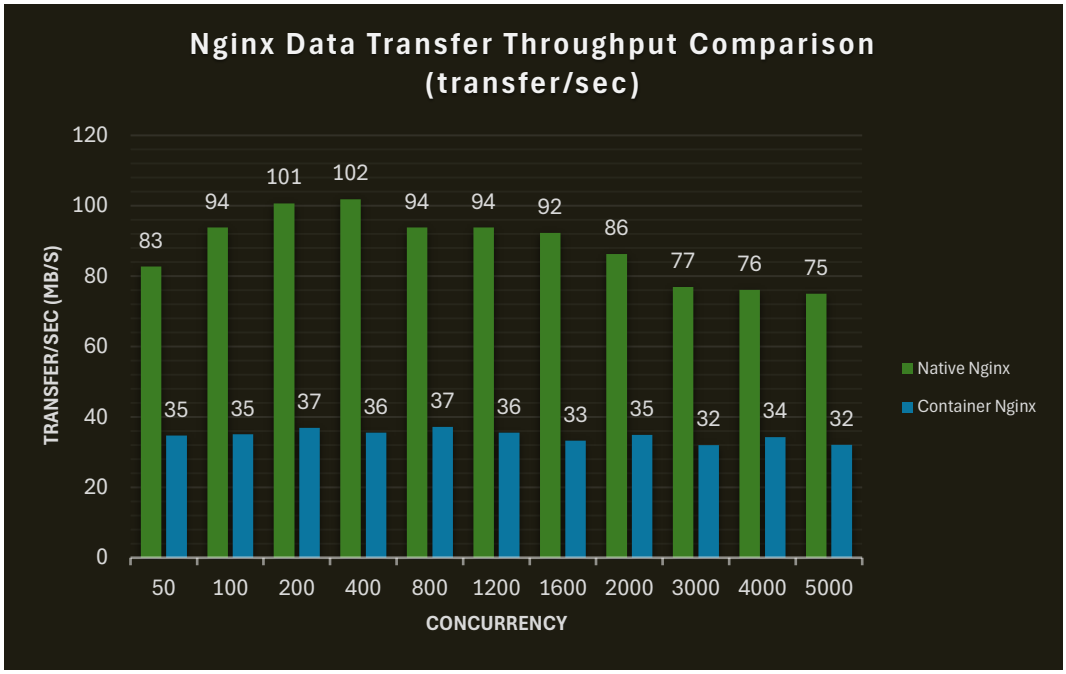

Figure 5. Chart comparing the data transfer throughput.

Key findings:

- Native Nginx outperforms containerized Nginx instance at all load levels.
- The maximum throughput is obtained at around 400 concurrent connections before the beginning of a gentle decline.
- The results support previous findings on the impact of Docker's added overhead on performance.

4.1.4 Error Rate Analysis

The figure below presents the rate of erroneous responses occurring during stress testing of native and containerized Nginx web servers (Figure 6). The results show that each technology avoided errors quite successfully until higher loads.

The first errors happened at 1600 concurrent connections on container-based Nginx, whereas native Nginx survived without errors until 3000 concurrent connections. However, after 4000-5000 concurrent connections, the number of errors grows exponentially in both instances. Compared to containerized instance, the

native Nginx server suffers from large number of errors after 4000+ concurrent connections. Which may suggest a tendency of a quicker overload in extremely stressful situations.

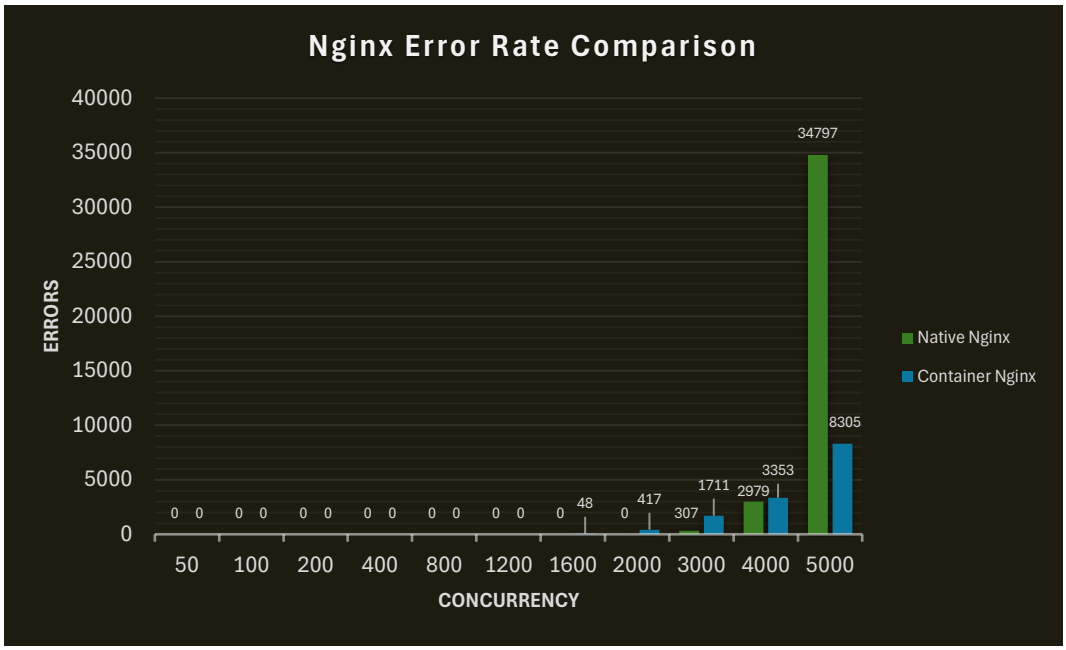

Figure 6. Chart comparing the rate of errors happening during data transfer.

Key findings:

- Both instances remained consistent up to 1600 concurrent connections.
- The number of errors grows rapidly after 3000+ concurrent connections.
- The number of errors for native instance is significantly higher than containerized instance at 5000 concurrent connections, indicating resource overload.

4.1.5 CPU and Memory Utilization

The following figures compare the CPU and memory utilization of the configured Nginx instances under varying workloads (Figure 7 & Figure 8). The results were collected using Prometheus and Grafana tools during the one-hour stress testing period.

Memory utilization shows a clear difference between native and containerized Nginx web server. The native instances' memory usage remains steady between 20-25 MB with increasing workload, indicating that the load produced by wrk-benchmarking tool is not causing a significant memory strain.

The containerized Nginx instances' memory utilization works more proactively. Idle memory usage begins at 10 MB and gradually increases parallel to the increasing concurrency level up to 60 MB. This dynamic and active fluctuation suggests that the Docker environment manages resources more efficiently, however, could cause a higher memory usage under continuously increasing load conditions.

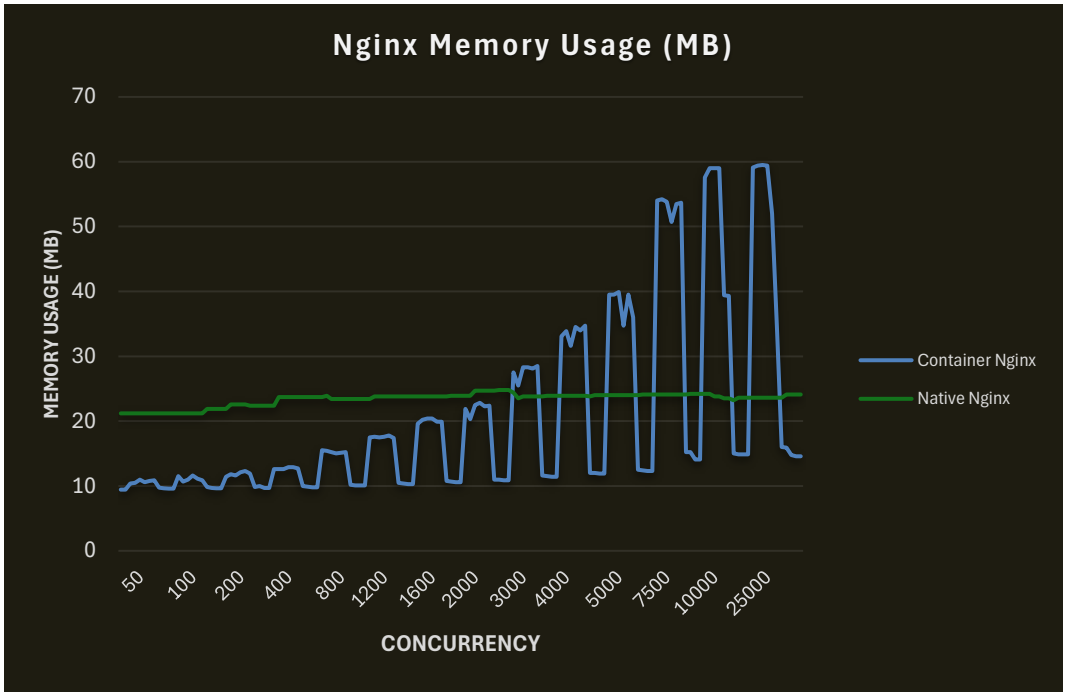

Figure 7. Nginx memory utilization comparison.

CPU utilization presents a significant difference between the two technologies. While Native Nginx reaches 270 % CPU usage throughout the testing period, containerized Nginx CPU usage stays at a moderate 160-170 % under varying workloads. This suggests that the native instance utilizes the available resources more efficiently.

It is worth noting that each technology recovers from highly stressful load situations rapidly back to idling numbers, indicating that both environments are able to release used resources without jamming overload.

For clarity, the maximum utilization in the figure is 400 %, because the test machine has 4 vCPUs available.

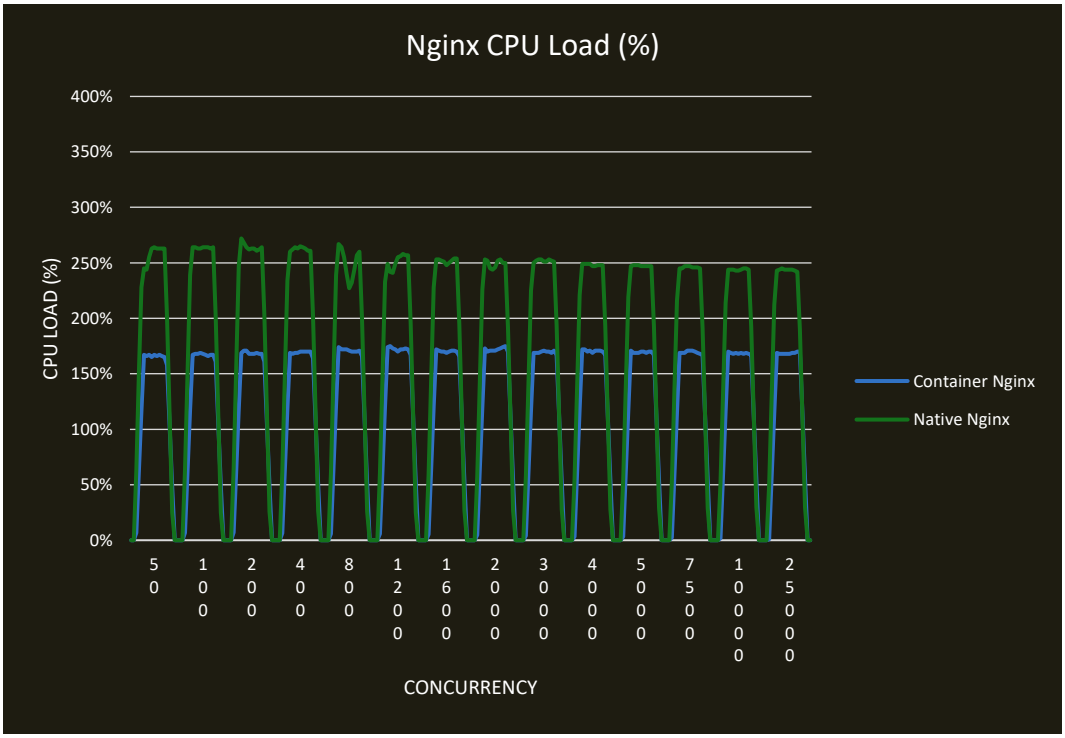

Figure 8. Nginx CPU load comparison.

Key findings:

- Native Nginx memory utilization is steadier throughout the testing process.
- Container-based Nginx utilizes memory more dynamically, proactively increasing with the increasing load.
- Native Nginx reaches significantly higher CPU load, which may indicate better CPU efficiency.
- Each technology recovers from workloads rapidly, showcasing good resource management and adaptability to various scenarios.

4.2 MySQL Performance Analysis

MySQL database instances were benchmarked using a popular open-source benchmarking tool Sysbench, which is especially popular with database stress testing operations. The goal was to compare natively installed MySQL database and container-based MySQL database with similar configurations under varying workloads.

Both MySQL instances were running version 8.4.4 with each database configured identically. The oltp_read_write.lua, included with Sysbench installation, is a ready LUA-based script template that simulates realistic workload, both read and write, against the databases. The script was automated to mitigate human error and ensure a balanced and fair comparison.

The command below defines the benchmarking parameters:

```
sysbench /usr/share/sysbench/oltp_read_write.lua \
    --mysql-host=127.0.0.1 \
    --mysql-port=3306 \
    --mysql-user=root \
    --mysql-password=thesis \
    --mysql-db=sbtest \
    --table_size=100000 \
    --tables=10 \
    --threads=<10, 20, 50, 100, 200, 300, 400, 500, 1000, 2000> \
    --time=180s run > native _${THREAD}t.txt
```

The important variable here is the threads. After each iteration the number of threads increases to simulate growing workloads. The benchmarking duration was set to 180s to ensure acceptable sample size, while avoiding unwanted noise from parallel system processes. The system was allowed a 90s reset after each iteration

to ensure balanced resource usage for each benchmarking iteration. Both databases crashed at a thread count of 2000, which is why it will not be included in the comparison charts.

The results obtained were saved in separate text files, which will be presented in the subsequent chapters using clear graphs depicting the differences between a containerized and a native MySQL instance. To complete the comparison, CPU, memory and disk I/O utilization collected via Grafana and Prometheus are also displayed in upcoming chapters.

Detailed configurations and scripts used are presented in the appendices section.

4.2.1 Transaction and Query Analysis

The figures below present a comparison of the benchmarked MySQL instances' performance with queries- and transactions per second under increasing thread numbers (Figure 9 & Figure 10).

Queries per second (QPS) measure how many SQL queries a system can handle per second. This number covers all reads, writes and other SQL commands. It is an important metric, especially with read-heavy applications such as websites and

databases.

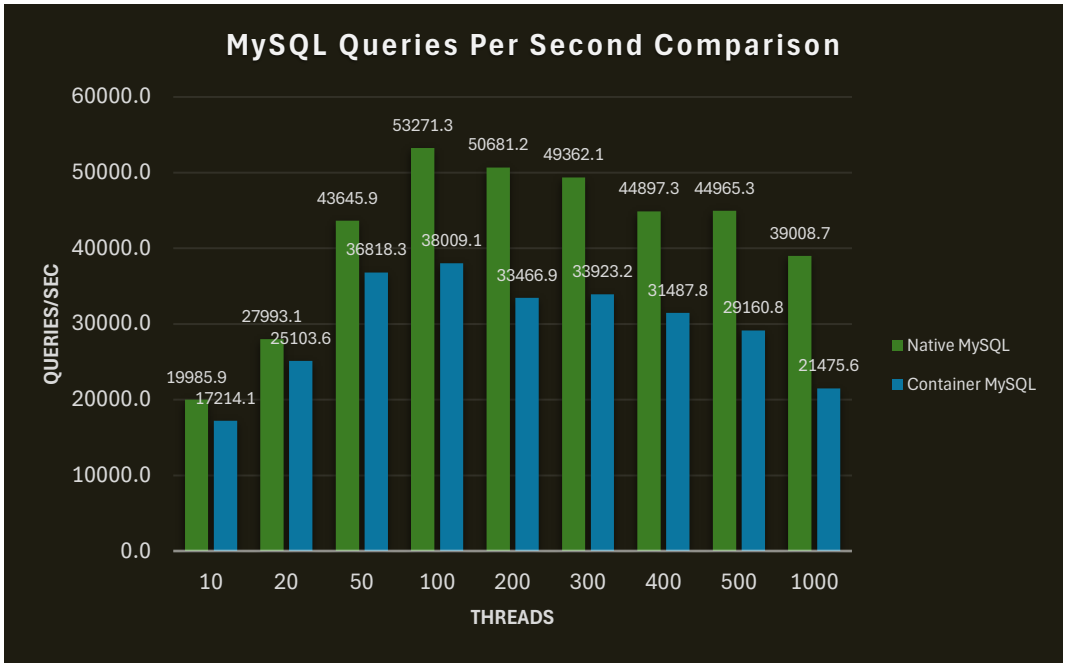

Figure 9. MySQL queries per second comparison.

Transactions per second (TPS) measure how many successful transactions the server can process per second. TPS is an important metric, especially within online shopping and bank applications where databases must process multiple transactions safely and efficiently.

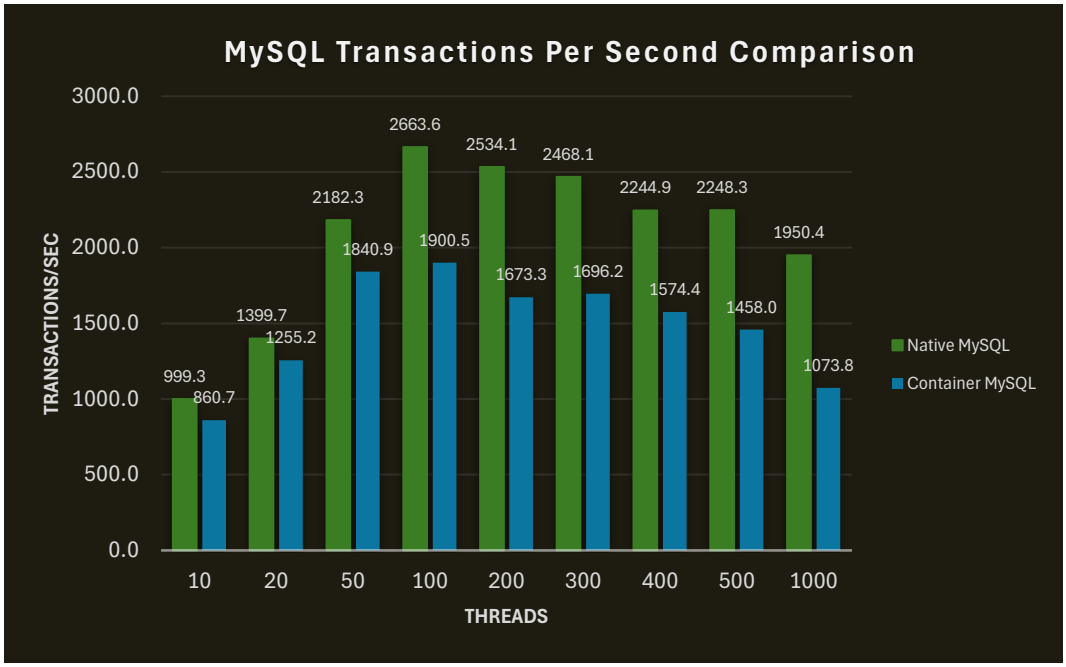

Figure 10. MySQL transactions per second comparison.

The results show that native MySQL reaches higher performance in both metrics on each thread level, proving that native MySQL database performs better especially under heavy load. Each technology reaches its peak performance at approximately 100 threads, after which performance starts slow degradation. Even though additional threads increase the performance up to a certain point, will excessive thread count lead to overload and increased latencies, reflecting to overall performance.

Key findings:

- Native MySQL provides consistently better performance with QPS and TPS.
- Container-based MySQL stays competitive at lower thread count, but scalability is lacking.
- Each technology peaks at around 100 threads, after which begins a slow decline.

4.2.2 Latency Analysis

Latency is a critical metric to measure database operations. This chapter will analyze the average latency and 95th percentile latency, which measures how quickly 95 % of the queries are processed successfully. The latter will provide a more realistic image of the application's latency alongside the average latency, as it considers random latency spikes and gives a good overview of a system's behavior under heavy workload.

The figures below show the average latency (Figure 11) and 95th percentile latency (Figure 12) under increasing thread counts. Both MySQL instances, expectedly, show exponential growth in latency as the thread count increases. However, the gap between technologies massively expands after 500 thread count.

Native MySQL remains consistent throughout the benchmarking, whereas container-based MySQL begin losing its performance after 200 threads with an enormous spike in the latency after 500 threads.

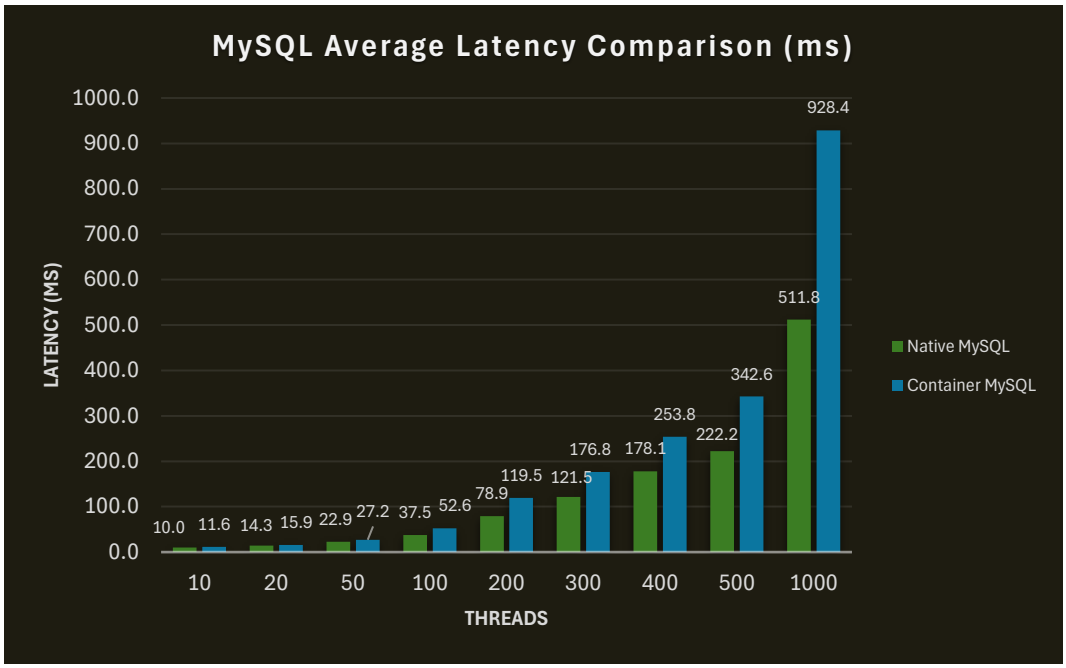

Figure 11. MySQL average latency comparison.

95th percentile latency displays very similar trends as the average latency, however, will emphasize the issues more clearly during higher thread counts. While containerized instance suffers from 1618,8 milliseconds, the native remains below 900 milliseconds.

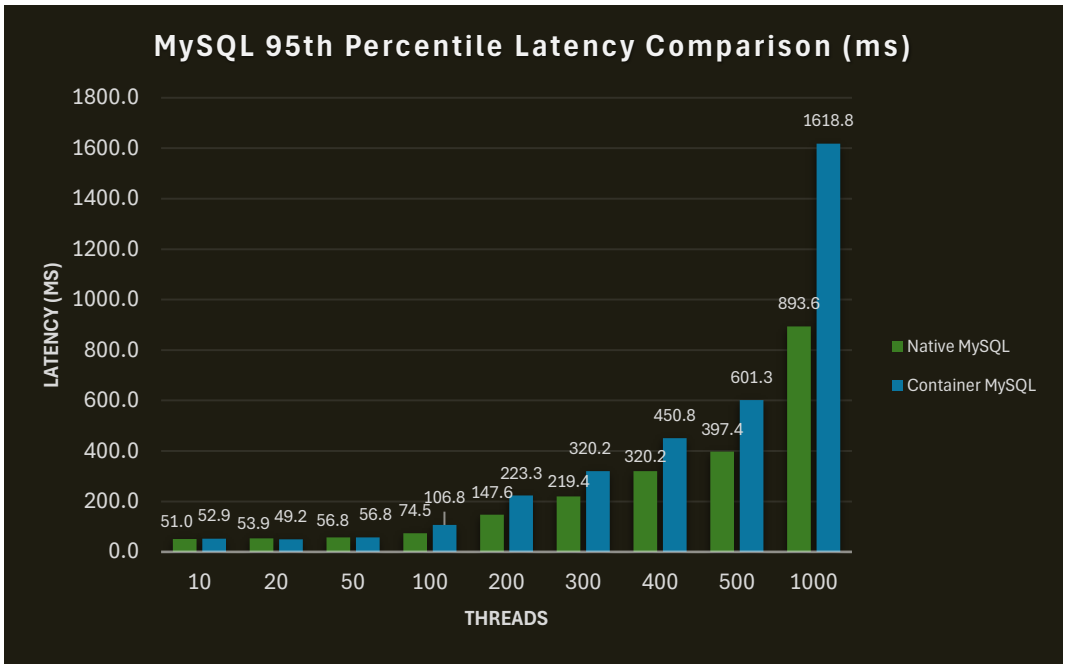

Figure 12. MySQL 95th percentile latency comparison.

Key findings:

- Native MySQL instance's average latency is much more consistent and lower.
- Container-based MySQL instance's latency grows more rapidly as thread count increases.
- 95th percentile latency indicates that container MySQL has weaker control over latency spikes.

4.2.3 CPU and Memory Utilization

Comparing CPU and memory utilization provides crucial information to support earlier performance metrics. Although the number of transactions and latency

provide somewhat comprehensive data about applications' performance, will resource efficiency reveal how effectively different environments can process heavy workloads.

Figure 13 below shows how both MySQL instances increase memory usage almost linearly with growing thread count. As expected, both technologies achieve peak memory usage during the 1000 thread interval. The differences in memory usage are minimal, suggesting that Docker's added virtualization layer does not introduce overhead or affect memory usage heavily.

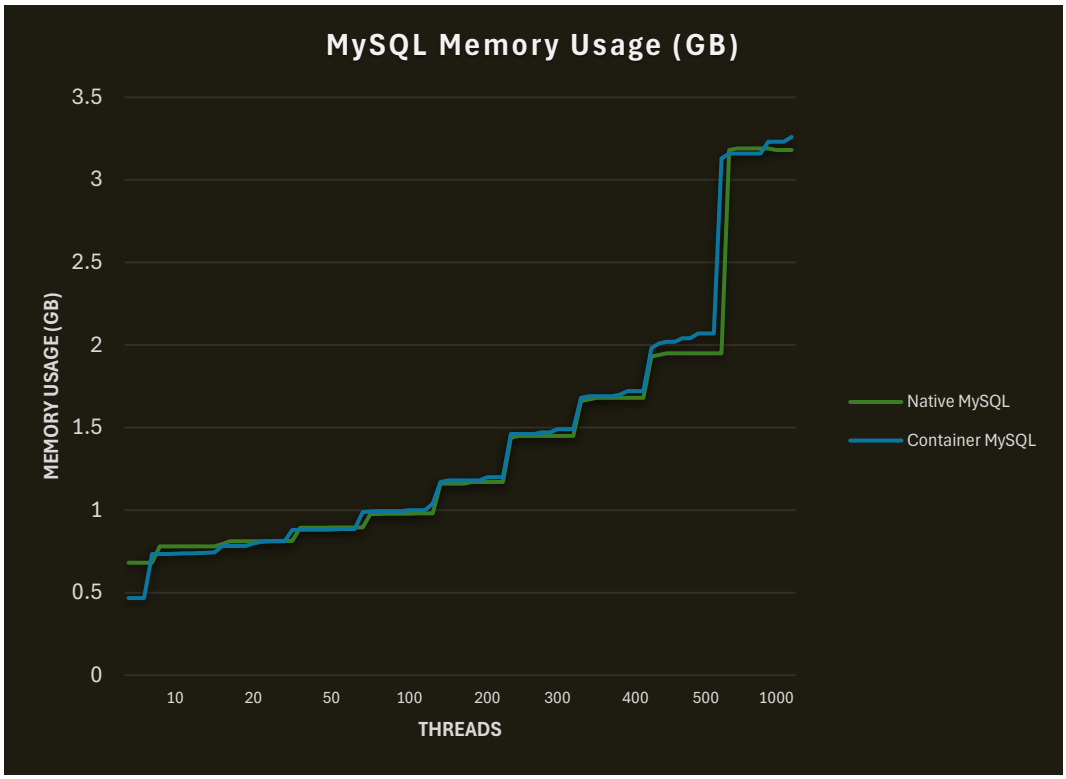

Figure 13. MySQL memory usage comparison.

In figure 14, displaying CPU load, the difference between containerized and native MySQL is much clearer. While native instance reaches the peak CPU utilization at approximately 100 thread counts, the containerized instance reaches it's peak earlier at approximately 50 threads. After peaking, each database seems to consume CPU consistently towards the 1000 thread count mark. This may indicate that each instance deploy resource limitations, or rather hit the resource ceiling.

In addition, both instances react to workload ending rather efficiently, as the CPU load drops back to idling numbers almost immediately between iterations.

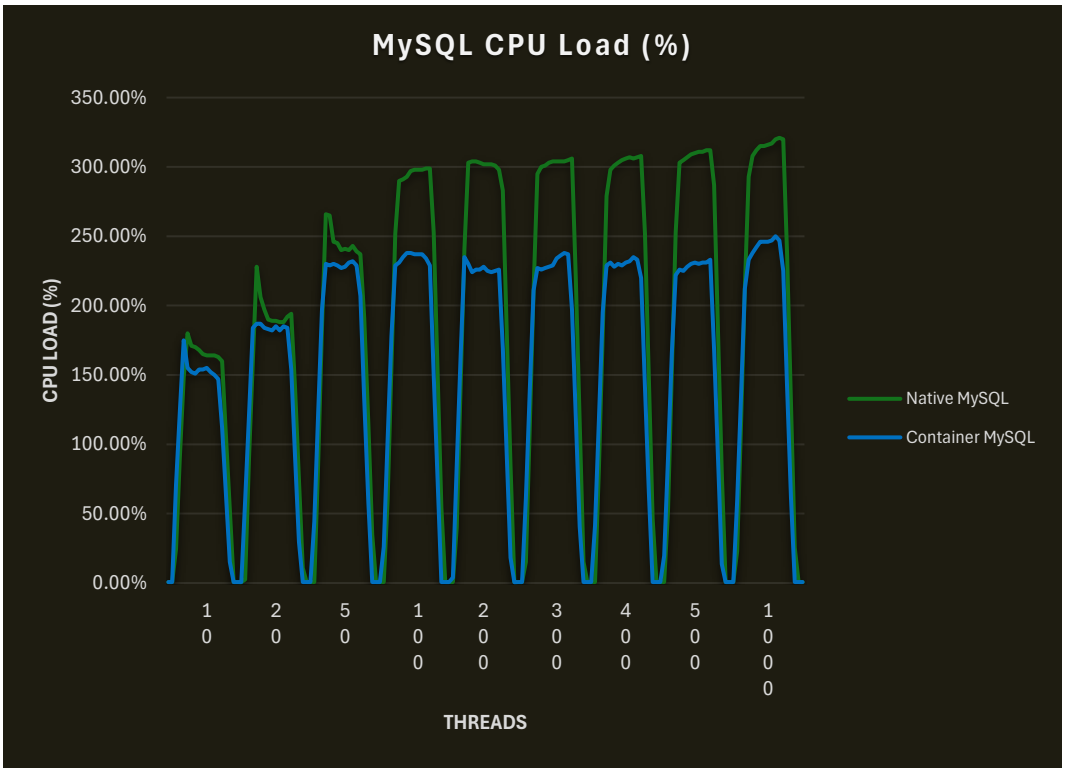

Figure 14. MySQL CPU load comparison.

Key findings:

- Memory usage is almost identical in native and containerized instance.
- Both instances recover rapidly from workloads, suggesting great, stable resource management.
- After peaking, both instances' CPU load remains consistent throughout the benchmark duration.

4.2.4 Disk I/O Analysis

Disk input/output metrics measure how much data MySQL database reads and writes on the disk during the benchmarking process. Among other things, disk write operations include index updates, table writes or logs. Disk read operations,

on the other hand, depict the retrieval of the required data from the disk to execute queries.

The figures below display the disk read and write behavior during the benchmarking under various workloads (Figure 15 and Figure 16). The number of read operations grows fairly linearly, which is expected as higher thread count equals more queries and required data.

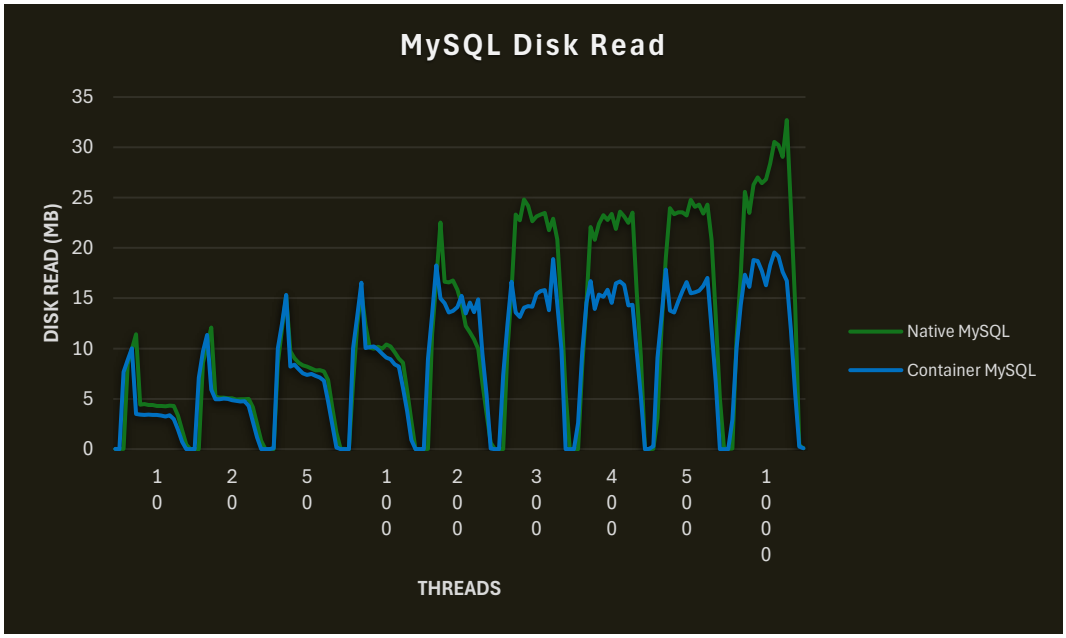

Figure 15. MySQL disk read comparison.

Write operations peak at around 100-200 threads, after which begins a slow decline. This may suggest that the system reaches the resource threshold causing the database's write performance to degrade under heavy load.

Overall, native installation appears to overpower container-based, supporting the observations made in the previous chapters comparing performance differences.

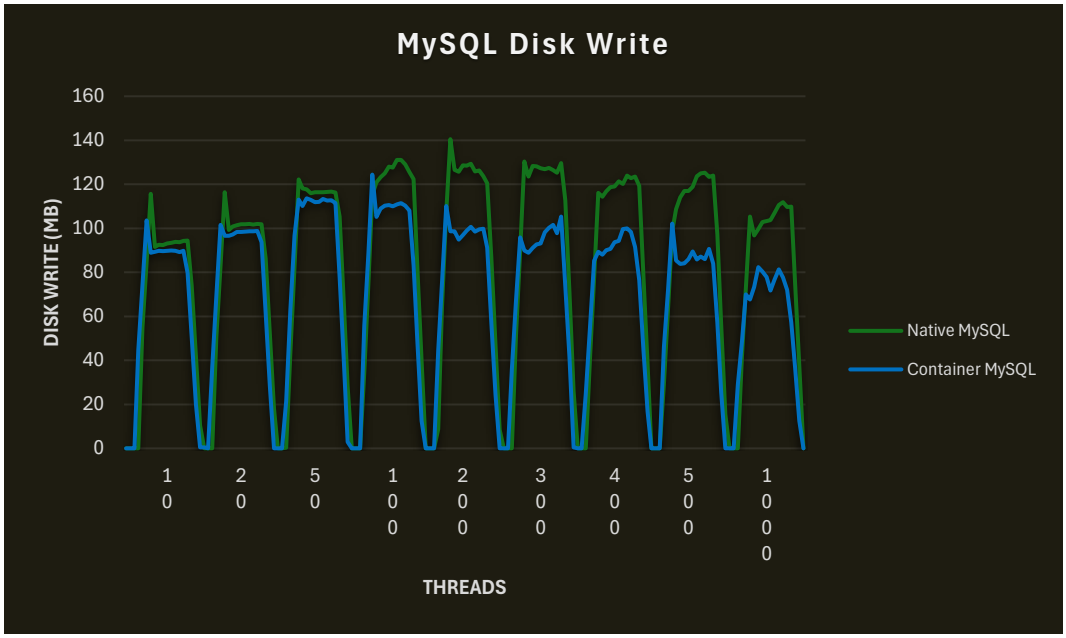

Figure 16. MySQL disk write comparison.

4.3 Redis Performance Analysis

Redis-server instances were benchmarked using a Redis built-in tool called redis-benchmark. The purpose of the test was to compare the behavior of native Redis instance to containerized Redis instance under increasing workloads. Both Redis-servers were configured identically to ensure a balanced and fair comparison. Each instance ran with persistence disabled, meaning that the benchmarking only evaluated in-memory performance and no writing on disk happened.

The testing was conducted using an automated script, which ran redis-benchmark command with increasing concurrency levels. The benchmarking began with concurrency level of 10 and iterated up to 5000 concurrent connections. Each iteration processed a million requests measuring the latency, and the performance of GET and SET operations. After each iteration, the system was allowed a 60s recovery to ensure reliability.

The command for stress testing explained:

```
redis-benchmark -h 127.0.0.1 -p 6379 \
```

```
-n 1000000 \

-c <concurrency> \

-t get,set \

--csv > native${CONN}c.csv
```

- Parameters -h and -p define the Redis-servers address and port.
- Parameter -n defines the total number of requests during an iteration.
- Parameter -c defines the number of concurrent connections, in this case from 10 – 5000.
- Parameter -t defines which Redis commands the test performs, in this case GET and SET.

The results obtained are displayed in the upcoming subchapters with clear graphs, where focus will be on performance metrics such as throughput, latency and CPU and memory usage trends.

4.3.1 Request Throughput Analysis

One crucial performance metric of the Redis-server instances is throughput, which focuses on two separate main commands: GET and SET. These operations reflect on a typical Redis-server use case – retrieving and saving data. Throughput (RPS) measures how many requests Redis-server can process per second with a specific concurrency.

Figures below show that the native implementation achieved higher throughput on almost all concurrency levels compared to container-based implementation (Figure 17 & Figure 18). The gap is most significant on low and medium concurrency levels (10 to 1000), where native instance's throughput beats containerized with virtually tens of thousands of requests per second.

The peak for native technology with SET-operations reaches 185908 requests per second with 50 concurrency, whereas containerized one reaches 155352 requests per second during the first iteration with 10 concurrency.

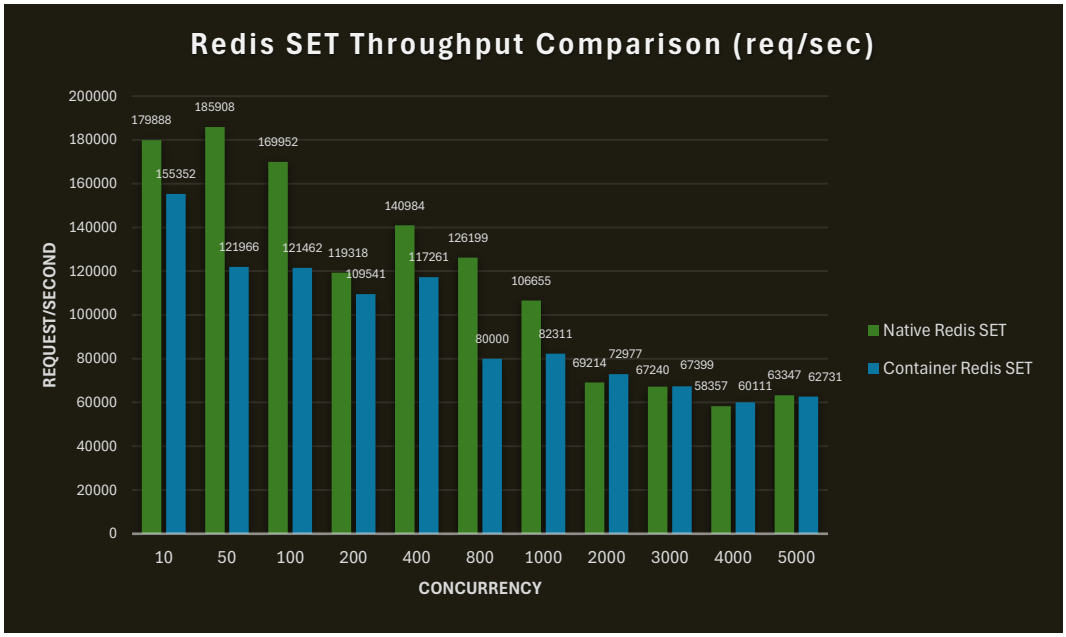

Figure 17. Redis SET throughput comparison.

GET-operations paint a similar picture, where each technology reaches their peak throughput with same concurrencies as SET-operations. Native instance performs a staggering 219250 requests per second, while containerized instance peaks at a moderate 172563 requests per second. It is worth noting that each technologies throughput begins to balance out after 1000 concurrent connections. With SET operations, the containerized Redis-server marginally overpowers the native Redis-server at high concurrencies.

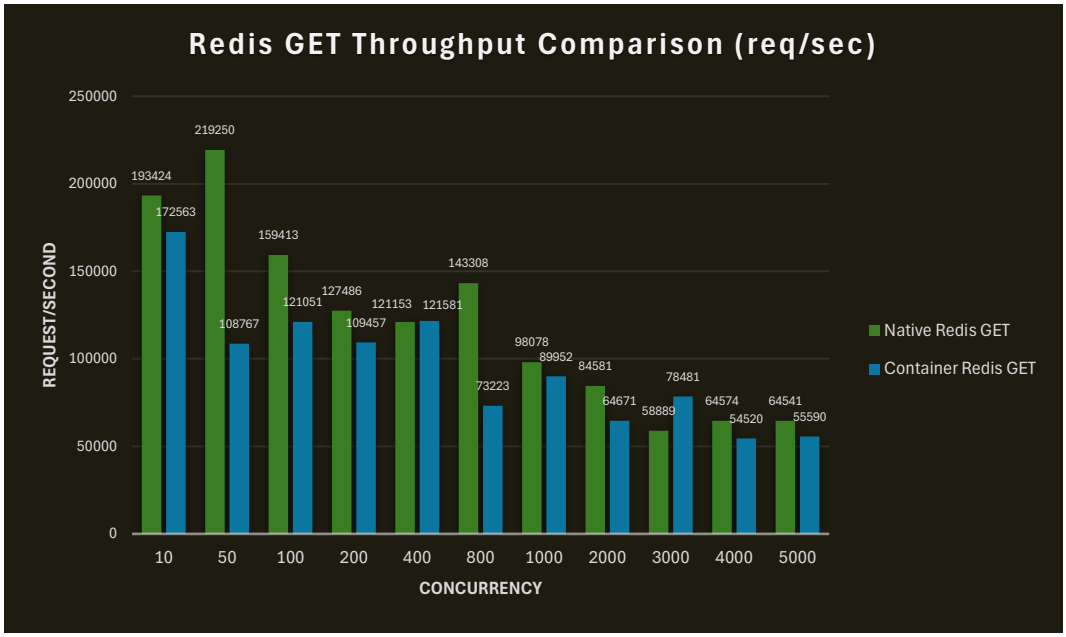

Figure 18. Redis GET throughput comparison.

Key findings:

- The native Redis installation provides better throughput, especially on low and medium concurrencies.
- Containerized Redis installation overcame the native in a few iterations during higher concurrency levels, suggesting good resource management.
- The large gap during early iterations narrows significantly during higher concurrency iterations.

4.3.2 Latency Analysis

One of the metrics that was measured during benchmarking was average latency and 95th percentile latency under increasing workloads. These metrics help assess how responsive the configured Redis instances are, both on average and during load spikes.

Average latency (Figure 19) illustrates the overall responsiveness of the server, whereas 95th percentile latency (Figure 20) tells the latency a user might experience during normal usage. 95th percentile latency means that 95 % of the requests

process faster than the respective number – meaning that only 5 % of the requests are slower.

The figures below depict the difference in latency between a containerized and native Redis-server. Each technology excels during low concurrent connections, but after 400 concurrent connections the gap begins to widen. The average latency rates grow quite linearly in both instances, however, Figure 20 displaying a more realistic latency comparison begins to show more drastic differences between the two Redis instances. While native installation's latency remains relatively balanced in both figures during the same concurrencies, the containerized Redis' latency nearly doubles in the latter figure displaying 95th percentile latency.

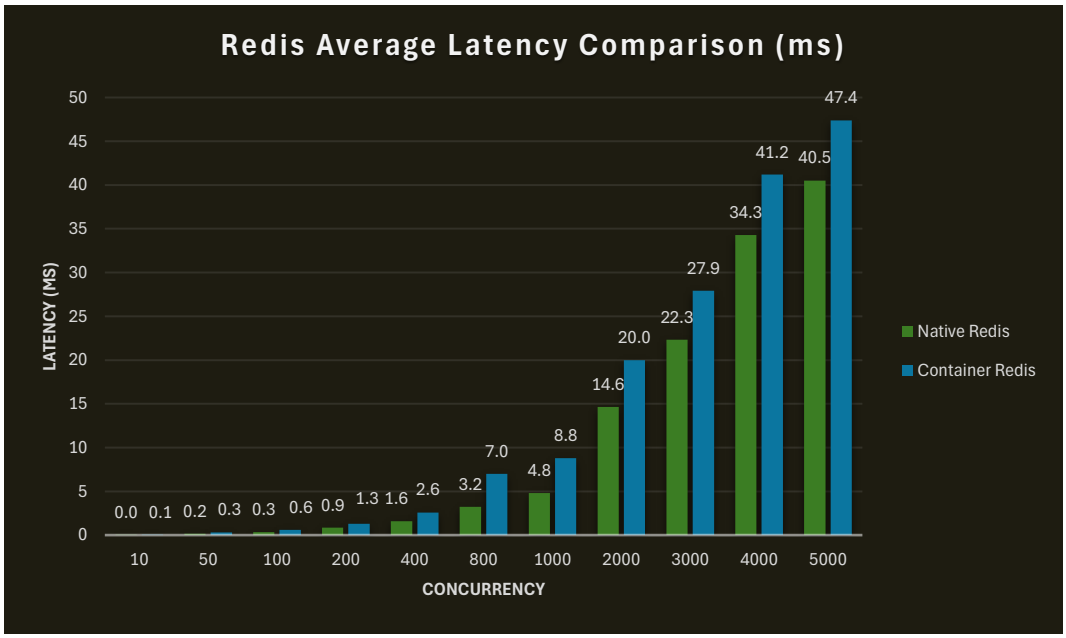

Figure 19. Redis average latency comparison.

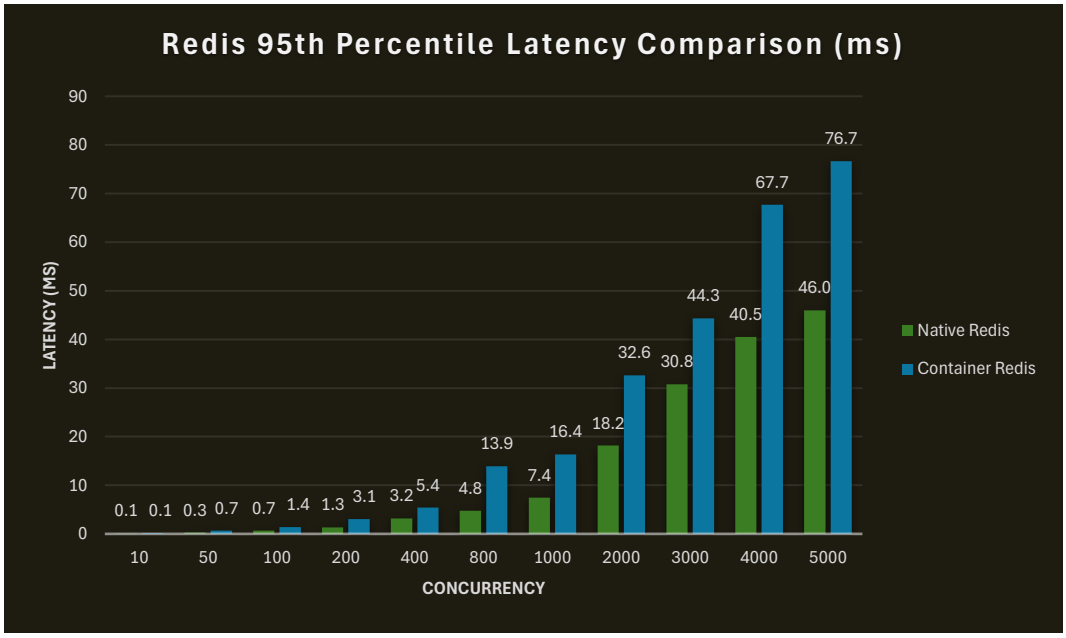

Figure 20. Redis 95th percentile latency comparison.

Key findings:

- During lower workload iterations both instances work fast and in a similar manner.
- Average latency grows quite linearly with native installation gaining the upper hand during each iteration.
- 95th percentile latency paints a more realistic picture of container-based implementations struggles, especially during high workloads.
- The stability of native Redis indicates better scalability and overall performance.

4.3.3 CPU and Memory Utilization

CPU and memory utilization were observed in Grafana throughout the benchmarking process and the figures below portray each metrics trends.

Memory usage portrayed in Figure 21 shows that memory usage was very moderate in each Redis-server instance compared to MySQL databases. Memory consumption in both technologies remained mostly below 50 MB even with higher

concurrent connections. Native Redis instance consumed slightly more memory, but the overall memory usage trends remained level in each compared instance. This confirms Redis' lightweight memory footprint, which is one of its corner-stones within high-performance caching solutions.

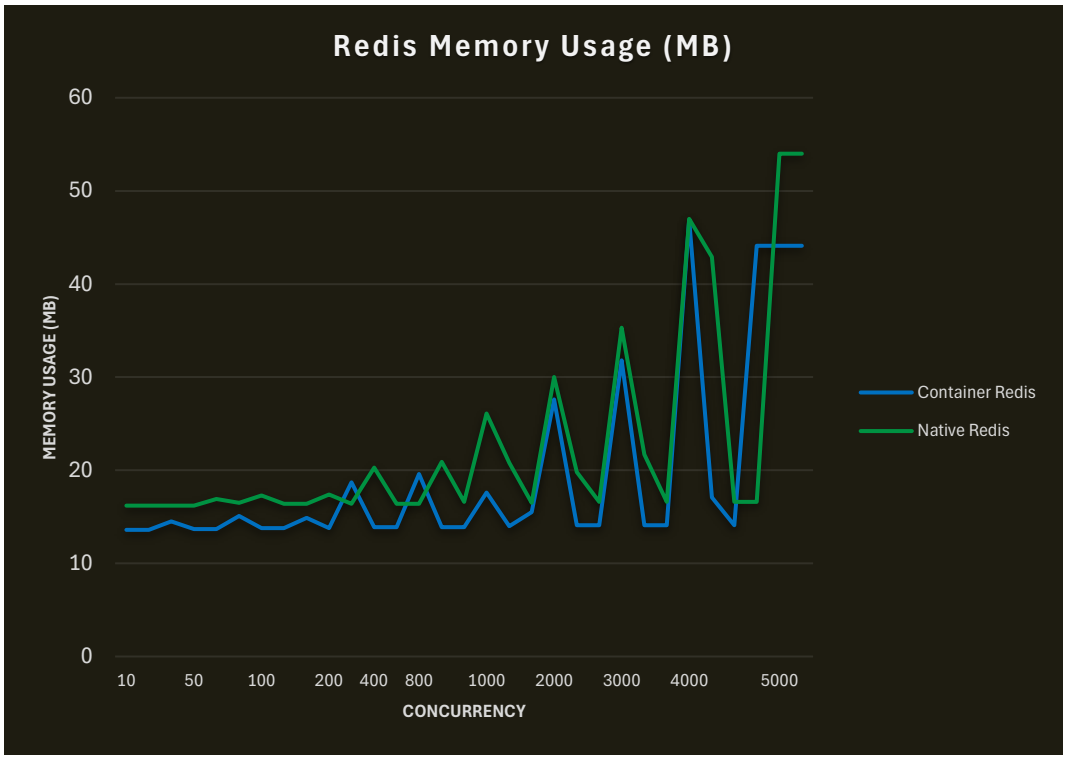

Figure 21. Redis memory usage comparison.

Obtained CPU load, on the other hand, presents more interesting results. As Figure 22 below shows, container-based Redis instance consistently consumed more CPU throughout the benchmarking process. During mid and high concurrency levels, container-based solution strained the CPU close to 20 % more than the native counterpart. This may indicate that Docker's additional virtualization layer might introduce CPU overhead, as these numbers did not translate to collected performance metrics such as throughput.

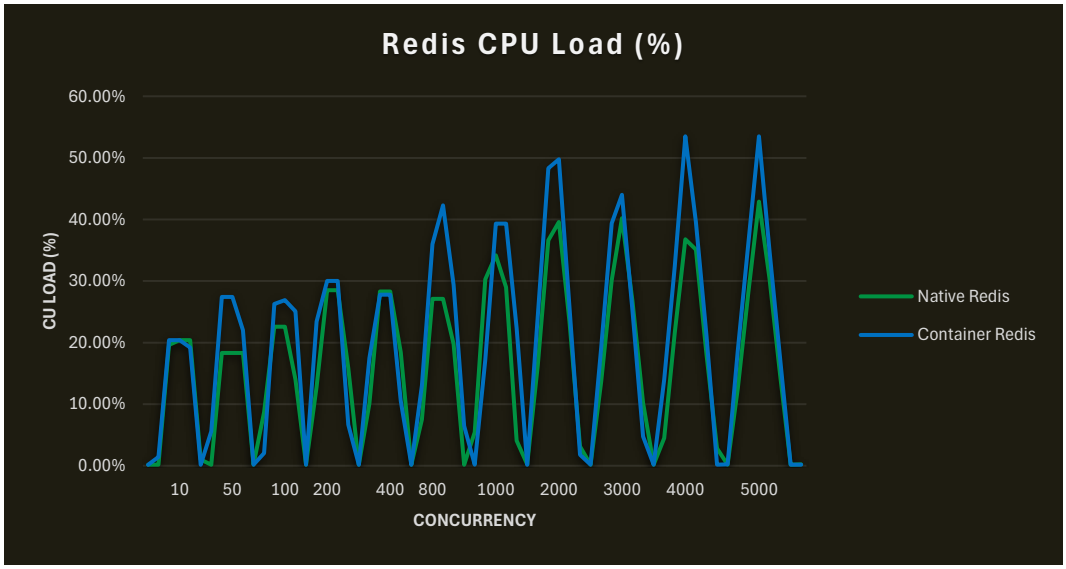

Figure 22. Redis CPU load comparison.

Key findings:

- Memory consumption remained low throughout the benchmarking, reflecting Redis' known lightweight architecture.
- Containerized Redis instance's CPU load was consistently higher than the native one's.
- The higher CPU load did not translate to performance numbers.

4.4 Overall Comparison of Containers and Virtual Machines

This chapter examines the key observations obtained from the entire testing suite regarding native and containerized applications and their differences. The review is based on the test results of three widely adopted applications: Nginx, MySQL and Redis. These applications provided a comprehensive picture of how resource management, performance and scalability differ within the compared technological approaches.

One significant finding concerns resource management. In virtually all occasions, the native installation proved to use more memory compared to container installation, which could partly be caused by platform differences and the fact that

Docker applications share the host operating system's kernel. On the other hand, CPU usage was distinctly higher with containerized Redis, suggesting that some CPU-heavy operations may cause heightened processing power need in container environment.

The results obtained clearly state that native installations provide more performance throughout the workload scale, especially with more resource-intensive applications like MySQL.

Scalability-wise, the native installations dominate during heavy loads, however, container-based installations offer excellent choice under moderate loads. With lower resource consumption, containers scale well horizontally.

In cloud-based environments resource efficiency, ease-of-use and simple management are fundamental characteristics. Containers excel in these areas with easy scalability and efficient management with the help of container orchestration tools, such as Kubernetes.

In summary, containers offer a great compromise with ease-of-use, flexibility and resource efficiency, however, native implementations provide more performance and predictability, especially under heavier workloads. The right choice between these technologies depends largely on the use case, convenience and performance requirements.

4.5 Limitations and Challenges

Several practical limitations and challenges were encountered during this study, which could affect the generalizability and applicability of the results. Although the test environments were to be constructed as balanced as possible, achieving perfect parity and coverage was not possible at all aspects.

Below are the main challenges:

- **The test environment (VPS):** Everything was done utilizing the VPS located in Germany, which introduced some out-of-control challenges and variables, such as resource delegation, possible latency or additional resource overhead.
- **Benchmarking tools:** Tools such as wrk, Sysbench and redis-benchmark measure specifically defined areas, which do not necessarily reflect the complexity of real-life scenarios.
- **Configuration optimizations:** Each respective application was sought to be configured identically, which does not necessarily provide the most dynamic, functional configurations.
- **Broad subject:** Containerization and virtualization are both extremely vast and established technologies, which makes choosing the right applications and benchmarks for maximum applicability rather difficult.

5 CONCLUSION

This chapter summarizes the study's key findings on performance differences between a selection of containerized and virtualized applications and their significance. After the summary, critical reflections on the success and limitations of the study are presented. The chapter concludes with recommendations for future research.

5.1 Summary of Findings

The benchmarking conducted in the study revealed clear differences between native and containerized applications. Most significant observation being that native applications performed better in virtually all tested performance metrics compared to containerized applications. This observation particularly applied in metrics such as throughput and latency, within all the selected applications: Nginx, MySQL and Redis.

Differences in CPU and memory consumption were volatile and highly dependent on the tested application. Overall native applications strained CPU more, whereas containerized applications consumed memory more eagerly and proactively with increasing workloads.

Where native implementations overcame the containerized counterpart's performance-wise, did containerized applications display more predictability with increasing stress levels. This could be a useful characteristic in environments with shared or otherwise limited resources.

Based on the observations and as a conclusion, it can be stated that native implementations are more suitable for situations where high throughput and low latency are critical, while container-based implementations may provide more stable and predictable solutions. It is worth noting that although containers fall short in performance, this may be an acceptable compromise in some environments due to added isolation, transportability and fast deployment capabilities.

5.2 Reflections

Conducting this thesis with all its practical applications and tests taught me tremendously about the features and differences of containerization and virtualization technologies. Although some of the literature and documentation mention the overhead caused by containerizations added virtualization layer, I did not expect the lesser performance of containerized applications to be as notable compared to native applications.

However, during the practical work, I noticed that managing containerized applications was often more straightforward. Thus, I started the application implementations first using containerization and after successful deployment, switched to native installations, where I tried to create the installation as identical as possible.

Benchmarking proved to be a very time-consuming process requiring a lot of patience. Most challenging aspect for me was the choice of benchmarking scenarios and load levels as I wanted to simulate realistic working environments to the best of my ability, which caused a lot of frustration in the form of trial-and-error.

Also, throughout the process it became apparent that raw performance is by far not the most important metric to consider while deciding the application environment. For example, scalability, ease-of-management and simple maintenance might be more significant benefits in many practical applications compared to raw performance.

Overall, this thesis has shaped my perception of the larger picture and understanding of the complexity and situationality of the decision between containerized and virtualized environments.

5.3 Recommendations for Future Research

For future research, I would recommend the utilization of a more flexible test platform to mitigate possible latency and resource limitations that VPS might entail.

In addition, introducing other operating systems and container orchestration tools could be beneficial in obtaining more comprehensive data and results.

One interesting avenue for further research could be the analysis of the impact of network configurations on overall performance and latencies. This could offer a deeper understanding of the benefits that each technology provides on a wider scale.

REFERENCES

Red Hat. (2023, December 13). Containers vs. VMs. *Red Hat.* https://www.redhat.com/en/topics/containers/containers-vs-vms

Golden, B. (2007). *Virtualization for dummies.* Wiley Publishing.

ByteHide. (2023, March 24). Understanding Containerization and Virtualization. *Dev.* https://dev.to/bytehide/understanding-containerization-and-virtualization-3nc3

Shirinbab, S., Lundberg, L., & Casalicchio, E. (2020, September 10). Performance evaluation of containers and virtual machines when running Cassandra workload concurrently. *Concurrency and computation, 32*(17), -n/a. https://doi.org/10.1002/cpe.5693

Gupta, U. (2015, July 28). Comparison between security majors in virtual machine and linux containers. *ArXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1507.07816

Susnjara, S., & Smalley, I. (2024, May 20). What is containerization? *IBM.* https://www.ibm.com/think/topics/containerization

CP/CMS. (2024, April 16). *In Wikipedia.* https://en.wikipedia.org/wiki/CP/CMS

Gartner. (2024, November 19). Gartner Forecasts Worldwide Public Cloud End-User Spending to Total $723 Billion in 2025. *Gartner.* https://www.gartner.com/en/newsroom/press-releases/2024-11-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-total-723-billion-dollars-in-2025

Red Hat. (2021, April 8). What is containerization? *Red Hat.* https://www.redhat.com/en/topics/cloud-native-apps/what-is-containerization

Kong. (2024, March 30). What are Virtual Machines? From On-Premise to Cloud and Beyond. *Kong.* https://konghq.com/blog/learning-center/virtual-machines

Red Hat. (2025, February 28). What is a CI/CD pipeline? *Red Hat* https://www.redhat.com/en/topics/devops/what-cicd-pipeline

Scale Computing. (2024, December 2). Container Virtualization Explained: The Future of Scalable IT Infrastructure. *Scale Computing.* https://www.scale-computing.com/resources/container-virtualization-explained

Ehrman, N. (2024, February 12). Container Runtimes Explained. *Wiz.* https://www.wiz.io/academy/container-runtimes

Powell, P., & Smalley, I. (2024, August 9). What is a container image? *IBM.* https://www.ibm.com/think/topics/container-images

Shamir, J. (2021, April 8). 5 benefits of virtualization. *IBM.* https://www.ibm.com/think/insights/virtualization-benefits

IBM. (2023, March 21). What is virtualization? *IBM.* https://www.ibm.com/think/topics/virtualization

Red Hat. (2024, December 9). What is virtualization? *Red Hat.* https://www.redhat.com/en/topics/virtualization/what-is-virtualization

W3Techs. (2025). Historical trends in the usage statistics of web servers. *W3Techs.* https://w3techs.com/technologies/history_over-view/web_server

Solo.io. (n.d.). Understanding Nginx. *Solo.io.* https://www.solo.io/topics/nginx

Redis. (n.d.). Introduction to Redis. *Redis.* https://redis.io/about/

6 APPENDICES

6.1 APPENDIX A: Nginx configuration files

Containerized Nginx web server's configuration files:

docker-compose.yml – blueprint for the container.

```
version: '3.8'
services:
  webserver:
    image: nginx:1.22.1
    container_name: nginx
    restart: unless-stopped
    network_mode: host
    volumes:
      - ./html:/usr/share/nginx/html
      - ./nginx_conf/nginx.conf:/etc/nginx/conf.d/default.conf
```

nginx.conf – main configuration file

```
user  nginx;
worker_processes  auto;
error_log  /dev/null;
pid        /var/run/nginx.pid;
events {
    worker_connections  4096;
}
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;
    include /etc/nginx/conf.d/*.conf;
}
```

default.conf – site configuration file

```
server {
   listen 80;
   server_name localhost;

   root /usr/share/nginx/html;
   index index.html;

   location / {
      try_files $uri $uri/ =404;
   }

   access_log /var/log/nginx/access.log;
   error_log /var/log/nginx/error.log;
}
```

Native Nginx web server's configuration files:

nginx.conf – main configuration file

```
user  www-data;
worker_processes  auto;
error_log  /dev/null;
pid       /run/nginx.pid;
events {
   worker_connections  4096;
}
http {
   include      /etc/nginx/mime.types;
   default_type  application/octet-stream;
   sendfile       on;
   keepalive_timeout  65;
   include /etc/nginx/conf.d/*.conf;
   include /etc/nginx/sites-enabled/*;
}
```

default.conf – site configuration file

```
server {
    listen 80;
    server_name localhost;

    root /usr/share/nginx/html;
    index index.html;

    location / {
        try_files $uri $uri/ =404;
    }

    access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;
}
```

6.2 APPENDIX B: MySQL configuration files

Containerized MySQL database's configuration files:

docker-compose.yml – blueprint for the container.

```
version: '3.8'
services:
  db:
    image: mysql:8.4.4
    container_name: mysql
    restart: unless-stopped
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: thesis
```

mysql.cnf

```
[mysqld]
host-cache-size=0
skip-name-resolve
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
datadir         = /var/lib/mysql
user=mysql
bind-address= 0.0.0.0
port            = 3306
[client]
socket          = /var/run/mysqld/mysqld.sock
```

Native MySQL database's configuration files:

mysql.cnf

```
[mysqld]
host-cache-size=0
skip-name-resolve
pid-file        = /var/run/mysqld/mysqld.pid
socket          = /var/run/mysqld/mysqld.sock
datadir         = /var/lib/mysql
bind-address= 127.0.0.1
port            = 3306
[client-server]
socket          = /run/mysqld/mysqld.sock
```

6.3 APPENDIX C: Redis configuration files

docker-compose.yml – containerized Redis' blueprint.

```
version: '3.8'
services:
  redis:
    image: redis:7.0.15
    container_name: redis-container
    ports:
      - "6379:6379"
    restart: unless-stopped
    volumes:
      - ./redis.conf:/usr/local/etc/redis/redis.conf
      - ./redis_data:/data
    command: ["redis-server", "/usr/local/etc/redis/redis.conf"]
```

Redis configuration is identical for both containerized and native installations.

```
bind 0.0.0.0
port 6379
protected-mode no
loglevel warning
logfile ""
save ""
appendonly no
tcp-backlog 511
timeout 0
tcp-keepalive 300
daemonize no
supervised no
maxmemory-policy noeviction
dir /data
dbfilename dump.rdb
latency-monitor-threshold 0
slowlog-log-slower-than 10000
slowlog-max-len 128
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes
```

6.4 APPENDIX D: Benchmarking scripts

Nginx benchmarking script:

```
URL="http://89.117.55.131:80"
THREADS=4
DURATION="180s"
CONCURRENCY_LEVELS=(50 100 200 400 800 1200 1600 2000 3000
4000 5000 7500 10000 25000)
for CONC in "${CONCURRENCY_LEVELS[@]}"
do
   echo "Running test with concurrency $CONC"
   wrk -t$THREADS -c$CONC -d$DURATION $URL > ${CONC}.txt
   echo "Test with concurrency $CONC complete. Waiting 2 minutes
before next..."
   sleep 120
done
echo "All tests completed."
```

MySQL benchmarking script:

```
#!/bin/bash
MYSQL_HOST=127.0.0.1
MYSQL_PORT=3306
MYSQL_USER=root
MYSQL_PASSWORD=thesis
MYSQL_DB=sbtest
TABLE_SIZE=100000
TABLES=10
DURATION=180
THREADS=(10 20 50 100 200 300 400 500 1000 2000)
for THREAD in "${THREADS[@]}"
do
   echo "Preparing database for ${THREAD} threads..."
   sysbench /usr/share/sysbench/oltp_read_write.lua \
      --mysql-host=$MYSQL_HOST \
      --mysql-port=$MYSQL_PORT \
      --mysql-user=$MYSQL_USER \
```

```
    --mysql-password=$MYSQL_PASSWORD \
    --mysql-db=$MYSQL_DB \
    --table_size=$TABLE_SIZE \
    --tables=$TABLES \
    --threads=$THREAD prepare
  echo "Running benchmark with ${THREAD} threads..."
  sysbench /usr/share/sysbench/oltp_read_write.lua \
    --mysql-host=$MYSQL_HOST \
    --mysql-port=$MYSQL_PORT \
    --mysql-user=$MYSQL_USER \
    --mysql-password=$MYSQL_PASSWORD \
    --mysql-db=$MYSQL_DB \
    --table_size=$TABLE_SIZE \
    --tables=$TABLES \
    --threads=$THREAD \
    --time=$DURATION run > ${THREAD}t.txt
  echo "Cleaning up after ${THREAD} threads..."
  sysbench /usr/share/sysbench/oltp_read_write.lua \
    --mysql-host=$MYSQL_HOST \
    --mysql-port=$MYSQL_PORT \
    --mysql-user=$MYSQL_USER \
    --mysql-password=$MYSQL_PASSWORD \
    --mysql-db=$MYSQL_DB \
    --table_size=$TABLE_SIZE \
    --tables=$TABLES \
    --threads=$THREAD cleanup
  echo "Sleeping for 90 seconds before next iteration..."
  sleep 90
done
echo "All benchmarks completed."
```

Redis benchmarking script:

```
#!/bin/bash
REDIS_HOST="127.0.0.1"
REDIS_PORT=6379
CONCURRENCY_LEVELS=(10 50 100 200 400 800 1000 2000 3000
4000 5000)
for CONN in "${CONCURRENCY_LEVELS[@]}"; do
   echo "Running Redis benchmark with $CONN concurrent
clients..."
   redis-benchmark -h $REDIS_HOST -p $REDIS_PORT \
      -n 1000000 \
      -c $CONN \
      -t get,set \
      --csv > ${CONN}c.csv
   echo "Sleeping for 60 seconds..."
   sleep 60
done
echo "Benchmarking complete."
```

6.5 APPENDIX E: Full benchmark result tables

Native Nginx web server's full benchmark results.

Connections	Avg Latency (ms)	Requests/sec	Transfer/sec (MB)	Errors
50	0.4	163041	83	0
100	0.6	184911	94	0
200	1.0	198503	101	0
400	1.8	200771	102	0
800	4.0	185038	94	0
1200	6.2	185041	94	0
1600	8.5	181956	92	0
2000	12.1	170185	86	0
3000	20.8	151670	77	307
4000	31.3	150006	76	2979
5000	82.6	147914	75	34797

Container Nginx web server's full benchmark results.

Connections	Avg Latency (ms)	Requests/sec	Transfer/sec (MB)	Errors
50	0.8	68145	35	0
100	1.5	68829	35	0
200	2.8	72558	37	0
400	5.8	69824	36	0
800	11.1	73096	37	0
1200	18.3	69715	36	0
1600	28.5	65297	33	48
2000	31.3	68604	35	417
3000	49.4	62853	32	1711
4000	57.6	67301	34	3353
5000	71.6	63001	32	8305

Native MySQL database's full benchmark results.

Threads	Transactions/sec	Latency avg (ms)	95th Percentile Latency (ms)	Queries/sec
10	999.29	10	51.02	19985.89
20	1399.65	14.29	53.85	27993.08
50	2182.29	22.91	56.84	43645.88
100	2663.57	37.54	74.46	53271.32
200	2534.06	78.9	147.61	50681.17
300	2468.11	121.48	219.36	49362.11
400	2244.87	178.06	320.17	44897.32
500	2248.26	222.2	397.39	44965.29
1000	1950.42	511.75	893.56	39008.71

Container MySQL database's full benchmark results.

Threads	Transactions/sec	Latency avg (ms)	95th Percentile Latency (ms)	Queries/sec
10	860.7	11.62	52.89	17214.07
20	1255.18	15.93	49.21	25103.6
50	1840.92	27.16	56.84	36818.33
100	1900.45	52.61	106.75	38009.08
200	1673.34	119.45	223.34	33466.88
300	1696.16	176.78	320.17	33923.22
400	1574.39	253.76	450.77	31487.78
500	1458.03	342.55	601.29	29160.76
1000	1073.77	928.42	1618.78	21475.57

Native Redis' full benchmark results:

Concurrency	Native SET RPS	Native GET RPS	Native AVG Latency (ms)	Native P95 Latency (ms)
10	179888.47	193423.59	0.038	0.063
50	185908.16	219250.17	0.154	0.271
100	169952.42	159413.36	0.327	0.655
200	119317.51	127485.98	0.857	1.319
400	140984.06	121153.38	1.593	3.207
800	126198.89	143307.53	3.212	4.759
1000	106655.29	98077.68	4.79	7.439
2000	69213.73	84580.9	14.64	18.159
3000	67240.45	58889.34	22.341	30.751
4000	58356.68	64574.45	34.289	40.511
5000	63347.27	64541.11	40.513	45.951

Container Redis' full benchmarking results:

Concurrency	Container SET RPS	Container GET RPS	Container AVG Latency (ms)	Container P95 Latency (ms)
10	155351.88	172562.55	0.053	0.127
50	121966.09	108766.59	0.3	0.671
100	121462.41	121050.73	0.588	1.399
200	109541.02	109457.09	1.304	3.055
400	117260.79	121580.54	2.578	5.407
800	80000	73222.52	7.012	13.887
1000	82311.3	89952.33	8.775	16.359
2000	72976.72	64670.5	19.982	32.623
3000	67399.07	78480.62	27.921	44.319
4000	60110.61	54519.68	41.191	67.711
5000	62731.32	55589.53	47.368	76.671