University of Amsterdam

Masters Thesis

Comparative performance analysis of distributed web applications running in three common virtualization scenarios.

Author:
Daan Vinken

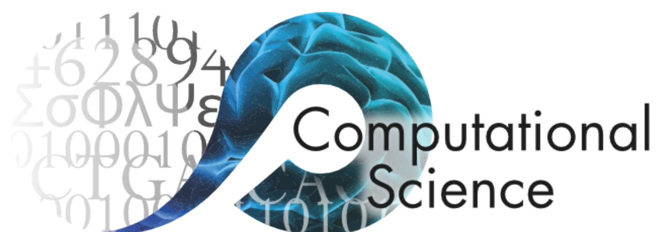
Supervisor:
Adam Belloum

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Computational Science

at

Adyen, Amsterdam

October 2022



Declaration of Authorship

I, Daan Vinken, declare that this thesis, entitled 'Comparative performance analysis of distributed web applications running in three common virtualization scenarios.' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the University of Amsterdam.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 10 October 2022

“All software sucks, be it open-source [or] proprietary. The only question is what can be done with a particular instance of suckage, and that’s where having the source matters.”

Alexander Viro, original author of Linux namespaces

UNIVERSITY OF AMSTERDAM

Abstract

Faculty of Science

Master of Science in Computational Science

Comparative performance analysis of distributed web applications running in three common virtualization scenarios.

by Daan Vinken

An ongoing trend in the software industry is moving web applications to run in the cloud. A common case is deploying (web) applications packaged as a container image, which run regularly on top of a virtual machine. This allows for better scalability, easier deployments, and often improved resource usage efficiency.

Existing research on the performance of virtualized environments often shows negligible performance penalties, as a result of using standardized benchmarks. However, due to the ever-increasing complexity of distributed systems, these often isolated performance measurements merely depict an optimal scenario. The overall performance can be heavily affected by the nature of a distributed system and the complexity of its workload.

This research aims to find the performance differences arising when shifting the web applications of these distributed systems to run inside virtualized environments. A simple open-source microservices application is used to realize a more realistic test setup. As a result, this thesis shows that distributed tracing could offer an accurate way to measure performance penalties on distributed web applications. However, during this research some caveats were acknowledged. This can provide a way for enterprises to gain insight into performance changes while using virtualization layers in their infrastructure. Additionally, it has been proven that different virtualization techniques are more resource intensive compared to non-virtualized environments.

Acknowledgements

I am very grateful for the people being involved in my thesis, providing support and valuable insights. First of all, I'd like to thank my direct supervisors Daan Schipper and Adam Belloum for their guidance and for giving me a chance to take a deeper dive on this subject and do my graduation project. I am not aware of a phase within my studies where I have been able to develop my technical skills more within the given timespan. Additionally, I would like to thank the whole containerization team at Adyen for welcoming me. Last but not least, I wish to extend my special thanks to my girlfriend for the everlasting support and patience, listening to my gibberish on virtualization.

Daan

Contents

Declaration of Authorship **i**

Abstract **iii**

Acknowledgements **iv**

Contents **v**

List of Figures **vii**

List of Tables **ix**

List of Algorithms **x**

Abbreviations **xi**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Scope 2
- 1.3 Problem statement 3
- 1.4 Outline 4

2 Background information **5**
- 2.1 Virtualization techniques 5
  - 2.1.1 Bare-metal 5
  - 2.1.2 Virtual Machines 6
  - 2.1.3 Containers 8
- 2.2 (Distributed) tracing 10

3 Related work **13**
- 3.1 Performance of virtualization 13
- 3.2 Overhead teardown 14
- 3.3 Research context 26
  - 3.3.1 Workload generation 27
  - 3.3.2 Comparing request flows 29

3.3.2.1 Statistical testing 31
3.3.2.2 Kolmogorov-Smirnov test [24] [12] 33
3.3.2.3 Anderson-Darling test [39] [12] 34

4 Methodology 36
4.1 Systems under test 36
4.1.1 Bare-metal 39
4.1.2 Container on bare-metal 40
4.1.3 KVM 41
4.1.4 Container on KVM 41
4.1.5 Request flow breakdown 41
4.2 Test setup 43
4.2.1 Workload generation 44
4.2.2 Data collection 45
4.3 Benchmarks 47
4.3.1 General workflow 47
4.3.2 Distributed tracing 48
4.3.3 System-level metrics 51

5 Experiments and results 54
5.1 Distributed tracing 54
5.1.1 Accuracy measurements 54
5.1.2 Data exploration 59
5.1.3 System benchmark 61
5.2 System-level metrics 72

6 Discussion 76

7 Conclusion and future work 82

A Remaining system traces 86

Bibliography 89

List of Figures

1.1 Overhead in the execution of Spark workloads. This figure depicts the fraction of unaccounted overhead for the production workload. [16] 2

2.1 An illustration of different access levels and how hardware virtualization (e.g. VT-x or SVM) adds an additional ring for the hypervisor. [8] 7
2.2 Virtual Machines vs. Containers 8
2.3 Visualization of where the layer of abstraction lies at for VMs (left) compared to containers (right) [26]. 10
2.4 Example of distributed tracing visualization (Jaeger UI). 11

3.1 Two visualizations from existing works showing the effect of virtualization on CPU performance. 16
3.2 The overhead of containers and Virtual Machines during the memory intensive STREAM benchmark. Showing both values and variability percentages.[21] 18
3.3 Two visualizations of Felter et al. showing disk IO benchmarks for different virtualization techniques. 20
3.4 Storage data throughput and its variability overhead of a Docker container vs. VM [21]. 22
3.6 Overview of a simple distributed system, where the red square indicates the region of interest. 28
3.7 Example plot of a latency distribution [44]. 30

4.1 Graphical representation of our SUT, with servers (purple), applications (green) and databases (red). 37
4.2 The Time Synchronized drift measured on one of the servers. 39
4.3 Overview of the four different (virtualized) environments that are benchmarked within this research. 39
4.4 An overview of the recorded traces of the complete request flow, shown in Jaeger. 42
4.5 An overview of the benchmarking system architecture. 45
4.6 A flowchart of the testflow used for our benchmarks. This is indicative, if numbers vary this will be clearly indicated. 48

5.1 Request flow for payment-service, including the *SleepTask* 55
5.2 The results of the accuracy tests for the larger request flow (*large_payment)*. It shows the p-values for testing the hypothesis whether request flows with a introduced delay in μs come from the same distribution as the baseline (0 μs).The complete trace span measured is the call to the *payment service*. 56

5.3 The results of the accuracy tests for the smaller request flow (*small_payment)*. It shows the p-values for testing the hypothesis whether request flows with a introduced delay in μs come from the same distribution as the baseline (0 μs). The complete trace span measured is the call to the *payment service*. 58
5.4 Example of a typical trace duration distribution plot for one run in a virtual machine and one on bare metal. 59
5.5 A plot representing the same data samples as Figure 5.4, focusing on percentiles. 60
5.6 Two plots depicting how two peaks of sampled data occurences fade into each other when running the same benchmark on virtual machines. 61
5.7 Example of 5 runs for both Bare metal and Containers for the *payment service*. This figure shows the consistency in lower percentiles, but also the variability in higher percentiles. 62
5.8 The test ranges of the test statistic value for the four scenarios. Each run is tested against all other 9 runs in the same catagory, depicting the distance between their CDFs. 63
5.9 The percentile distribution plot of trace span durations for the *order service* call. 65
5.10 The percentile distribution plot of trace span durations for the *SmallFileTask* task. 66
5.11 The percentile distribution plot of trace span durations for the *LargeFileTask.read* task. 67
5.12 The percentile distribution plot of trace span durations for the *LargeFileTask.write* task. 69
5.13 The percentile distribution plot of trace span durations for the *payment service* call. In this scenario the *order service* was excluded and only the bare metal and VM scenarios are considered. 71
5.14 The percentile distribution plot of trace span durations for the *put_user* call. 72

A.1 The percentile distribution plot of trace span durations for the *user service* call (GET). 87
A.2 The percentile distribution plot of trace span durations for the *linpack-benchmark1* task (*user service*). 87
A.3 The percentile distribution plot of trace span durations for the *linpack-benchmark2* task (*payment service*). 88

List of Tables

3.1 Numeric comparison of the research papers on memory performance overhead virtualized environments discussed. . . . 17

3.2 Numeric comparison of the research papers on memory performance in virtualized environments discussed. This figure depicts the percentual overhead. . . . 19

3.3 Overview of relevant overhead from other research on the performance implications of virtualization. . . . 22

3.4 Several examples of web pages with the number of requests made on a single page load, including the chance of being in the upper 1% percentile. 30

3.5 Comparison of well-known statistical tests when comparing distributions . 32

5.1 Overview of percentiles and AD test for the order service span in μs. . . . 64

5.2 Overview of percentiles and AD test for the *SmallFileTask* span in μs. . . 66

5.3 Overview of percentiles and AD test for the *LargeFileTask.read* span in μs. 67

5.4 Overview of percentiles and AD test for the *LargeFileTask.write* task span in μs. . . . 68

5.5 Overview of percentiles and AD test for the payment service span in μs. . 69

5.6 Overview of percentiles and AD test for the (PUT) user service span in μs. 71

5.7 Overview of the results from the system-level metrics analysis, exported with Prometheus node exporter. All results are averaged or aggregated over a 1 minute timespan. . . . 73

A.1 Overview of percentiles and AD test for the *user service* call (GET) in μs. 86

A.2 Overview of percentiles and AD test for the *LinpackBenchmark1* task in μs. . . . 87

A.3 Overview of percentiles and AD test for the *LinpackBenchmark2* task in μs. . . . 88

List of Algorithms

1 The pseudo-code of the algorithm determining the run that best fits all other runs, using the AD-test. 50

Abbreviations

AUFS	**A**dvanced (Multi-layered) **U**nification **F**ile**s**ystem
CDF	**C**umulative **D**istribution **F**unction
ECDF	**E**mpirical **C**umulative **D**istribution **F**unction
ES	**E**lastic**s**earch
ID	**I**dentifier
IO	**I**nput/**O**utput
IT	**I**nformation **T**echnology
KVM	**K**ernel-based **V**irtual **M**achine
LC	**L**inux **C**ontainers
NAT	**N**etwork **A**ddress **T**ranslation
NIC	**N**etwork **I**nterface **C**ard
NTP	**N**etwork **T**ime **P**rotocol
OS	**O**perating **S**ystem
PSP	**P**ayment **S**ystem **P**rovider
SLA	**S**ervice **L**evel **A**greements
SUT	**S**ystem **U**nder **T**est
TCP	**T**ransmission **C**ontrol **P**rotocol
UDP	**U**ser **D**atagram **P**rotocol
VM	**V**irtual **M**achine

Chapter 1

Introduction

1.1 Motivation

An ongoing trend in the software industry is moving enterprise applications to cloud environments, where virtualized environments are the norm. As Gartner states in their annual research report on cloud trends "By 2025, 51% of IT spending in these four categories will have shifted from traditional solutions to the public cloud, compared to 41% in 2022. Almost two-thirds (65.9%) of the spending on application software will be directed toward cloud technologies in 2025, up from 57.7% in 2022." [30]. Organizations often make this choice to reduce operating costs and simplify the architecture of their applications. This transition often results in faster deployment, more efficient resource utilization, and better scalability of their services [2].
With virtualization, generally two technologies are considered. Both containers and virtual machines (VMs) are virtualization techniques. Although containers are often perceived as 'lightweight VMs,' they are two vastly different concepts. Both virtualization techniques benefit from the characteristics named above, making them attractive for software-focused companies. Unfortunately, in most cases, this transition towards virtualized environments eventually results in overhead caused by the layer of abstraction that offers this virtualized environment. This is not necessarily a blocking issue as the advantages can still outweigh these limitations. Although this overhead may, for example, decrease the performance of a single service (e.g. latency), the transition could still improve the overall performance of the application by allowing for better scalability. Naturally, it is important to be aware of any performance implications, specifically for services that require a high-performance application offering low latency and high availability. This thesis is written at Adyen, a Payment System Provider (PSP) and bank conforming exactly to these requirements.

1.2 Scope

Although virtualization has been around for decades, a large increase in adaption can be seen over the past years, with new technologies like Docker and Openstack gaining more attention. Alongside the increased number of appliances, more research has been done on the subject. Specifically, we look at what effect these technologies have on the performance of all kinds of applications, ranging from High Performance Computing (HPC) to web services. Most research studies apply an existing benchmark to a single virtualized instance. These benchmarks execute a high workload close to the environment its resource limits. Although this can already give good insight into the capabilities of different virtualization environments, it merely depicts an optimal scenario [25] [17] [14] [21]. This results in a trade-off between testing against a representative system and testing against a small isolated system to exclude unwanted effects. To resemble a real-life scenario, it is sensible to look beyond a system that works at its maximum capacity. Organizations often tend to run a system at a lower average resource usage, to make sure that spikes do not cause any unforeseen issues. As mentioned above, virtualization is often used in distributed environments, where it offers improved scalability and resiliency. Analysis on Google Cluster Data has already shown around 2016 that 96.2% of the VMs do not operate as a standalone application. On average, a virtual machine cooperates with 19.2 other virtual machines, and then 49.1% of these virtual machines even interact with more than 1000 other virtual machines [23].

Due to the ever-increasing complexity of these distributed systems, the aforementioned isolated performance measurements do not always resemble production environments. The overall performance can be heavily affected by the nature of a distributed system and the complexity of its workload [16]. In Ousterhout et al. [32] for example, the Apache Spark distributed data analytics framework is being benchmarked. As can be seen in Figure 1.1 the unexplained overhead is significant for production workloads, compared to typical benchmarks.

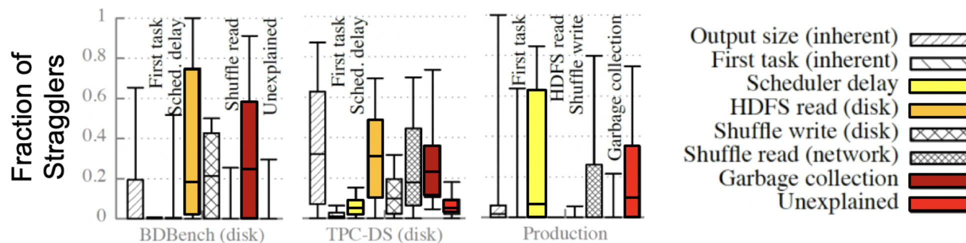


FIGURE 1.1: Overhead in the execution of Spark workloads. This figure depicts the fraction of unaccounted overhead for the production workload. [16]

This phenomenon is quite common. It seems that there is a gap in this field of research, whereas few benchmarks on typical distributed systems are performed in an actual distributed setup, close to a production environment. This research aims to fill this gap by taking into account the common characteristics of a distributed enterprise application.

One common subject regarding virtualization is scalability. The increase in scalability when moving towards virtualized environments is often the main reason for organizations to make this switch. Comparing the scalability of bare metal servers / VMs to containers is sometimes considered inequitable, whereas with the former, possibilities for automated orchestration are very limited. To clarify, this research has no objective of demonstrating the advantages/disadvantages of virtualization. Instead, it is assumed that the choice to move towards containerized web applications is already made, but the performance implications of this are yet to be determined.

1.3 Problem statement

The goal of this research is to find the performance differences that arise when moving the web applications of these distributed systems to run inside containers. Before benchmarking performance, it is important to establish what the correct metrics are to make a fair comparison. The focus will be to let the testing environment and workload be as close to the production environment as is practically feasible. Based on these goals, the following main research questions are listed.

- What is the effect on performance when moving web applications from bare metal servers to containers in a highly distributed system?
- What metrics can be a good indication of how the performance of a distributed system is being affected?

Based on the last question, we introduce one more research question below. We already mention code instrumentation here. In the next chapter we'll provide more context on what this is and how it works.

- Does code instrumentation (tracing) offer a suitable way to measure the performance of distributed web applications?

Before answering these questions, it is important to determine which metrics define good performance for a distributed web application that requires low response time and high availability. As mentioned above, we are looking at metrics seen from the user

perspective, such as meeting Service Level Agreements (SLAs). This means that the focus is mainly on latency, not throughput. With increased scalability, it should be possible to achieve higher throughput more easily.

1.4 Outline

The remaining part of this thesis is outlined as follows. In Section 2, background information is provided on the technologies applied in this research. Subsequently, Section 3 will give a comprehensive review on existing literature. Section 4 will describe the applied methodology and elaborates on the techniques involved. The results of the applied methodology will be discussed and are set out in Section 5. Finally, Sections 6 and 7 will present any conclusions drawn from this research, following a discussion of the results and the applied methodologies.

Chapter 2

Background information

This chapter aims to give a brief background on the different virtualization techniques applied in this research. First of all, it is important to have a clear distinction between the four different environments that are considered for this research. This includes a high-level overview of their working principles. Finally, we present a quick overview of distributed tracing, also known as code instrumentation.

2.1 Virtualization techniques

In the world of computer science, virtualization often refers to the abstraction of some software component into a logical object. In general, there are two types of virtualization, namely hardware-level virtualization (VM), and OS-level virtualization (containers). Now we look at the four different scenarios that will be tested in this research [35].

2.1.1 Bare-metal

In the context of the rest of this research, bare metal will be the baseline. In the broader sense, applications running on bare metal always have all of the server its resources available to themselves. This is limited by other processes running on the same server and also using these resources. Due to the lack of an additional layer of abstraction, such as additional network or input/output (IO) interfaces, this results in applications capable of providing higher network and IO throughput [25]. The extra layer of abstraction introduced by virtualization often causes extra resource usage (e.g. CPU cycles) for the same operation on a bare metal machine.

2.1.2 Virtual Machines

A VM is a virtual operating system that functions as a mimic of a regular OS, having its own resources like CPU, memory, and I/O interfaces. A simple example of a VM is running a Windows OS on an Apple Mac OSX system. However, there are many more use cases, especially in cloud and edge systems. In the latter case, virtual machines are often running Linux. In those cases, the underlying OS could be running another, or perhaps even the exact same, distribution of Linux. Running cloud and edge applications on VMs offers numerous of advantages amongst platform independence, isolation, and resource abstraction. Furthermore, the use of virtualization, in general, allows for better support for (elastic) scalability [43].

The core component of this type of virtualization is the hypervisor. This layer of abstraction manages the hardware and separates the physical resources from the virtual environments. Resources can be assigned to a VM according to preset values. When a process within the guest OS requires more resources from the host machine, the hypervisor issues a request to the host to gain a larger partition of the shared pool of resources. There are two types of hypervisors:

- **Type-1 hypervisor (bare metal)**
  A type-1 hypervisor runs on bare metal, without a host OS as a layer in between. In fact, the host OS itself becomes the hypervisor. This is common practice for production-like workloads/performance. A well-known example of a Type-1 hypervisor are Kernel-based Virtual Machines (KVMs).
- **Type-2 hypervisor (hosted)**
  Type-2 hypervisors run on top of the host OS. This type of hypervisor is often used for development purposes or individual usage. This type of hypervisor introduces additional latency for basically any operation, albeit very little these days. All operations of the virtual machines and hypervisor have to pass through the host OS. Well-known examples are VMware Fusion and VirtualBox.

To investigate the performance implications for the ‘thinnest’ layer of abstraction, this research does not consider type-2 hypervisors, corresponding to production environments with the lowest virtualization overhead.

Intel and AMD, being the biggest CPU manufacturers in the market, realized around 2005 that the major challenges of virtualization were full virtualization. Both due to the performance overhead and due to the complexity arising when designing and maintaining

virtualization solutions. In response, Intel and AMD independently created new processor extensions of their respective CPU architectures, called Virtualization Technology (VT-X) and Secure Virtual Machines (SVM) respectively. These extensions allow the hypervisor to run a guest OS that is expected to run with kernel privileges. Nowadays, other CPU vendors also support these technologies under different names.

Hardware-assisted virtualization not only proposes new instruction sets, but also introduces a new privileged access level, called ring -1. The hypervisor can now run at the newly introduced privilege level, ring -1, meaning that the guest OS can run on ring 0. This setup is also shown in Figure 2.1.

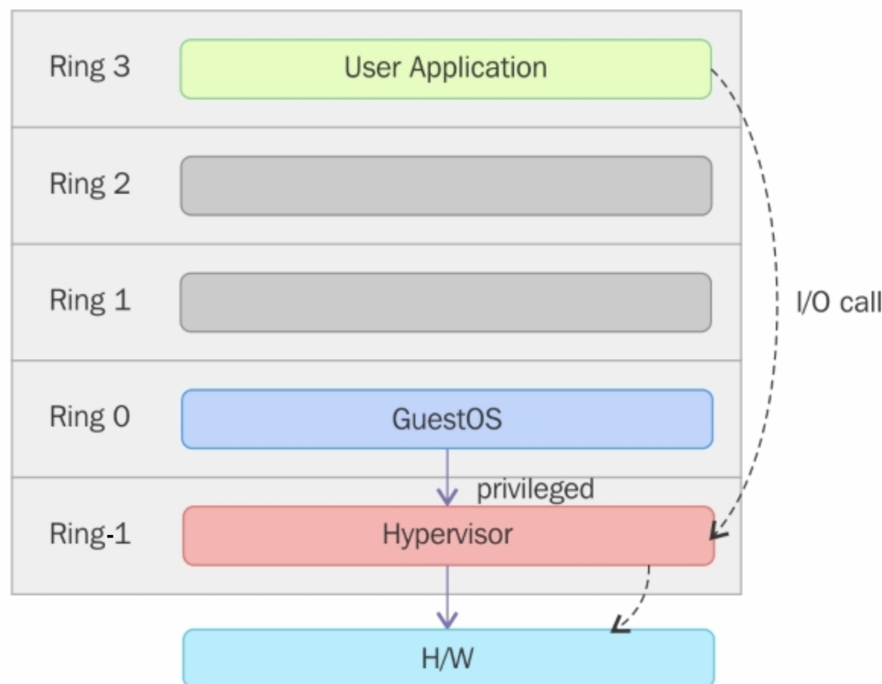


FIGURE 2.1: An illustration of different access levels and how hardware virtualization (e.g. VT-x or SVM) adds an additional ring for the hypervisor. [8]

Furthermore, with hardware virtualization, the hypervisor is relaxed and needs fewer CPU cycles compared to type-2 hypervisors, which reduces performance overhead. Simply put, the hardware assisted virtualization provides the support to have a hypervisor running on the kernel and avoid isolation with the guest operating system. This improves performance and reduces the complexity of running VMs.

Kernel-based Virtual Machine

Kernel-based Virtual Machine (KVM) is an open source hypervisor technology has been built into the mainline Linux distribution since 2007. KVMs allow users to turn Linux into a hypervisor that enables one or more VMs to run directly on the kernel. Every VM is implemented as a regular Linux process, scheduled by the standard Linux scheduler,

with dedicated hardware like a network interface, GPU, CPU, memory, and disks. KVMs are known to be one of the fastest implementations of virtual machines, as proved by multiple research papers [34] [20] [1]. Naturally, there is no winner-take-all, as for some operations other hypervisors. like Xen, still outperform KVMs.

2.1.3 Containers

Containers are, simply put, a way to group processes and manage them together as isolated environments. This means that compared to VMs there is no additional OS layer or hypervisor needed. This difference is also depicted in Figure 2.2.

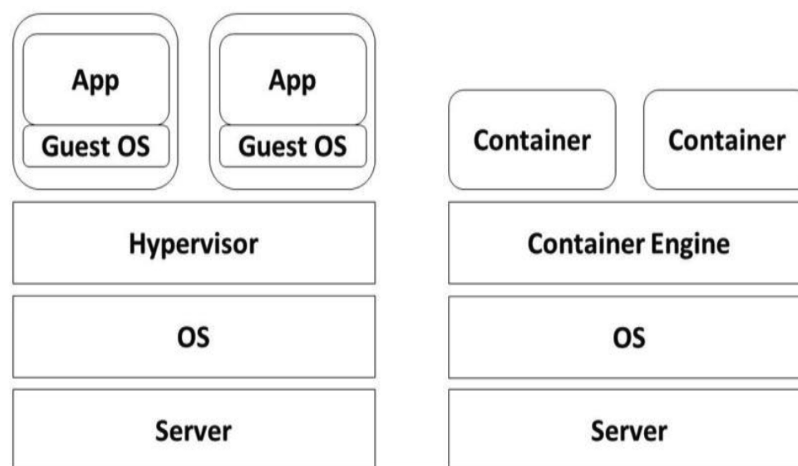


FIGURE 2.2: Virtual Machines vs. Containers

Containers share underlying features of the operating system, such as the kernel, but are otherwise separated from other containers. In this way, isolated environments can thus be created sharing the resources of the host machine, without the overhead of a hypervisor. Containers do not mimic a hardware layer via a hypervisor. In essence, any process in Linux can be considered a container, having its boundaries set by its namespaces and control groups.

Namespaces and control groups (cgroups) are fundamental aspects of how Linux systems operate. Namespaces are an abstraction where a global system resource makes it appear to the process within the namespace that they have their own isolated instance of that global resource [1]. A straightforward example is an Ubuntu container image where the filesystem is abstracted by the *mnt* (mount) namespace. From a perspective within the container, it seems as there is a full filesystem for a complete Ubuntu OS, with no higher layer in the filesystem (root level). However, this is because the container filesystem itself

[1] https://man7.org/linux/man-pages/man7/namespaces.7.html

is limited by the *mnt* namespace set on the host OS. In fact, control groups themselves are also a namespace. *cgroups* are a mechanism for partitioning/aggregating different processes, and all child processes, into hierarchical groups specific to the operation. Simply said, namespaces limit what can be seen from within a namespace, e.g. other process ids. *cgroups* limit how much resources you can use, for example, memory or CPU.

Container runtime & APIs

Nowadays, containers are being used as a tool rather than just under the hood of a UNIX system. The adaptation of enterprise applications is immense. IT research company Gartner claimed in 2020 "by 2022, more than 75% of global organizations will be running containerized applications in production, up from less than 30% today." [29]. This trend seems to become reality. At the time of writing, Docker is a very well-known and the most widely used tool. The latter is confirmed by a survey conducted by Enlyft [2], where among 50,000 companies using containers, 97.77 % uses Docker.

In the current software era, there are often two components at the foundation of using containers, the container runtime and the container engine. The engine is the user-facing piece of software. It is responsible for a wide range of activities like user input, API, pulling images from a registry, or managing meta-data. Examples are Docker or LXC (Linux Containers). Commonly, the user-facing process does not run the containers themselves, but are dependent on the container runtime. The latter has different, more low-level, responsibilities including setting up namespaces, consuming the meta-data, or communicating with the system kernel. Docker and many other container engines rely on the container runtime *runc*. The architecture of the Docker ecosystem has changed frequently in the past. For Docker specifically, there is a layer in between called *containerd* which runs as a daemon. It is mainly responsible for managing and running the containers (via *runc*), pulling/pushing images, and managing IO.

On a high-level distinction, there are two types of containers currently used as a tool in practice, OS containers and application containers. OS containers are virtual environments that share the kernel of the host OS but provide isolation of the user space. This type is more similar to a lightweight VM, as it contains most libraries and tools that come with common Linux distributions. Examples of OS containers are LXC and Solaris. On the contrary, there are application containers. While the former are meant to run multiple processes and services, application containers are designed to run a single service. An example is a container that runs a single Java application and, therefore, contains the JDK distribution. Most Docker containers are application containers.

[2]https://enlyft.com/tech/products/docker

User-space vs kernel-space

An important aspect of Linux processes / containers is the interaction between user space and kernel space. The former refers to all operations that live outside of the kernel. This is often higher-level code, such as applications written in C or Java. Consider an example where a Java application is running in a (Docker) container. The code and runtime live in the user space. In a broader sense, all applications function by modifying the data. However, these data are commonly stored in memory and on disk. These data basically live in the kernel space and are accessed by system calls. A common example of a system call is `fopen()` , which associates a stream with a file based on its path.

In the context of this study, when comparing VMs with containers, it is important to understand where the abstraction layer is located. This seperation is depicted by Figure 2.3.

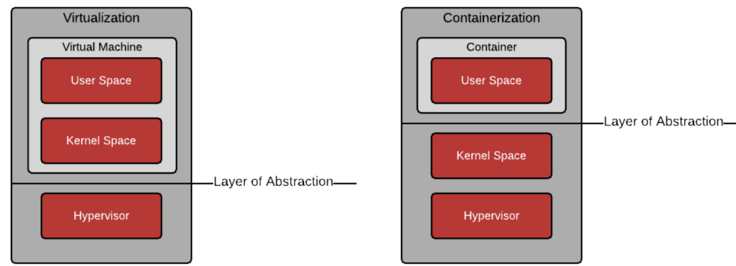


Figure 2.3: Visualization of where the layer of abstraction lies at for VMs (left) compared to containers (right) [26].

From this visualization it becomes clear that, in contrast to containers, with VMs, the user space and kernel space lie in the same layer of abstraction.

2.2 (Distributed) tracing

Tracing is a type of correlated logging that helps to gain visibility into the operation of a system, often for performance profiling or debugging. It can provide insight into what exactly a particular individual service is doing as part of the whole system. In the context of distributed systems, new challenges arise when tracing an application. Most distributed systems have components that scale independently, whereas it is common for redundant services to run on different servers, even geographically distributed. Most production-grade distributed systems are very heterogeneous, which makes instrumentation challenging. Distributed tracing techniques can solve these problems, allowing for proper structure in the insights of your distributed application [33].

Distributed tracing works by the so-called *spans*. Take for examle an HTTP request received by an application. At this point, the operation is assigned a unique trace identifier (ID)e. Every subsequent operation that is performed in the context of this request, called a *child span*, is tagged with the trace ID of that first request, plus its own ID and the parent span ID. Every span comes with its own metadata, which includes (but is not limited to) the service name, IP address, logs/events of the operation itself, configurable tags of the operation, and stacktraces/error messages.

Commonly, these traces are stored in a database, with a buffer inbetween (e.g. Kafka), allowing high throughput for all incoming events (traces). Subsequently, these traces can be queried and visualized with dedicated tracing tools such as Jaeger (Figure 2.4). Another way to use these traces is by applying some kind of anomaly detection, NewRelic is a well-known tool for performing these kinds of analyses.

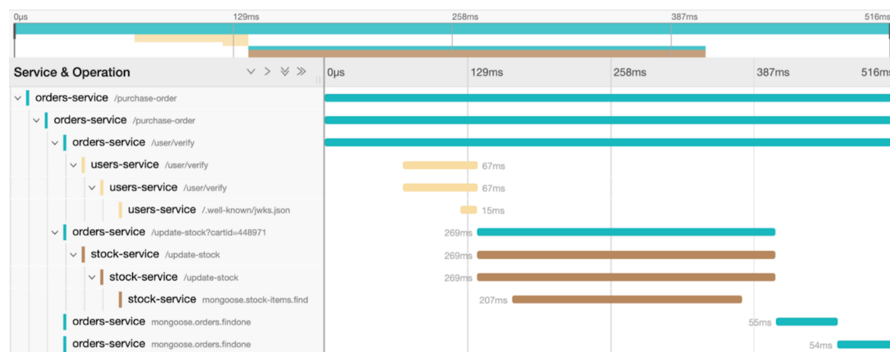


FIGURE 2.4: Example of distributed tracing visualization (Jaeger UI).

One thing to keep in mind when using Opentelemetry distributed tracing is the overhead that comes with it. Several investigations indicate that distributed tracing can incur significant overhead [37][13]. Sampling is often used in such a way that only a segment of the request is recorded, generally around 1%. Sambasivan et al. [38] tries to compare the overhead of different tracing frameworks. However, their results contain mostly surveys of different tracing frameworks/techniques, no actual experiments were applied to those systems. They express some drawbacks in their conclusion, without presenting any numerical results. Sigelman et al. (2010) [41] presents Google its DAPPER, which is a distributed tracing framework. The paper presents some overhead measurements of their original paper. They show that with proper sampling (1/1024), overhead is negligible. When sampling every request (1/1) they show a decrease of approximately 16 % in latency and a decrease in throughput of only 1.6 %. Note that this study is already 12 years old at the time of writing. Opentelemetry code instrumentation has gained a lot of contributions and acceptance in the industry. We see these measurements as a worst-case scenario, and the currently available tools might result in significantly less overhead. Unfortunately, we have not found any more recent studies on this.

Although it could be interesting for the industry, this research also does not consider the overhead of distributed tracing. A bare metal baseline is assumed, where each other environment has the same tracing configuration. Therefore, no significant differences in overhead are expected amongst virtualization layers that could affect performance comparisons of the underlying system. With this assumption, we will check that we are not exhausting any resources such as CPU or networking by the tracing framework.

It is of interest to evaluate whether distributed tracing offers a way to measure the performance of distributed web applications. In doing so, it is important to know the accuracy of these traces. Specifically, because the tracing can be very fine-grained. In principle, every method defined in the source code of the application could be traced. However, in the case of recording every log statement, the duration of data collection on the span might take longer than the actual log statement. This means that traceability should be added to your application while carefully taking best practices into account. Information on the accuracy of span durations is very scarce. This might be simply because that is not the main use case of this technique, where end-users might care more about the code paths than the actual durations. Another good practice is to set up time synchronization between servers with distributed tracing.

Chapter 3

Related work

In this chapter, the existing literature on virtualization techniques, distributed systems and benchmarking will be discussed. The goal of this chapter is to review existing literature, as well as to support upcoming decisions and design choices while working towards answering the earlier stated research questions.

3.1 Performance of virtualization

This section outlines the different approaches and results of the literature which focus on performance comparisons of different virtualization techniques. Starting with a high-level overview of limitations that virtualization can have, different aspects of the introduced overhead are discussed in more detail.

In the past two decades, virtualization has gained significant attention in the software industry and is now being applied in almost any organization its software stack. It started off with commercial use on IBM mainframes, and was reinvented by VMware in the late 90s. Around 2000, Xen and KVMs have been introduced to the public [15]. Virtualization comes with several benefits, amongst energy/cost reduction, faster and more flexible deployment strategies, and improved scalability. Unfortunately, there are also caveats when running applications in a virtualized environment. The added layer of abstraction introduced commonly causes virtualization overhead. This can affect the choice of moving applications towards virtualized environments. Specifically, if the application is focused on stable and high performance (e.g. low latency), organizations might refuse switching to virtualized environment.

McDougall et al. [27] describes the perspective back in 2009 and the challenges ahead with respect to virtualization, with a focus on hardware virtualization (VMs). They state

that the most common way to reason about virtualization overhead is by looking at how much extra resources the application consumes when running in a virtual environment. This can be CPU/Memory usage, but also IO operations, having longer instruction paths to run additional layers of the stack for IO virtualization. Less obvious are concepts like cache misses or extra cycles for the Translation Lookaside Buffer (TLB). This can happen because of arising capacity conflicts caused by unaccounted for instructions due to the virtualization layer[27]. Additionally OS interactions can add overhead on these resources, namely system calls made between the user space and kernel space. This type of overhead is often considered insignificant compared to the overhead in VMs.

Next to VMs, these OS interactions are specifically interesting for containers, where CPU/memory overhead is often considered negligible [25] [15]. Most research does not highlight the minor differences between containers and bare-metal / VMs, which can still be of interest for many organizations. Pointing out the differences between VMs and bare metal could be considered more self-evident because the hypervisor often introduces more CPU cycles for the same operations.

For this research, the focus lies on type-1 hypervisors (bare metal), because this is the most obvious choice when aiming for the best performance. A type-2 hypervisor (hosted) can be seen as a thicker layer of abstraction, implying more overhead. Different researches investigate the performance differences between available type-1 hypervisors like Xen, KVM and VMware Vsphere. The main conclusion that emerges from this existing research is that there is no single outperforming hypervisor. However, Xen and KVMs seem to have the least overhead compared to others. In a more recent work by Algarni et al. [1] it was pointed out that Xen is more suitable for intensive IO workloads, while KVMs perform better for CPU / memory throughput and caching behaviour. In the end, performance of different hypervisors depends on factors like the application's nature and host architecture, and no winner takes all. Therefore, the hypervisor has to be chosen carefully and taken into account when comparing it with other virtualization techniques [34] [20] [1].

3.2 Overhead teardown

A high-level overview of the different performance issues that come with virtualization has been presented. This paragraph will further elaborate on the overhead mentioned above for both containers and virtual machines. In that way, we work towards a hypothesis for the upcoming benchmarks and allow ourselves to reason about any possible optimizations.

Historically, common hypervisors struggled to provide proper baseline speeds for IO and networking due to indirect IO paths that, for instance, sent every packet through the host OS its user space. This has led to considerable research on complex acceleration technologies such as hypervisor bypass and improved polling techniques.
In this context, VMs have come a long way to achieve acceptable performance, but are still suboptimal. On the contrary, containers already started off with near-native performance. Most of the upcoming related work being discussed here ranges from the 2013 - 2017 period. Most likely, new improvements have been made to the performance of both containers and VMs during and after this period. Therefore, some performance implications might be outdated, but there is a lack of more recent work that performs these benchmarks. This also implies the need for a new updated performance comparison of different virtualization techniques.

Bare metal machines are assumed as the baseline for performance, where virtualization overhead is not present. It is unexpected that any form of virtualization would outperform a bare-metal execution environment, which naturally does not have any virtualization layers. In the upcoming subsections, different system metrics will be discussed to create an overview of the available information on virtualization overhead. We try to present overviews of different papers and their results in a schema and/or table. However, often this does not result in a one-to-one comparison due to different testing environments. Therefore, to avoid bias, the overviews will have a quantative focus by avoiding exact numbers when suitable.

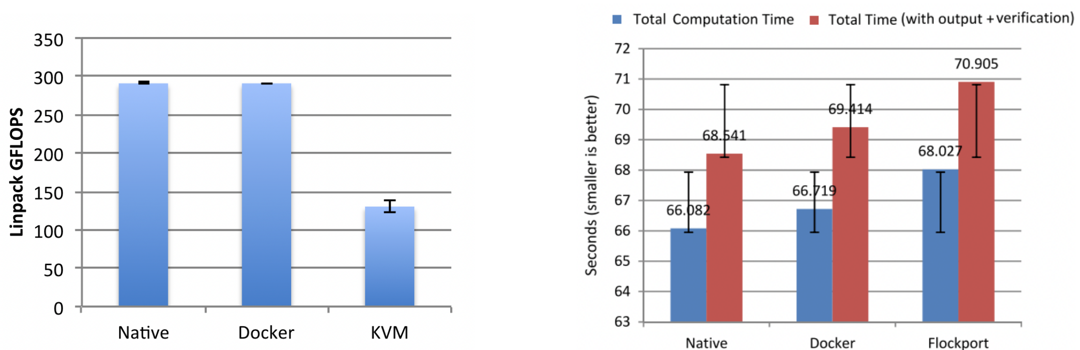
CPU

Mazaheri et al. (2016) [25] uses the Cloud Tester Benchmark Suite. The benchmark we discuss here has regular memory access and focuses on the floating-point operations capabilities of the CPU. The general observation made here is that the virtual machine (KVM) introduces some overhead, although it may be only 7.93% less GFlops. With a KVM, the kernel acts as a hypervisor, providing the minimal abstract layer between the hardware and the guest OS. For containers, no CPU overhead is introduced. It is intuitive to reason about this, as a container is simply a process running on a UNIX system, bounded by its namespaces. The only CPU-specific overhead could be the extra CPU load from the container runtime, e.g. Docker.

Felter et al. (2015) [14] also emphasizes this performance overhead in their full IBM research report [15]. They perform CPU/memory benchmarks with the LINPACK library. During these benchmarks, most of the time is also spent performing mathematical floating-point operations. The paper itself presents a performance comparison between the execution environments Linux (bare metal), Docker (container), and kernel-based virtual machines (KVM). The result of this benchmark can be seen in Figure 3.1a. First

of all, it shows that the number of Floating Point Operations per second (FLOPs) decreases by more than 50% in a virtual machine. Additionally, a slightly higher variability is observed with the black error bars.

In IBM its published article, they state that containers and VMs impose almost no overhead on CPU and memory usage [14]. Contrarily, in their full research report they point out that the KVM performs remarkably worse than the other environments when no CPU specific optimizations take place. This shows the costs of abstracting away the host its system information from the execution environment (VM). As the guest OS is unable to detect the exact nature of the system, the CPU employs more generic instruction sets, with fewer CPU-specific performance optimizations.



(A) Comparison of Gflops during Linpack benchmark in three different execution environments [15].

(B) Comparison of execution time during the Y-cruncher benchmark for Docker and Flockport vs bare-metal [19].

FIGURE 3.1: Two visualizations from existing works showing the effect of virtualization on CPU performance.

The latter is also pointed out in their research, whereas configurability is the weakness of virtual machines. CPU performance is manually optimized by configuration of large pages, CPU model, vCPU pinning, and cache topology. KVMs do not run without Hardware-Assisted Virtualization, which is available in common AMD / Intel processors since 2005 and gained a lot of attention in the past decades. This allows the hypervisor to run in root mode, often allowing better compatibility with the host's CPU [11].

For a benchmark like LINPACK, performing the same operations over and over again, it would indeed be practically feasible to come close to bare metal performance when specific optimizations are added. However, it is questionable whether this extent of performance optimizations is feasible for real-life applications performing far more complex, unpredictable, sequences of operations. Additionally, a KVM might not be exposed to the underlying CPU architecture, for example, not being aware of its memory or cache architecture. However, exposing these types of underlying features of the host machine to the guest OS decreases the portability and maintainability of an application. The

latter properties are often the reason for enterprises of moving towards a virtualized environment and may therefore not be applied in real-life production environments.

For containers, this analogy does not hold. A trend can be noticed where, in the above-mentioned research, most threefold (bare metal, container, VM) performance researches do not focus on the performance implications of containers. Although considered insignificant, they often show some effect on performance.

Kozhirbayev et al. (2016) [19] performs different benchmarks to measure CPU performance. The CPU intensive benchmarks Y-cruncher and LINPACK were selected, showing some decrease in performance for containers compared to bare metal. For the former number crunching benchmark, the differences are very small, as shownin Figure 3.1b. The total computation time increases by 1% for Docker and 3% for LXC. Linpack measures throughput and results in approximately 413 MFlops as a baseline. Docker shows a decrease in throughput of 2.5 % while LXC results in a decrease of only 0.6 %. This implies that, although it can be considered insignificant compared to virtual machines, these operations are affected by the layer of abstraction of a container. Nevertheless, this might not necessarily be related to CPU operations, but could also have been caused by disk I/O affecting the benchmark by longer IO waits.

Source	**Container**	**VMs**	**Remark**
Mazaheri et al. (2016) [25]	±0%	8 %	Matrix-matrix multiplications and fourier transformations.
Felter et al. (2015) [14]	±0%	31 %	Floating-point operations with LINPACK.
Kozhirbayev et al. (2016) [19]	- 3 %	N.A.	Does only take containers into account.

Table 3.1: Numeric comparison of the research papers on memory performance overhead virtualized environments discussed.

Altogether, it seems that containers impose no significant overhead on CPU bounded workloads. However, VMs do suffer from extra CPU cycles, as the hypervisor needs to forward operations to the underlying host. It should be taken into account that measurements are never completely isolated to a single metric, such as CPU usage. Other metrics, like memory and disk IO performance, can affect these measurements. Finally, existing research shows that host-specific optimizations can increase performance. However, this decreases the maintainability and portability of how VMs are used nowadays. Therefore, those performance optimizations can negate the purpose of using the VMs. We suspect this is also depicted by the overhead percentage shown in Table 3.1. Reading both studies, it looks like the bare metal case within Felter et al. was able to use more CPU specific optimizations on bare metal, resulting in a higher overhead compared to Mazaheri et al.

Memory

When measuring the performance of Random Access Memory (RAM), the focus often lies on the throughput that could be achieved. Looking at read/write latency might not be representative for memory performance. This performance is hampered due to involvement of factors like caching, pre-fetching, and simply performance, which are unrelated to memory. An example of this is the overhead of thread execution schedulers. It is important to note that there is often a significant difference in achieved throughput when having random memory access versus sequential memory access. This is mostly related to the way memory is accessed, such as the way page tables are read within virtualized environments. More on this follows later in this section.

In Felter et al. [14] (2016), memory performance is being investigated with the STREAM[1] benchmark. This is a simple synthetic benchmark that performs simple vector calculations while measuring sustainable memory throughput. These operations have regular patterns, meaning sequential access, and thus allow hardware prefetches to detect those patterns and correctly prefetch data. These measurements are again performed on Linux, Docker, and KVMs, resulting in negligible differences with a maximum difference in the median of 1.4% between the three environments. This is considered too small to assume any performance implication, taking noise / variance into account.

Li et al. [21] (2017) also presents performance measurements using the STREAM benchmark comparing KVMs versus containers, exploiting a lower throughput for KVMs up to 5 %. The complete result for the memory throughput of Li et al. is shown in Figure 3.2, depicting both the throughput overhead and its variability.

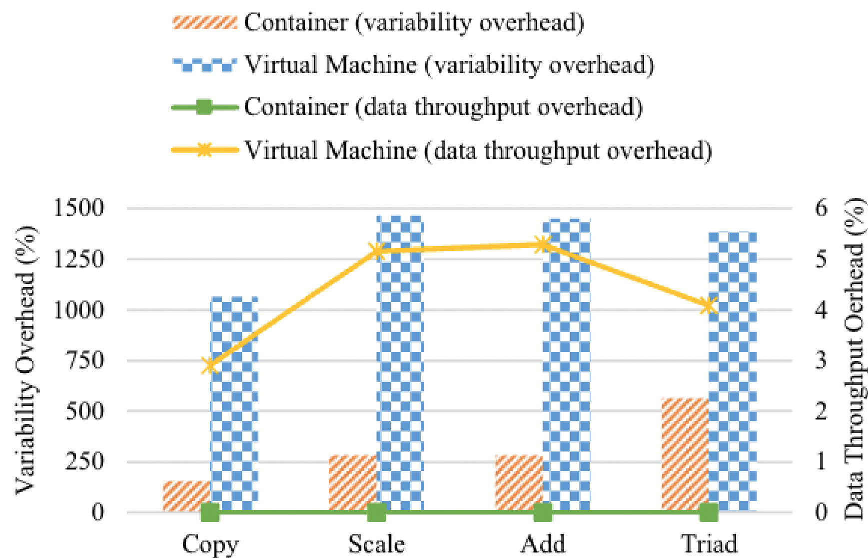


Figure 3.2: The overhead of containers and Virtual Machines during the memory intensive STREAM benchmark. Showing both values and variability percentages.[21]

[1] https://www.cs.virginia.edu/stream/

Mazahari et al. [25] supports these findings using the same benchmarking techniques, with a more significant decrease in memory throughput of 15 % for KVMs compared to bare metal. In all cases, no virtualization overhead was found for containers with respect to memory throughput.

In the same paper, also with STREAM, random memory access has been benchmarked. In this benchmark type, random 8-byte words are read from memory, modified, and written back. In this case, the memory locations are randomly generated, to ensure there is no dependency between successive operations. The results again show a similar performance; however, containerized and bare metal execution outperform KVMs by 13 %. In the case of random memory access on a multi-core CPU, each access typically results in a miss for both the core caches and the translation lookaside buffer (TLB). The latter is a table managed by the OS that functions as a memory address cache. These introduced latencies cannot be hidden, and therefore performance is dependent on the latency of the hardware page table walks and the effective latency of loading main memory. This explains the slower performance of KVMs for those cases, as it performs two hardware page table walks for both the host and the guest system. Mazahari et al. also points out that memory throughput for KVMs is heavily affected by performing random access operations, resulting in a decrease of 14 % of the measured Giga updates per second.

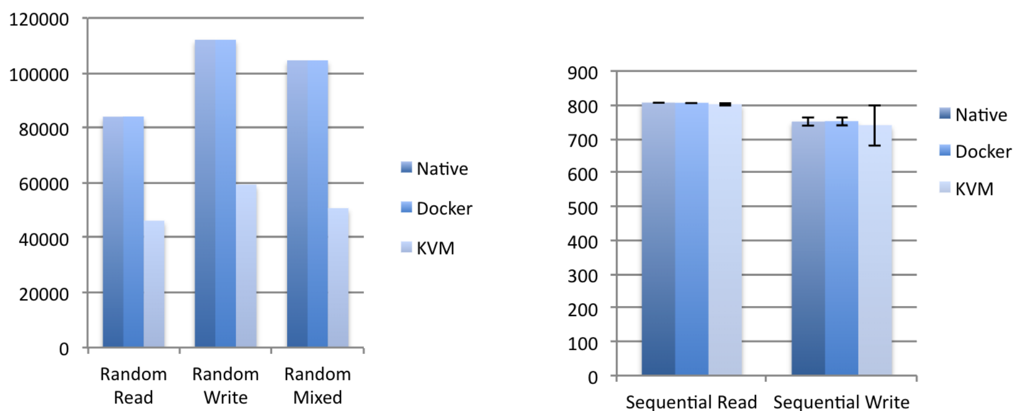
Source	**Sequential access**		**Random access**		**Remark**
-	**Container**	**VMs**	**Container**	**VMs**	
Mazaheri et al. (2016) [25]	±0%	15 %	±0%	14 %	Points out variability overhead of 250 % to 500 % (VMs).
Li et al. (2017) [21]	±0%	5 %	±0%	15 %	
Felter et al. (2015) [14]	±0%	±0%	±0%	5 %	Benchmark is again heavily optimized for VMs.
Kozhirbayev et al. (2016) [19]	4 %	NA	NA	NA	Docker was slightly better than Linux containers.

Table 3.2: Numeric comparison of the research papers on memory performance in virtualized environments discussed. This figure depicts the percentual overhead.

From Table 3.2 it can be concluded that KVMs suffer from some virtualization overhead during memory operations. Containers do not show these issues when also focusing on KVMs, or are at least not being discussed. The following note is made on containers by Mazahari et al. [25]. For sequential memory access, the results showed a variability overhead for containers of 250 to 500 %. In other words, the memory performance loss incurred by both virtualization techniques is mainly embodied with the increase in performance variability. Kozhirbayev et al. [19] draws the same conclusion, focusing only on containers versus bare metal. They state that no differences were found in the memory performance of containers versus bare metal. However, in the figure of their STREAM benchmark, it is visible that containers perform slightly worse regarding their throughput, but not more than a few percent.

Disk IO

I/O performance can be affected more easily by virtualization as discussed in Chapter 2, where it is known to introduce quite significant overhead for specific cases. In Mazahari et al. [25] the IOR (Interleaved or Random) benchmark is performed in the three different environments. IOR is an IO benchmark with parallel capabilities that is commonly used to test the performance of parallel storage systems using a variety of interfaces and access patterns. Slowdown/speedup from both Random IO operations per second (IOPS) and bandwidth are presented in its results, where a few findings stand out. First, in relation to virtual machines, it shows that reads can have up to 30 % slowdown for both bandwidth and latency for KVMs. For write operations, the virtualization overhead with KVMs becomes less, but still varies from 3 % to 7 % depending on the specific write pattern. The lower read latency for KVMs is supported by Felter et al. [15]. For random read throughput of IO, shown in Figure 3.3a, it decreases by approximately 50 % for VMs. This is because all IO goes through QEMU (the hardware emulator), resulting in a significant decrease in performance. Furthermore, there is a clear distinction between blocks/bytes and read/writes. Translating this to the context of real life workloads, for instance logging, writing small byte-ranges to a file, could be severely affected by those virtualization techniques.



(A) Comparison of random access IO throughput for containers (Docker) and KVMs [15]. Shown in operation per second (IOPS).

(B) Comparison of sequential access IO throughput for containers (Docker) and KVMs [15]. Shown in MB/s.

Figure 3.3: Two visualizations of Felter et al. showing disk IO benchmarks for different virtualization techniques.

For containers, Mazahari et al. also presents some performance decrease, albeit less significant. Again, depending on the access pattern, read operations are up to 4 % slower for VMs. Furthermore, for MPI parallel write IO, Docker shows a slowdown of approximately 10 %. However, for random IOPS containers shows a speed-up of parallel direct IO operations of 3.3 %. This is at least remarkable, as theoretically bare metal would be the baseline with the highest performance. Therefore, the reading slowdown mentioned

above of the same magnitude of 4% for VMs could also be considered inaccurate, as any speedup is unexpected when an extra layer of abstraction is added by virtualization. However, Li et al. [21] supports these measurements, as shown in Figure 3.4. They present a data throughput overhead of approximately 5% for block data reads and more than 50% overhead for byte-sized disk reads.

Regarding containers, Felter et al. [14] did not observe any difference in performance between Docker and bare metal for IO throughput. Note that in this case, a volume is mounted as a Docker volume, bypassing the special layered file system that comes with Docker by default. Bhimani et al. [5] confirms this by stating that in the case where Docker volumes are being used, IO operations through that path are independent of the choice of storage driver, and should be able to operate at the IO capabilities of the host. This is also confirmed by Felter et al. as shown in Figure 3.3a and 3.3b.

Nevertheless, this performance penalty due to the filesystem is specific to Docker and not necessarily containers. In Kozhirbayev et al. [19], the focus lies on IO throughput for containers only, where a decrease in throughput of approximately 30% for Docker is pointed out. For LXC containers (Flockport), a decrease of approximately 11% is observed. For Docker this measurement does not give us a fair comparison, as in this case Docker's own multi-layered unification file system (AUFS) was used. Still, for LXC containers the same filesystem as bare metal is used, implying that there are actual performance implications even for throughput.

Li et al. [21] performs IO performance tests with Bonnie++[2], while focusing on reading and writing, and on byte and block size data. A clear observation is that for block read & writes containers impose no significant overhead. This shows the difference between using Docker its own overlay filesystem, as used in Li et al., and mounting volumes for IO like in Felter et al. However, both reading and writing byte-size data is approximately 45% slower compared to bare metal performance. This is shown in their Figure 3.4.

[2]https://linux.die.net/man/8/bonnie++

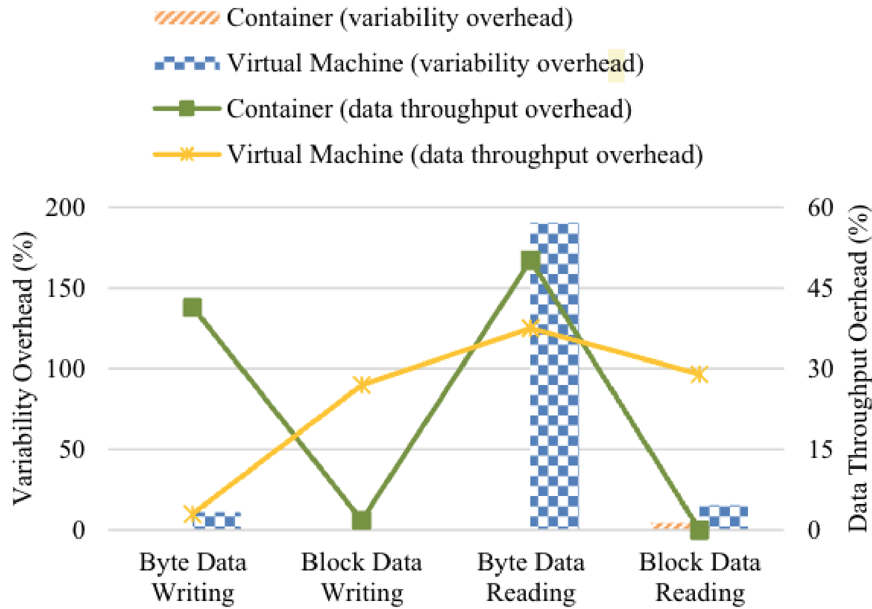


FIGURE 3.4: Storage data throughput and its variability overhead of a Docker container vs. VM [21].

Now again this is due to Docker its own file system, which comes with a well-known performance implication. However, it is remarkable that byte-sized reads/writes are so significantly worse for containers that even VMs outperform them by almost 40 %. This still implies a clear distinction between blocks/bytes and read/writes, where further clarification is desired. Again, consider an application that regularly logs small entries to one or more files. In this case, running the application in a container could cause severe performance issues. These performance issues would not arise during a block-sized IO benchmark while measuring throughput.

	Read		Write		
Source	**Container**	**Virtual Machines**	**Container**	**Virtual Machines**	**Remark**
Li et al. (2017) [21] byte-size & block-size resp.	±42%	5 %	±0%	3%	Byte size significantly slower for containers. However this uses Docker's overlay filesystem.
	0 %	30 %	2 %	±27%	
Felter et al. (2015) [14]	±0%	±55%	±0%	55 %	Percentage is for random reads/writes. Sequential showed only a minor difference for writing ±2%/
Mazahari et al. (2016) [25]	±0%	20 - 30 %	±0/%	3 - 7 %	

TABLE 3.3: Overview of relevant overhead from other research on the performance implications of virtualization.

From the existing research studies discussed above, it can be concluded that disk IO is often heavily affected by running in a VM. For containers, this performance penalty cannot be considered negligilbe, only if volumes are mounted for the IO operations. Depending on the type of application, this could be a suitable solution. Nevertheless, we conclude that applications revolving around heavy disk IO should be treated carefully when running in a virtualized environment.

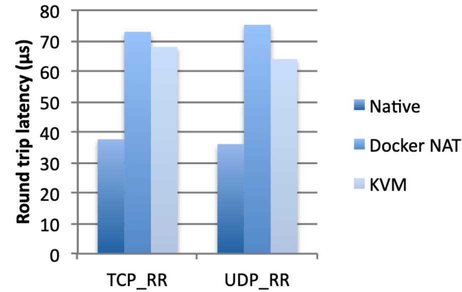
Networking

As mentioned before, VMs always mimic a set of hardware interfaces for the guest OS, for instance a network adapter. Additionally, most container setups have a virtual network interface assigned. This means that every call to a network outside of the virtualized environment would have an extra hop to take, when reaching the outside network. Again, in this case, containers have the benefit of not requiring another virtual Network Interface Card (NIC), meaning that one would expect better networking performance compared to VMs. Felter et al. [15] shows that for TCP troughput measurements, the differences between VMs, containers, and bare metal are negligible. They do show TCP transmission efficiency, by measuring the number of CPU cycles per transferred byte. They show containers, using Network Address Translation (NAT) suffer more, approximately 25 % from efficency than VMs. However, it should be noted that a very specific hypervisor bypass setting was used here. As stated above, when Xen and KVMs were being used in applications right after their launch, they have been struggling to provide IO performance close to native, having every packet passing through the hypervisor. This led to considerable research on network acceleration technologies, such as polling drivers and hypervisor bypass [22]. The latter is used for the measurements by Felter et al., allowing the VM to communicate directly with the host kernel. Although this significantly improves performance, it might not always be the feasible or chosen solution for organizations due to security/networking restrictions. Again, it should be noted that this is considered the best-case scenario. In a real-life production environment, where VMs are often preferred to increase portability and efficient resource usage, things like hypervisor bypass and CPU pinning might not be maintainable. This is often only applied to very specific applications, for instance with load balancers, where network performance is critical. However, for those applications, it might be more straightforward to not use virtualization at all.

Li. et al [21] shows an overhead of approximately 55% overhead in networking data throughput using iperf, also TCP connections. Containers only showed 2% overhead of data throughput, but showed a significant increase in variability of 55 %. The numerical values of throughput and variability are shown in Figure 3.5a. They point out that they have concerns about hypervisor-related configurations that might affect their results and aim to inspire further investigation. However, from Felter et al. [15] and Li et al. [21], it can be concluded that VMs suffer from virtualization overhead for network performance in absence of hypervisor bypass. As discussed above, it commonly not feasible to run all VMs in a cloud-native environment using hypervisor bypass.

Resource Type	Average	Standard Deviation
Physical machine	29.066 Mbits/sec	1.282 Mbits/sec
Container	28.484 Mbits/sec	1.978 Mbits/sec
Virtual machine	12.843 Mbits/sec	2.979 Mbits/sec

(A) A table of measured network throughput for KVMs (no hypervisor bypass) Docker containers, and bare metal machines [21].



(B) Network round-trip latency comparison for Docker containers and KVMs. Both TCP and UDP RTT is tested, [15].

Due to the extra hops made intrinsic to virtualized environments, especially in the context of a distributed system that has more communication, the round-trip response time (RTT) might be more of interest. These hops (extra layers) might introduce more latency that might not be directly measurable at the network layer. IBM research report [15] shows that the RTT of the network, read latency, is significantly worse for VMs, measuring both TCP and UTP RTT. As shown in Figure 3.5b, KVMs add 30 of overhead to each transaction compared to the non-virtualized network stack, an increase of 80% for their setup. For containers, the RTT also increases significantly up to approximately 35 μs. Unfortunately, Felter et al. does not discuss this more in detail; however, the effects of Dockers Network Address Translation (NAT) are clearly visible. The latter can be viewed as a sort of “userland-proxy”, which is created for every forwarded port.

There is only a small difference between UDP and TCP, since Felter et al. only uses a transaction of a single packet for this test. Contrary to a throughput test, the advantages of UDP are not brought to practice.

Furthermore, for containers networking performance, refer again to Kozhirbayev et al. [19]. They perform an IO networking benchmark with the Netperf[3] benchmarking tool. This test is performed by running on two different hosts and measuring throughput for UPD and TCP, for bare-metal, LXC and Docker containers. For both TCP and UDP, Docker containers show a decrease in transfer rate (per second) of approximately 8.5 % compared to native (bare metal). Kozhirbayev et al. [19] also points out that the impact of Network Address Translation (NAT) can be mitigated by using the host network (*nethost*), similar to the hypervisor bypass *vhost* that was previously mentioned for VMs. However, as discussed, this removes some of the virtualization advantages; for containers, this would negate the advantages of network isolation by namespaces.

From the existing work discussed here, it can be concluded that networking is heavily affected by running applications on a virtual machine, where RTT can get worse by

[3]https://linux.die.net/man/1/netperf

up to 30 μs. Additionally, throughput can be reduced by 50 % or less. Both cases do not consider hypervisor bypass. Containers show no significant decrease in network throughput. However, they also show a significant increase in latency RTT of 35 μs. This is due to Docker its NAT and could have been improved since the time of publication due to technical advances. We do not present a table here, as there are too many differences in the way network performance is measured among these studies. Additionally, the specific optimizations used in Felter et al. might give a biased view on what is feasible in a production environment.

Finally, we list an overview of the metrics that are used by existing works discussed above. These metrics can function as a guideline for our work, deciding which metrics to take into account and how.

- **CPU**
  Different benchmarks exist, where often three units of measurement are being used.
  - **Total exeuction time** The most straightforward way is to measure the total execution of a deterministic set of operations. Although this is easy to grasp, it can be heavily affected by other performance penalties, e.g. IO workloads or interactions with RAM. Nevertheless it represents better how an application its performance is affected.
  - **FLOPs** Floating point operations per second is an appropriate way to measure how much CPU is being used by an application. More specifically this is a very exact way of measuring how much 'work' is performed by the CPU in a second.
  - **CPU usage (percentage)** The most common way we see CPU usage is by the percentage of CPU that is being used. It represents the percentage of the amount of time a CPU spends processing non-idle tasks.
- **Memory**
  Memory is a bit more straightforward, where commonly two types of metrics are being used.
  - **Throughput (maximum)** As shown in existing work, there are benchmarks available that measure what is the maximum bandwidth that can be achieved by the Random Access Memory (RAM). Often a combination of small/large memory blocks and sequential / random access is taken into account.
  - **Memory usage (bytes)** Another way to measure this is measuring the number of bytes being used. This approach focuses more on the application side of measuring performance.

- **Disk IO**
  Existing work shows that Disk IO is heavily affected by virtualization layers. The following metrics are common ways of measuring this delay.
  - **Throughput** The amount of bytes that can be written to the disk is representative for processing large files, but often depicts an optimal scenario. Also for disk IO, often random / sequential read and writes are measured separately. Futhermore, smaller and larger file segments are taken into account.
  - **Latency** The time it takes to actually access a file (and thus open a file handle) might be more useful for the context of this research. Opening the file, meaning obtaining a file handle, is affected more by the virtualization layer than actual throughput.
- **Networking**
  Lastly we consider networking, which is often most heavily affected by virtualization layers. Two common ways of measuring networking performance are listed below.
  - **Throughput (maximum)** The maximum throughput that can be achieved is a common way of measuring networking performance. However, in the context of distributed web applications, little bottlenecks are expected here. As (synchronous) applications are often not bounded by network throughput. Nevertheless, the networking throughput can suffer from severe performance penalties for VMs.
  - **Latency** We have seen in the existing work of this section, that also latency can be heavily affected for both scenarios. Specifically in distributed web applications, on average containing more networking calls, might suffer from these penalties. Therefore network latency is considered an increasingly important metric.

3.3 Research context

As described in the previous sections, there is already a lot of existing work on the performance implications of different virtualization techniques. These provide information on the performance metrics of the system affected by virtualization. This gives us a good overview of what can be expected as possible bottlenecks, also within distributed systems. However, most of these studies perform common benchmarks in an isolated single instance. These benchmarks have existed and evolved over decades. These studies

have proven that these benchmarks provide valuable insights. However, the trend nowadays to move to cloud-native applications results in a rise of distributed systems like the microservice architecture. In our view, this calls for a different approach to measuring performance, where we should take into account the properties of the distributed system itself when measuring performance in different environments. Here, this research tries to fill the gap by aiming to find a more suitable way to measure performance within distributed systems. This is done to provide better insight into how distributed systems are truly affected by virtualization. One of these approaches is to use distributed tracing. A detailed description of how this works will follow in the following subsections. In doing so, it can become harder to generalize for any system as the benchmarks become less reproducible; however, it might open up new insights on the performance of distributed systems in virtualized environments.

3.3.1 Workload generation

Curiel et al. [10] discusses workload generators for web-based systems, describing their characteristics, current status, and challenges. Their aim is to compile the most important research contributions in the area in recent years and to structure those for a better understanding of the current challenges when generating representative workload.

Curiel et al. states that, in general, generating workloads can be challenging for two main reasons. First, the workload must be representative for the system under test (SUT). This SUT can be in any domain such as e-commerce, scientific computing, or databases, making it difficult to create a general-purpose workload. Second, the workload space is changing at a fast pace because technologies change rapidly and new applications appear constantly. Finally, the workload is often a partial reflection of human behaviour, introducing an additional level of variability.

For this research, we are basically 'measuring behind the load-balancer', directly at a group of load-balanced web services. This means we assume a fixed workload. Although this might not represent a real-life scenario, including dynamic workloads would affect the consistency of our measurements across different environments. An illustration of this setup is shown in Figure 3.6. This means that user behaviour, for instance access patterns, is not taken into account, with the aim to scope the measurements of our research. The range of interest is a group of web services that interact in a synchronous flow.

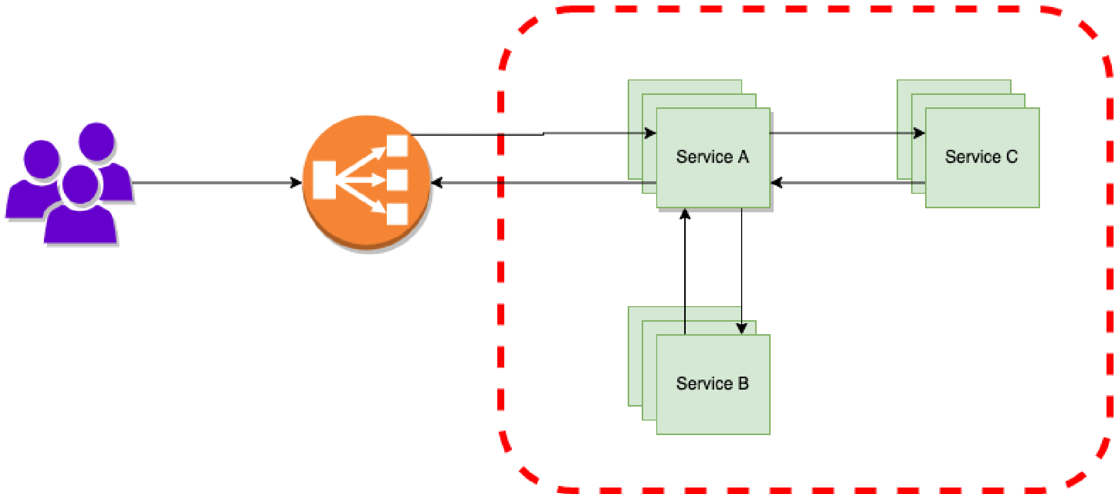


FIGURE 3.6: Overview of a simple distributed system, where the red square indicates the region of interest.

Maintaining steady throughput for these benchmarks is against what follows from the existing literature on benchmarking. For instance, Casazza et al. [7] from Intel characterizes server performance for virtualization benchmarking. They state "It is simplest to measure performance when all measurements are conducted in a time window after all workloads are in a steady state. Although this may be well suited as a benchmark, it fails to represent many real-world usage models." Although a fair point is made, our research considers a specific scenario where user behaviour has less variance and a more constant throughput. The focus lies more on redundant services, running behind a load-balancer.

Take Adyen as an example, a big leader in the payment industry. They process payments globally, and thus experience less effects of, for instance, timezones differences. The throughput of requests per second is relatively constant, with some peaks and gradual increases/decreases throughout the day. In addition, most user interactions constitute a single HTTP request. Therefore, services are less affected by user-interaction patterns and busier time periods throughout the day.

One might wonder why we do not consider redundant services when this is the scope of this research. As mentioned in Section 1, we do focus on representative distributed systems that often include redundancy. However, in order to scope this research and exclude more factors that can affect the accuracy and variability of the measurements, such as the addition of load balancers, it has been chosen not to do so.

3.3.2 Comparing request flows

While taking a different approach from existing work [15] [25] [21] [17] in determining virtualization overhead, the goal is to measure performance based on response time mutations by distributed tracing. There are frameworks out there to compare request flows by code instrumentation. Examples are Google its own DAPPER [41] and Facebook its Canopy [18]. Both use similar techniques to extract information from the request flow by comparing traces, but are unfortunately not publicly available. Nevertheless, these in-house built frameworks are expected to be too specialized for the organizations needs and are built for real-time monitoring. This makes existing frameworks bad candidates to be applied within this research, whereas we want to compare past events extracted from persisted storage while considering different runtime environments.

A different approach was taken for similar purposes (to ours) by Sambasivan et al. (2011) [36]. They describe and implement a technique to gain insight into the performance changes of a distributed system. They do so by comparing request flows from two executions, using code instrumentation data. They refer to new request flows, for instance, containing a code change, as *mutations*. Request flows that come from the original state of the system are referred to as *precursors*. In the context of this research, the change can be considered as moving to a virtualized environment. Consequently, the mutation is to run an application in, for instance, a container, and the precursor is our baseline bare-metal environment. Furthermore, Sambasivan et al. focuses on response-time mutations and structural mutations. The former is self-evident, while the latter corresponds to requests that take different paths through the system. This research will focus solely on response-time mutations of individual traces. However, structural mutations can still occur, for instance, with multithreaded applications due to their non-deterministic nature. Due to context switching and the internal OS scheduler, some request flows can have different orders of internal concurrent operations.

Typical latency distributions do not follow a known distribution [31] [36]. Latency distributions often have a long right-sided tail, as shown in the example in Figure 3.7.

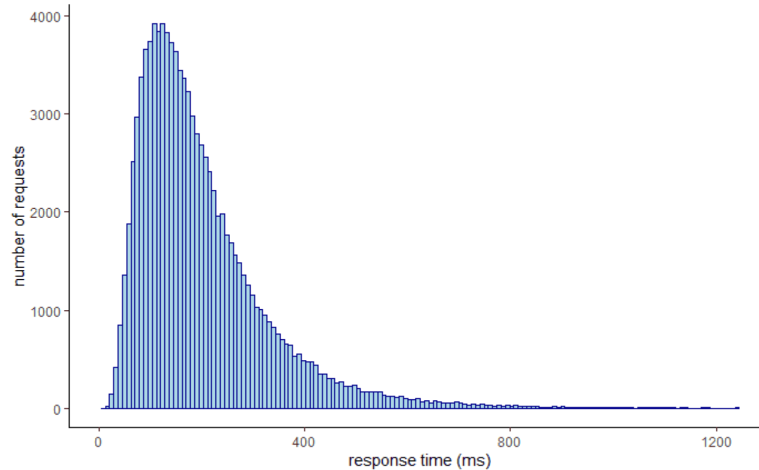


FIGURE 3.7: Example plot of a latency distribution [44].

This tail is essential for latencies and should not be arbitrarily discarded, especially in the context of distributed systems. A simple example is the Service Level Objective (SLO) of a company that wants their latency for a single web service to be below a specific value (e.g. 20 ms) in the 99 % percentile. Now, this is a common case where outliers in the last 1 % percentile are discarded. In this case, 99 out of 100 calls lie within this percentile. However, for instance this SLO could apply to hundreds of web services of a company. A simple end-user request flow, such as loading a modern webpage, depends on at least 150 requests. In this case, it means that $(1 - 99^{150}) * 100\% = 77.85\%$ of the time, the webpage would experience the upper 1% of the latency distribution. Table 3.4 shows a number of public webpages with the number of requests that occur to load all the resources on the website.

Website	**# of requests**	**% of page loads experiencing 1% percentile**
google.com - Homepage	36	30.35
google.com - Search 'I'm a teapot'	64	47.44
amazon.com - Homepage	291	94.63
linkedin.com - News feed	195	85.91

TABLE 3.4: Several examples of web pages with the number of requests made on a single page load, including the chance of being in the upper 1% percentile.

The analogy above is for webpages; however, this analogy also holds for APIs consisting of numerous smaller microservices. This shows that the tail of the latency distribution can be considered more important than the actual mean value [4]. The end user might not notice the loading of a webpage taking a bit longer on average, but would notice that one fragment takes significantly longer. However, it is important to note that the user

does not notice all requests made when opening a page. Slowly loading images might be visible, however, slow retrieval of marketing analytics scripts might be negligible.

3.3.2.1 Statistical testing

There are numerous statistical procedures to choose from when comparing two sets of samples with an unknown underlying distribution. The latter means that this research requires non-parametric tests, which do not rely on assumptions about the existence of distribution parameters. The outcome is whether two sets of samples come from the same distribution, also called goodness-of-fit. Additionally, for this research, it is of interest to know how significant any detected difference is. We consider a few possible options based on additional sources [9] and similar research [36]. Unfortunately, existing frameworks such as Google its DAPPER [41] and Facebook its Canopy [18] do not expose what kinds of statistical tests are being used. Therefore, most alternatives are listed from research papers solely testing different capabilities of two-sampled two-sided nonparametric tests:

For this research, we consider a few different types of statistical test, all having a variant that is nonparametric and two-sided. Using either a parametric or one-sided test would not be possible, as for the former, we need to know the underlying distribution. To use a single-sided test, the underlying distribution parameters of one of the data sets would be required. The statistical tests listed below seem suitable for the purpose of this research. The Shapiro-Wilk test and Chi-square test are left out of scope, as the literature proves, they are less powerful for the use case of this research. The Shapiro-Wilk test is more suitable for sample sizes below 50. There are some alternatives to this test for larger sample sizes, however, still up to $n < 5000$. Furthermore, the chi-square test considers binned data by simply comparing histograms. This introduces the risk of leaving out important information about the tail of a distribution [28] [42].

- **Kolmogorov-Smirnov test (KS-test)**
  The two-sample KS-test quantifies the distance between the empirical distribution functions (eCDF) of the sampled data sets. The resulting statistic value is representative of the maximum distance between the two CDFs. Sambasivan et al. [36] also uses the KS-test to compare request flows, and this proves successful for differences in request flow timing up to 2ms. For this research, its most important property is that the test is particularly sensitive to changes around the median [39] [24].
- **Cramér–von Mises test (CVM-test)**
  The two-sample CVM-test also is a criterion to quantify the goodness-of-fit of

two eCDFs. It is considered an alternative to the KS-test, using the summed squared differences of the two functions. Compared to the KS-test it is slightly more sensitive to changes in the shape of the distribution. [3]

- **Anderson-Darling test (AD-test)**
A modification to the KS-test is the AD-test. It has some modification in its methodology, not using the maximum difference but calculating the averaged logarithmic distance between the two eCDFs. This makes the test more sensitive to differences in the distribution its tails for two datasets. Furthermore, it has been shown that, compared to other tests, the AD-test is more sensitive to smaller changes, specifically at larger sample sizes [12] [39].

- **Mann-Whitney (Wilcoxon) test (MW-test)**
The MW-test, also known as the Wilcoxon rank sum test, first ranks (sorts) all values in both data sets. Subsequently, a P-value is calculated that is dependent on the differences between the mean ranks of the two data sets. The MW-test is generally known as a less powerful statistical test and takes the shape of the distribution even less into account. Nevertheless, as it makes few assumptions about the data, it is applicable in cases where it would be incorrect to use other more powerful tests (KS-test). An example would be if the data set is not continuous, where the KS-test would then falsely assume that it is [40] [9].

In Table 3.5, we describe a high-level comparison of the properties of each test related to what is needed for this research. This comparison is based on the earlier referenced research studies considering these statistical tests, and is based on our own interpretation of their conclusions.

Statistical test	**Sensitivity median**	**Sensitivity variance/shape**
Mann-Whitney	+-	-
Kolmogorov-Smirnov	++	+
Anderson-Darling	+	++
Cramer-von Misses	+	+

Table 3.5: Comparison of well-known statistical tests when comparing distributions

As mentioned before, it is important to compare the means of our sample distributions and also to take the tails into account. As described before, even samples outside the 99 percentile can have significant effect on the total (distributed) system its performance. Considering the comparison above, it seems that both the Kolmogorov-Smirnov test (KS-test) and Anderson-Darling (AD-test) test are good candidates for comparing request

flows. Sambasivan et al, 2011 [36] compares request flows using the KS test, and prove to detect time differences of 2 ms for a trace span.

From the analysis of the existing literature on statistical tests, we expect the KS-test and AD-test to be most valuable for the purpose of our research. The following subsections describe the two tests more in-depth. Both tests are based on the cumulative distribution function (CDF) of the sample data. In simple terms, they calculate the distance between the distributions. This research focuses on performance comparisons of virtualized environments. As we are not implementing the statistic tests ourselves and neither focusing on their internals, we use existing implementations of the SciPy[4] Python library. Therefore, no derivations are presented in this section. Only a brief introduction to the mathematical approximation of both tests will be given in the following subsections.

3.3.2.2 Kolmogorov-Smirnov test [24] [12]

As discussed, the purpose of the research will not necessarily be to look at the mean value of latency, but rather at the distribution of values as a whole. Therefore, the Kolmogorv-Smirnov test is a well-fitted choice here, to be able to detect the request flows with the largest differences between different environments. This corresponds to the approach taken by Sambasivan et al.

The Kolmogorov-Smirnov test was originally a test to measure the deviation of empirical distributions from a known theoretical distribution. This means comparing a set of data samples to a known distribution (e.g. normal distribution) $F(x)$. The KS statistic for a set of samples and a known CDF is described by Formula 3.1 below.

$$KS_n = \sqrt{n} \max_x |F_n(x) - F(x)| \tag{3.1}$$

In this equation, $F_n(x)$ is the empirical cumulative distribution function (eCDF) with a sample size n. For this test, the null hypothesis H_0 is that $F_n(x)$ comes from the underlying distribution $F(x)$. It is rejected if KS_n is larger than the critical value KS_α at a given α. A table with critical values for different sample sizes is not included here but can be found in the original article (Massey et al. [24]). A simplistic view of this test is to think of a band with a thickness of $2 * KS_\alpha$ that is drawn around the center of the known CDF. If the eCDF falls out of this band, the null hypothesis is rejected.

[4] https://scipy.org/

As discussed before, this research requires a two-sample version of the statistical test, since latency (traces) generally does not follow a known distribution. This means that we do not have parameters to characterize the distribution. The two-sample version of the KS-test is described by Formula 3.2 below.

$$KS_{nn'} = \sqrt{\frac{nn'}{n+n'}} \max_x |F_n(x) - F_{n'}(x)| \tag{3.2}$$

In this equation, $F_n(x)$ and $F_{n'}(x)$ are the eCDFs of two sample sets, based on sample sets of size n and n' respectively. The same analogy follows, where the null hypothesis is H_0 that $F_n(x)$ and $F_{n'}(x)$ are from the same underlying distribution. It is again rejected if KS_n is greater than the critical value KS_α at a given α.

3.3.2.3 Anderson-Darling test [39] [12]

As discussed above, one of the other most viable options for statistical tests is the Anderson-Darling test (AD-test). At the beginning of this chapter, it was concluded from existing research that the AD-test is expected to be more sensitive to changes in the shape between two distributions. Other research has shown that it commonly outperforms the KS-test [28] [12].

The AD-test is originally simply an alternative to other statistical tests for detecting difference between a sample data set and a known normal distribution. The one-sample AD-test its statistic can be calculated by Formula 3.3 below.

$$AD = -n - \frac{1}{n}\sum_{i=1}^{n}(2i-1)\left(\ln\left(x_{(i)}\right) + \ln\left(1 - \left(x_{(n+1-i)}\right)\right)\right) \tag{3.3}$$

In this formula, $\{x_1 < ... < x_{(n)}\}$ is the ordered sample set of size n. The null hypothesis H_0 of this test is that this set of samples comes from the underlying distribution $F(x)$ of Formula 3.3, and is normally rejected if the statistic is greater than the critical value AD_α which can be calculated using the corresponding tables provided by the original article, in which case α is the specific critical value (e.g., 0.25).

For a two-sample test, the equation transforms to an extended equation defined by Formula 3.4 below.

$$AD = \frac{1}{mn}\sum_{i=1}^{n+m}\left(N_i Z_{(n+m-ni)}\right)^2 \frac{1}{iZ_{(n+m-i)}} \tag{3.4}$$

In this equation, $Z_{(n+m)}$ represents the combined data sets of ordered samples X_n and Y_m of sizes n and m, respectively. Furthermore, N_i is the number of observations in X_n that are smaller than or equal to the number of observations i in $Z_{(n+m)}$. Again, H_0 stating that X_n and Y_m come from the same distribution is accepted if the test statistic turns out to be smaller than the critical value corresponding to the sample size.

Lastly, both the AD-test and KS-test seem good candidates for the purpose of this research. However, we should take into account that with large sample sizes ($n > 1000$), p-values tend to drop to zero if the variability is relatively high. Both tests are not very suitable for larger sample sizes. However, as we know from existing literature, time measurements within virtualized environments often have a high variability which in turn asks for large sample sizes to capture all system properties. Therefore it could appear that the best test for comparing cumulative distribution functions wihtin this research might be simple percentiles or even an eyeball test.

Chapter 4

Methodology

This chapter will discuss the methods applied to gain insight into the performance of virtualized web applications in a distributed environment. First, the system under test (SUT) will be discussed, including the different virtualization setups. This is followed by a detailed description of the complete measurement setup, including workload generation and data collection/storage. Finally, this chapter will describe the steps in which data are collected and processed, leading to the final results.

This research considers four different virtualization setups, including a bare-metal server setup, which seem typical for a subset of enterprise applications moving to the cloud or simply virtualized environments. The applications will run in the following environments:

- Bare-metal dedicated servers
- Containers running on top of bare-metal servers.
- KVMs
- Containers running on top of KVMs

More detailed descriptions per environment and specific to the SUT will be discussed in the following subsections.

4.1 Systems under test

The SUT for this research is a simple e-commerce Java Spring Boot application consisting of the following three microservices, which can be found on Github[1]:

[1]https://github.com/daanvinken/microservice-example-tracing

- Order service
- Payment service
- User service

The application has been adapted to represent a common enterprise application. This includes calling other services, databases, and performing some heavy IO / CPU workloads. This resembles typical workloads for companies with high-performance web endpoints, where API calls involve multiple services. As mentioned before, this research is performed at Adyen, where typical services/jobs perform many IO operations, like logging or remote service calls. With respect to the complexity of the application, this research chooses to restrict the Java implementation from complicated code patterns. Therfore, we consider at the De Capo benchmark research paper [6], a report and benchmarking guideline for Java applications (Blackburn et al., 2006 [6]). They state that complexity in Java code flow produces more variety and more complex behaviour at runtime. This research prefers to be as close to a real-life production environment as practically feasible. However, this quickly results in a trade-off between reproducibility and consistency and the system representing a real-life production scenario.

For the upcoming benchmarks, the focus lies on an arbitrarily designed and chosen request flow, which is depicted in Figure 4.1. Later in this section the complete workload on those services will be set out step by step, however, for now the request flows on a server-to-server level will suffice for discussing the different environments. As mentioned in Section1, this system should be imagined to function behind a load balancer, which in our case is replaced by a load generator.

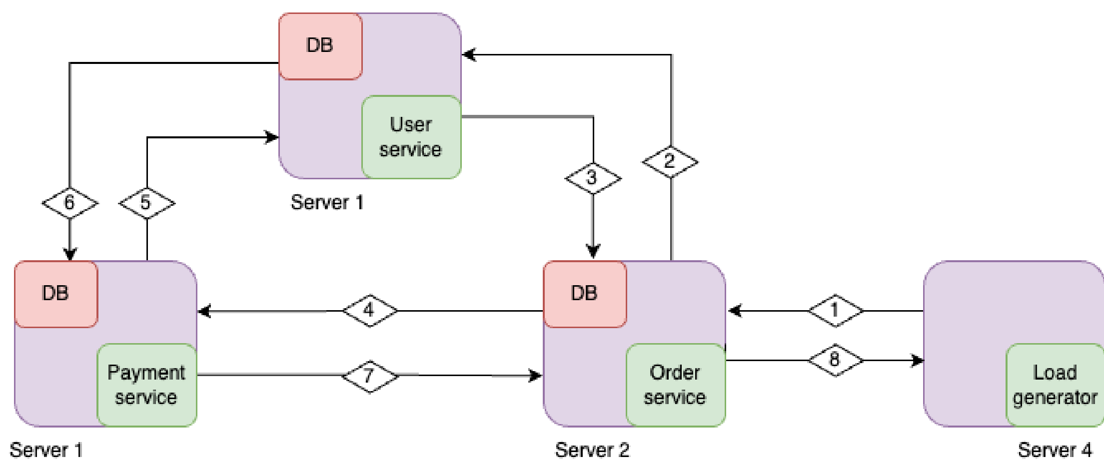


FIGURE 4.1: Graphical representation of our SUT, with servers (purple), applications (green) and databases (red).

In all cases, the servers are connected via a Virtual Local Area Network (VLAN). Figure 4.3 shows the four different setups to which the upcoming benchmarks will be applied. For convenience, the default Postgres database container from Docker Hub [2] is used as a database instance. Running the database in a container can have performance implications. However, due to having the same setup in all four scenarios, no difference in performance impact is expected.

Initially, one might expect disk IO to be slower as well; however, within the container image definition volumes are defined to overcome the performance penalty for disk IO. We have seen in existing research, set out in Chapter 3, that this makes any performance difference insignificant.

Hardware

The servers in practice are from a UK-based Fasthosts[3] data center. The chosen servers have an AMD Ryzen 5 PRO 3600 CPU with six hyperthreaded cores with 3.6 Ghz clock speed frequency. The caches are L1: 384KB, L2: 3MB, and L3: 32MB on a single socket. This CPU is supported by 32 GB of DDR4 memory and 2 SSDs of 480 GB. To increase performance and avoid extra hops in routing, a private network (VLAN) is established between these servers. This is a direct layer 2 connection between these servers, configured via an additional NIC. In practice, this should result in lower latency and higher bandwidth (claimed up to 10 Gbit/s) compared to the public IP network.

Furthermore, the Network Time Protocol (NTP) is set up on these servers to minimize the clock drift between the servers. One of the servers is configured to sync with three public NTP servers. Additionally, all other servers sync with this single server through their configured VLAN. In this configured setup, we have not measured an estimated error higher than 80 μs. NTP calculates this value itself, which also becomes available to the node exporter, which will follow later. An example of the estimated time error can be seen in Figure 4.2 below. As can be seen in this figure, around 11:24 a small correction on the drifted time occurs, performed by NTP.

[2]https://hub.docker.com/layers/library/postgres/14.5

[3]https://www.fasthosts.co.uk/cloud-servers/bare-metal

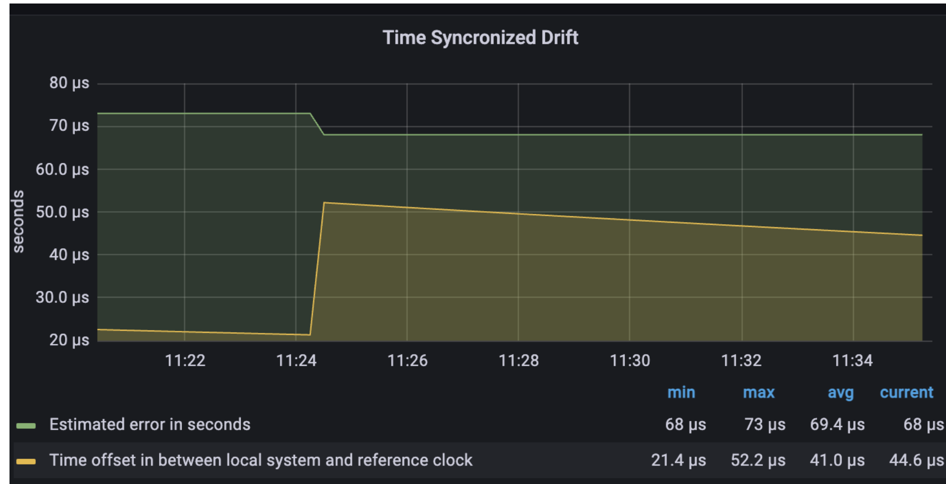


FIGURE 4.2: The Time Synchronized drift measured on one of the servers.

The following sections provide some context and considerations on the test setups in practice, which are depicted in Figure 4.3.

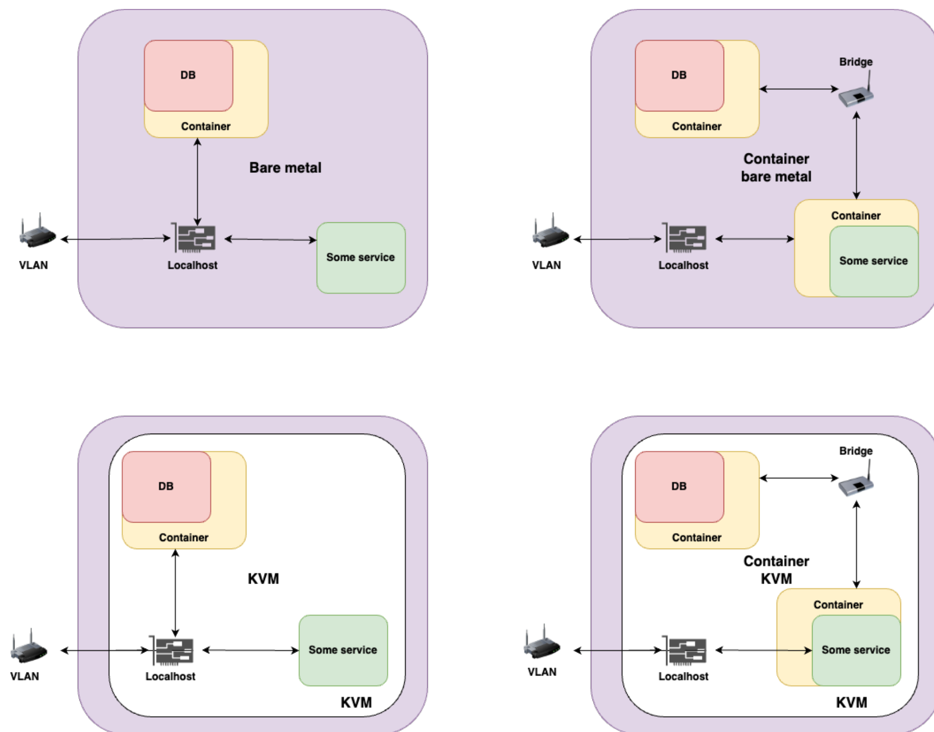


FIGURE 4.3: Overview of the four different (virtualized) environments that are benchmarked within this research.

4.1.1 Bare-metal

The bare-metal setup, commonly named a dedicated server, is the baseline for the upcoming measurements. It should technically not be possible to outperform this scenario,

as it always has the lowest number of abstractions. In this context, an abstraction, like virtual IO interfaces, could imply for instance extra CPU cycles or network hops. As indicated in Figure 4.3 the database container is attached to *localhost* and is then connected to the application.

In this research, we try to mimic a real-life production environment within the boundaries of available resources and time. By these boundaries, it is not possible to have a remote database. Therefore, databases are run locally on the same host, in a container. This is another common setup for enterprises, letting the database reside 'closer' to the application. Note that running a database in a container is not a typical setup. However, we prefer consistency within our database setups. The other scenarios involve services running in a container. Later on, it will become clear why running the database in a container is a more representative setup. Typically, databases run remotely or just as a daemon on the same server. But for those reasons, we maintain consistency and apply the same database setup in the bare metal scenario.

4.1.2 Container on bare-metal

With the ongoing trend of containerizing (web)applications, mentioned in Section 1, this setup seems to be a common case for companies. Running their applications in containers allows them to take advantage of the scalability and isolation that containers offer.

Chapter 3 shows that performance differences can arise between different container runtimes. Kozhirbayev et al. [19], for example, found that Flockport (LXC) can outperform Docker during benchmarks. However, as mentioned before, from a survey of Enlyft it is known that from 50 000 software companies, 97 % uses Docker. With this market share, it seemed more representative for this research to use Docker. It is important to take best practices into account, for instance, not using Docker its overleaf filesystem (AUFS) when performing many disk IO operations.

As can be seen in Figure 4.3, in the case of a containerized web service, the container is connected to the database over a Docker network (bridge). Initially, this may seem like an unfair comparison. Nevertheless, having an isolated (virtual) network for intra-container communication is very common. Especially with the rise of container orchestration systems. A well-known example of this would be intra-pod communication within a Kubernetes cluster. On the contrary, imagine that the database and web service container would both run on the host network. In that case, we would not take into account the additional network hops to the virtual network interfaces. Following

the above, we certify that using a bridge network for containerised scenarios is more representative for real-life production environments.

4.1.3 KVM

With an increasing number of companies running their applications in a cloud environment, virtual machines are becoming the common case for most systems. Some cloud providers do not even offer dedicated (bare-metal) servers. If a cloud provider does, they often have a small separate section for dedicated servers, while most if not all other offered services run in some kind of virtualization layer. This also implies the need to see how (distributed) web applications perform in this type of virtualization.

As discussed in Section 3 there are different types of hypervisors available. However, KVMs seem to be the most common option that also has the best overall performance. Additionally, some of the biggest cloud providers like Google Cloud Platform (GCP)and Amazon Web Services (AWS) use KVM-based hypervisors.

To have a fair comparison with our bare-metal baseline, the applications will be connected via the guest OS its localhost. This is a virtualized 'local' network interface within the VM.

4.1.4 Container on KVM

Due to the combination of both trends in the software industry, moving to containerized applications and more systems running in the cloud, this setup is becoming the most common case. Good examples are any container running on AWS, GCP or Azure. Some of the most used cloud products for containers services, respectively Elastic Kubernetes Service (AWS), Elastic Container Service (AWS) or Google Kubernetes Engine (GKE), all run in virtual machines. Again, with this setup, we use a bridge network between containers on the same virtual host. However, now this virtual network resides on top of the virtualized 'local' network interface within the VM.

4.1.5 Request flow breakdown

One of the ways to measure the performance of our SUT is distributed tracing. For the upcoming analysis of these measurements, it is good to be aware of the request flow a bit more in-depth. The application has been designed in a way that it emphasizes different types of system properties, like CPU, disk IO and networking. Here, we shall shortly go

over the specific operations that are being traced in the complete request flow and are included in our measurements.

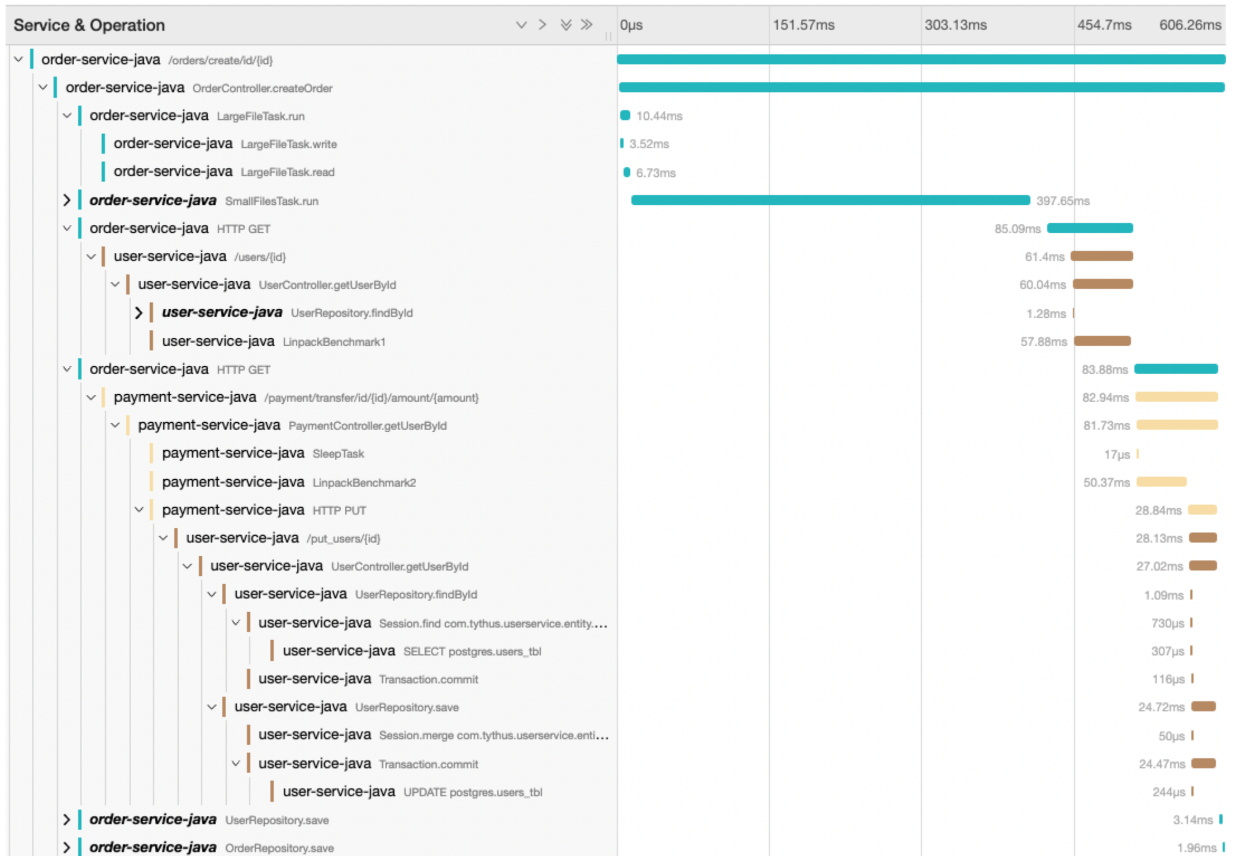


FIGURE 4.4: An overview of the recorded traces of the complete request flow, shown in Jaeger.

Some trace spans depicted in Figure 4.4 above are considered for our measurements and are set out below. Within this research, the wording latency, trace spans, and traces will be used interchangably.

- `/Orders/create/id/{id}`
  This is the *order service* call, which is the entrypoint of our API. As can be seen in Figure 4.4 this task contains a set of operations, which will be discussed below. The service performs some disk IO tasks, after which it calls other services. Finally it stores an 'order' to its own database.

- `LargeFileTask.write`
  This task writes a large text file, which content is stored in memory, to a location on the disk. The file is approximately 5 MB. The choice to include some file operations into our request flow is made, as disk IO seems to be heavily affected by virtualization. Therefore we prefer to include such operations into our SUT.

- `LargeFileTask.read`
  The same file is read from the disk again within this task.

- `SmallFileTask.run`
  This task, also contained within the *order service*, consecutively switches between reading and writing a small file from the disk, with a size of 140 bytes. It does this 10 times before returning.

- `/put_users/{id}`
  This PUT HTTP request to the user service where a simple SELECT and UPDATE operation on the user database is executed.

- `/users/{id}`
  This trace span represents the GET HTTP request to the *user service*, which finds a user based on its ID in its database. Afterwards, a small Linpack benchmark if executed called *LinpackBenchmark1*.

- `LinpackBenchmark1`
  This task runs a small Linpack benchmark implemented in Java[4]. This in order to mimic some CPU intensive worload.

- `/payment/transfer/id/{id}/amount/{amount}`
  This is the call to our *payment service*. The service itself contains the *SleepTask*, and its duration is set to 0 μs by default. Furthermore, it also executes a slightly longer Linpack benchmark called *LinpackBenchmark2*. Finally, it calls the user service (*/put_users/{id}*) which updates some properties in its attached database.

- `LinpackBenchmark2`
  This task runs a slightly larger Linpack benchmark also implemented in Java.

4.2 Test setup

Benchmarking computer systems comes with its caveats, especially in a distributed environment. Getting closer to a real-life representation of a distributed production system under test requires a different approach to performance measurements. This real-life representation involves running with a concurrent user load of an enterprise application, while not exhausting resources. This exhaustion of resources is unacceptable for enterprise applications. It exposes businesses to the risk of failing to meet their SLAs due to unaccounted latency spikes, and doing so enables them to take account of resource utilization peaks.

[4]https://netlib.org/linalg/linpack.b

This contradicts how most benchmarks are performed, as mentioned in Section 3. In those cases, resources are exhausted for a long time, trying to achieve the highest possible throughput on a single isolated instance. This merely depicts an optimal scenario. This research attempts to measure the performance of virtualized environments in a more representative setting. Instead of looking at narrowly scoped tests for instance specific to CPU bound or IO bound standard benchmarks, this research looks at the system as a whole, taking more common properties of distributed web applications into account. For example, we include calls to other services into our measurements. Hereby we investigate what could be the effects for large enterprises when moving their applications to virtualized environments.

This research will use the following methods of measurement.

- **Distributed tracing**
  This research uses Opentelemetry distributed tracing [5]. A Trace records the paths taken by requests (made by an application or end-user), from the moment the request is made, propagated through multi-service architectures, all the way to the final response. This technique is often applied with microservice and serverless applications. With this technique the aim is to gain deeper, more fine-grained, insights in which parts of applications experience performance bottlenecks due to virtualization.

- **System level metrics**
  Aditionally, a number of system-level metrics among CPU load, memory usage, or number of context switches will be monitored. The outcome of these measurements could, for instance, indicate whether more resources are needed, while running the same application in a different virtualized environment.

4.2.1 Workload generation

For this research, Apache Jmeter[6] will be used as a workload generator and load tester. This is a well-known tool from Apache, used to measure and analyze the performance of different services, with a focus on web applications. Some features important for this research are its headless mode, multithreading support, reporting/analysis capabilities, and extensibility. The latter means that it supports a constant-throughput timer. This is important for our research, as in order to compare different virtualized environments, throughput should be kept constant. In Section 3 it has already been discussed why we choose not to simulate user behaviour.

[5] https://opentelemetry.io/
[6] https://jmeter.apache.org/

For constant throughput, there is one important side note. It is practically infeasible to set the exact number of requests per second. Say, an application has an average response time of 30 ms, but has outliers of 2000 ms, which is common when looking at the 99th percentile of APIs. With load testing, often multiple threads spawn requests at the same time. Therefore the exact next timing of a next request is variable and can be different after every finished request. Therefore, some variability is expected in the number of requests per second.

We have chosen to set the approximated number of requests per second to 20. Based on our first measurements on the SUT, while not limiting throughput, it was found that the system could process at least 25 requests per second. Choosing this value slightly lower, at 20 requests per second, allows us to ensure that we are not overloading our application.

4.2.2 Data collection

This subsection describes the complete test setup, including the essential parts for data collection and analysis. A schematic overview of the entire test system can be seen in Figure 4.5. The diagram contains five larger squares that represent the servers used in our test setup. The hardware of these host machines is already elaborated in Section 4.1, which is identical for each server.

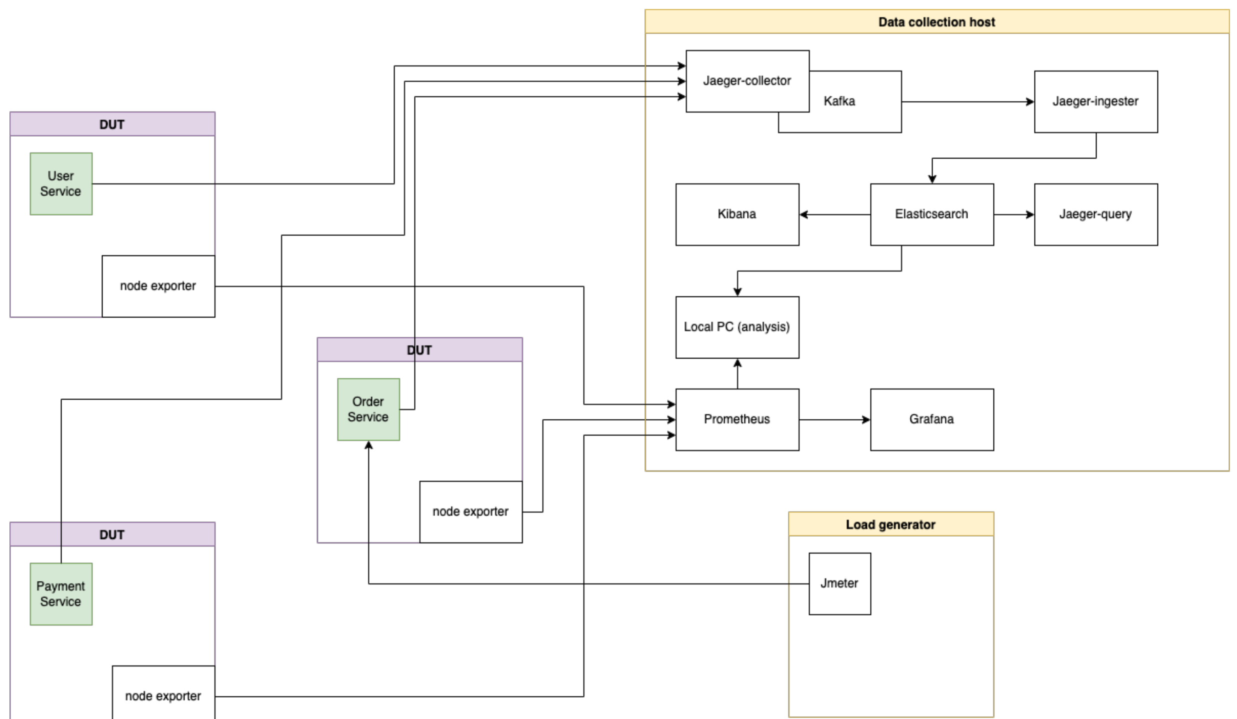


FIGURE 4.5: An overview of the benchmarking system architecture.

Naturally, the servers that are not under test could be any machine, but the three devices under test (DUTs) should preferably be identical. This is done in order to have consistency in the test setup and, therefore, to exclude external factors affecting the measurements. In Figure 4.5, green services correspond to those described in Section 4.1. All applications on the data collection host are deployed as a Docker container, all at once via docker-compose[7]. We now elaborate on the element used in the test setup above:

1. **Jaeger-collector**
   Jaeger's collector element is a stateles data collector, which means that many can run in parallel to match the incoming throughput. However, to minimize the risk of losing data and allowing for high throughput, this research uses Apache Kafka as an intermediary buffer between the collector and our eventual storage.

2. **Kafka**
   Kafka is a distributed event store and stream processing platform, which functions as a buffer for our collected telemetry data. The application's main properties are high throughput and low latency while handling one or more real-time data feeds. These real-time feeds can be (sub)divided into *topics* which are grouped sets of real-time data feeds.

3. **Jaeger-ingester**
   The Jaeger-ingester is a service that reads span data from a pre-onfigured Kafka topic and writes them to Elasticsearch (or Cassandra).

4. **Elasticsearch**
   Elasticsearch is a popular open source NoSQL database that offers distributed storage and an analytics engine.

5. **Kibana**
   It is a data visualization dashboard for the database, which allows easy visualization and data exploration. Kibana is from the same organization as Elasticsearch and offers seemless integration between the two.

6. **Jaeger-query**
   The query service serves the API endpoints and a browser-based UI to easily explore the tracing data, as can be seen in 2.4.

7. **NTP**
   On all hosts, Network Time Protocol (NTP) is running as a daemon. NTP is a protocol designed to synchronize computer clocks over a network. We configure

[7]https://gist.github.com/daanvinken/e6db746b064b41a290d19c70b0a9e108

the data collection host to synchronize with external NTP servers (over the public Web). The DUTs will synchronize with this cluster, resulting in a consistent state across our VLAN. This improves the accuracy of our collected tracing data, minimizing the effects of clock skew on each DUT.

8. **Prometheus**
   Prometheus is an open source alerting system monitoring framework, originally built at SoundCloud. It can record real-time metrics and store those in its time series database. It uses an HTTP pull model, with flexible queries and real-time alerting.

9. **Node exporter**
   On each server runs node-exporter[8], which is a metric exporter for hardware and OS metrics exposed by UNIX kernels. This is the source of data for Prometheus in this setup. It is written in Go and extends Prometheus by scraping different locations on the filesystem for specific metrics. An example is the location `/proc/vmstart` on most Linux systems, which contains regularly updated bits of system information like memory, paging, processes, IO, CPU, and disk scheduling.

10. **Grafana**
    Grafana is a tool for visualizing real-time data, with out-of-the-box support for prometheus exporters like node exporter. This is useful for early-stage data exploration.

4.3 Benchmarks

In this section the different measurement setups are being discussed. We set out the workflows and ways of collecting and gathering data.

4.3.1 General workflow

Every benchmark below follows the same workflow, executed from a single host by a shell script. The flow shown in Figure 4.6 is used for all tests, in a loop. This flow is often executed 11 times to make sure we gather enough results.

[8] https://github.com/prometheus/node_exporter

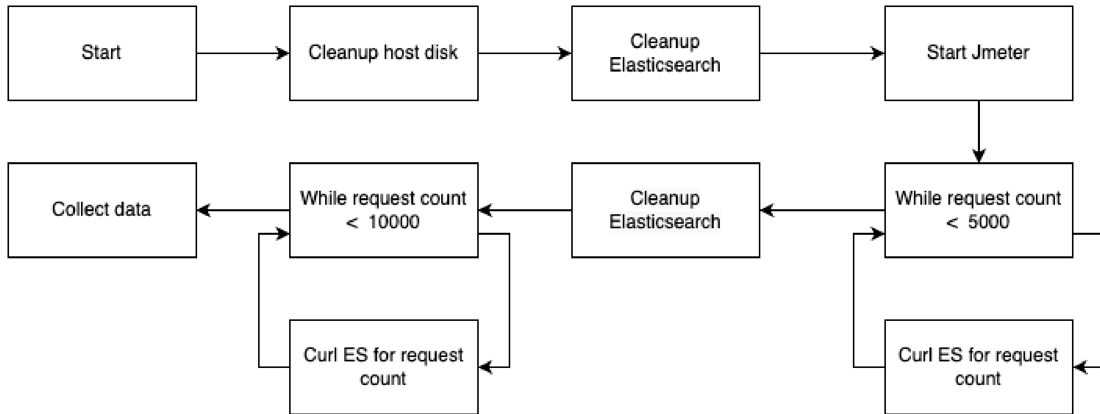


FIGURE 4.6: A flowchart of the testflow used for our benchmarks. This is indicative, if numbers vary this will be clearly indicated.

The host disk cleanup (or VM) is to clean up the files that have been written to the disk during each run. This is to prevent the disk from becoming full. Furthermore, after the data were collected, we clean up all the data in Elasticsearch (ES). The same cleanup occurs after approximately 5000 requests are loaded, via Kafka, into Elasticsearch.

The warmup period was carefully chosen, where we've seen after multiple runs in different environments an equilibrium was reached after 5000 requests. After this point, the variability of response times did not increase nor decrease. For the measurements related to trace spans, we used 10 000 requests for the actual measurements.

For measuring system-level metrics, it turns out the needed warm-up period, required for a steady state, is slightly longer. We've set this to 8000 and measured system-level metrics for the duration of 22 000 requests.

4.3.2 Distributed tracing

As mentioned in Section 3, existing research on measuring latency and trace data states that sample response times are not governed by a well-known distribution. Therefore, the non-parameterized statistical tests discussed will be used to determine whether there is any performance difference.

Before running any benchmarks on different virtualization techniques, it is important to determine which statistical tests are most suitable for the purpose of this research. To do so, it has been chosen to first verify these tests using the same distributed web application. These initial tests are run only on bare metal, introducing delayed operations by putting *sleeps* in the source code and comparing the request flows. In this way, it is possible to approximate the accuracy of our testing methodologies in advance.

An important note with this approach, specifically for Java applications, is that the well-known *Thread.sleep()* or its wrapper *TimeUnit.MICROSECONDS.sleep()*, are not a good option. This method always pauses the current thread its execution. This means that the actual time it sleeps before waking up and continues execution depends on the OS timers and schedulers. With a low system load, the actual time can be close to the specified sleep time, but with a higher workload on the OS, the actual time can vary heavily. Another option is to keep the thread busy and, therefore, not letting the OS its scheduler know that the thread is paused. In this way, the thread can keep checking when the specified time has passed, which is expected to result in more accurate timings. This method is often called a *busy wait*. The way to do this is simply to check the time passed in a loop for a while, as shown in Listing 4.1. We have verified that this is code is executed, as it might be negated by compiler opitmizations.

```
1     public static void busyWaitMicros(long micros){
2         long waitUntil = System.nanoTime() + (micros * 1_000);
3         while(waitUntil > System.nanoTime()){
4             ;
5         }
```

Listing 4.1: Java code for busy wait

Two tests with the highest estimated chance of success, namely the Kolmogorov-Smirnov and Anderson-Darling test, have been set out in chapter 3. To have a fair comparison between different environments and the runs contained in them, we include both tests in our final results. For the accuracy tests, we simply look at how the tests compare to the baseline (bare metal) trace span durations for a chosen service. In this case, we choose the *payment service*, which in turn calls the *user service*. We deliberately choose for this service, as it does include some networking and CPU bounded workload while excluding a possible increase in variability due to heavy disk IO.

In order to compare the actual different traces among our four test scenario, we propose an algorithm for the best fit selection. Simply put, this algorithm finds the most representative run for each of the four scenarios. It does so by finding the latency samples distribution that is the closest to all other sample distributions within the same scenario (e.g. container only). The high-level pseudo-code of this algorithm is presented in Algorithm 1 below.

Algorithm 1 The pseudo-code of the algorithm determining the run that best fits all other runs, using the AD-test.

```
1:  for scenario_i in all scenarios do
2:      for span_j in all spans do
3:          Set AD^n_min(total) = ∞
4:          for run x_n in all runs X_n do
5:              Set accumulated test statistic AD^n_total = 0
6:              for run y_m in all runs Y_m do
7:                  Calculate AD-test statistic AD_m for x_n and y_m.
8:                  AD^n_total += AD_m
9:              end for
10:             if AD^n_total < AD^n_min(total) then
11:                 Store x_n as best fit for this run x_optimal
12:                 AD^n_min(total) = AD^n_total
13:             end if
14:         end for
15:         if scenario == 'baseline' then
16:             x_baseline = x_optimal
17:         end if
18:         Calculate AD-test statistic AD, comparing (x_baseline,x_optimal).
19:         Store statistic in a 2D map Z, where z^i_j = AD
20:     end for
21: end for
```

As presented, we look at a group of runs within a single scenario (e.g. container only) and single trace span (e.g. payment service call). We know from Section 3 that the test statistic is representative the distance between two CDFs. The algorithm tests all runs within this same group, and sums up the test statistics. The run in a group with the lowest accumulated test statistic is chosen as the best fit and thus most representative trace span for that group. This setup is useful, as with high variability among different runs in the same group, it can become hard to properly compare latency for different scenarios. We propose this approach as a way to still analyze trace spans amongst different scenarios, while having a one-to-one comparison.

Measurements The accuracy measurements are performed with sleeps ranging from 50 μs to 500 μs. For every sleep length (0, 50, 100, 200, 300, 400, 500), ten experiments were performed containing approximately 10 000 requests. This number excludes the 5000 requests during the warm-up period. All other measurements follow the same test flow as depicted in Figure 4.6

4.3.3 System-level metrics

As an enterprise, it is not only interesting how the end-user is affected by the switch to a virtualized environment. Additionally, it is good to be aware of any changes in system-level metrics. Let us say that the memory footprint of the same service running in a container is higher than that of the same service running on bare metal. In this case, the enterprise might need to overprovision the same application.

To gather system-level metrics, Prometheus is the tool of choice. Prometheus is an open-source event monitoring and alerting application. It records real-time metrics in a time series database which is filled by using an HTTP pull model. It supports flexible queries, real-time alerts and a wide range of extensions due to so-called exporters.

A well-known exporter for Prometheus is node-exporter[9], which is a metric exporter for hardware and OS metrics exposed by UNIX kernels. It is written in Golang and extends Prometheus by scraping different locations on the filesystem for specific metrics. An example is the location `/proc/vmstart` on most Linux systems, which contains regularly updated system information on memory, paging, running processes, IO, CPU, and disk scheduling.

Node exporter by default supports around 96 different metrics, so a selection is made on which are relevant for our research purposes. We focus on the metrics also considered in existing research, with additionally metrics regarding context switching / thread interrupts. The latter seems interesting to us, as we've not seen these meeasurements in other research. Regarding the outcome of these metrics, not all machines will be considered. Based on the combination of workload (e.g. IO heavy) and metric type, a sensible selection is made and explained later on in the results. Below a brief overview of the considered metrics is listed.

- **Memory user space apps**
  The amount of memory used by processes running in user space. This is calculated by taking the total available memory subtracted by the effective free memory and RAM claimed by the OS. Examples of the latter are caches or pagetables.
- **Memory system total**
  The amount of memory used by the whole system. It is calculated by taking the total available memory, subtracted by the effective free memory, RAM cache memory, and any buffers stored in memory. Basically, reclaimable memory is substracted.

[9] https://github.com/prometheus/node$_{e}xporter$

- **CPU system total**
  CPU metrics are extracted from `/proc/stat`, within the Linux file system. It presents the time that is spent on different types of operations. For the CPU *system* it considers the CPU time spent in the kernel, e.g. performing system calls.

- **CPU user space apps**
  Contrarily, this represents the time that the CPU spends on processes in the user space.

- **CPU IO wait**
  This is the time that the CPU spends waiting for the I/O devices. This can be an interesting metric, as from the existing literature, we know that IO is often heavily impacted by virtualization.

- **Network received**
  The amount of data received on a network interface per given time interval.

- **Network transmitted**
  The amount of data sent through a network interface per given time interval.

- **Context switches**
  The number of context changes that occur during a given time interval.

- **Thread interrupts**
  The number of times an OS scheduler interrupts a thread during a given time interval.

Finally, we clarify the difference between context switches and thread interrupts. Actually, these are two types of interrupts. The difference lies in what happens after the interrupt itself. In both cases, during an interrupt, the current state is stored in a temporary area, which is commonly an OS-dedicated stack. Now, with a thread interrupt, something else might be scheduled, but afterwards, the scheduler will make sure the thread returns exactly where it left off. In contrast, for a context switch, the location of that stack and any extra state information are stored elsewhere in the thread itself. After the scheduler returns to its original operations, it returns to the point of execution where the newly switched thread was interrupted the last time a context switch occurred.

This means that a context switch is more expensive. For instance, all its registers, instruction pointers, and memory page tables can be totally different. Simply put, with thread interrupts, the OS interrupt handler always makes sure it can quickly continue

with the original thread after an interrupt. Therefore, as its state is maintained, there is generally more trust in the interrupt handler to restore back all the initial registers.

Little research on comparing virtualization performance stresses the effect of context switching and the effects on performance when adding virtualization layers. Therefore, we prefer to include some measurements to see if this might explain any performance penalties that might arise.

Measurements

The measurements are performed as follows. The test starts off with a warm up period of 5 000 requests. As soon as this is finished the test continues, without any interrupt, to perform another 22 000 requests. Jmeter is configured to maintain approximately 20 requests per second, using 3 threads concurrently. The UNIX timestamp right after the warmup period and at the end of the test is stored. These timestamps are used to retrieve the data, per metric, from the prometheus database. This whole test is performed 3 times for each scenario (VM, container etc.) and checked for consistency across the different runs. We mainly look at the second and third run, as the first run might still show some more variability before reaching an equilibrium. If the second and third run show similiar trends, we analyze the retrieved metrics stored during the third run.

Chapter 5

Experiments and results

5.1 Distributed tracing

In this section, the performance measurements for distributed tracing will be discussed. However, first the accuracy measurements described in Section 4.3.2 will be set out. Finally, a detailed analysis of the measurements performed will be given.

5.1.1 Accuracy measurements

For the accuracy measurements, we consider the following two tests. This selection was made based on existing literature described in Section 3:

- Kolmogorv-Smirnov Test
- Anderson-Darling test

We compare these tests by increasing the duration of a so-called *SleepTask* as discussed. The request flow considered here is depicted in Figure 5.1.

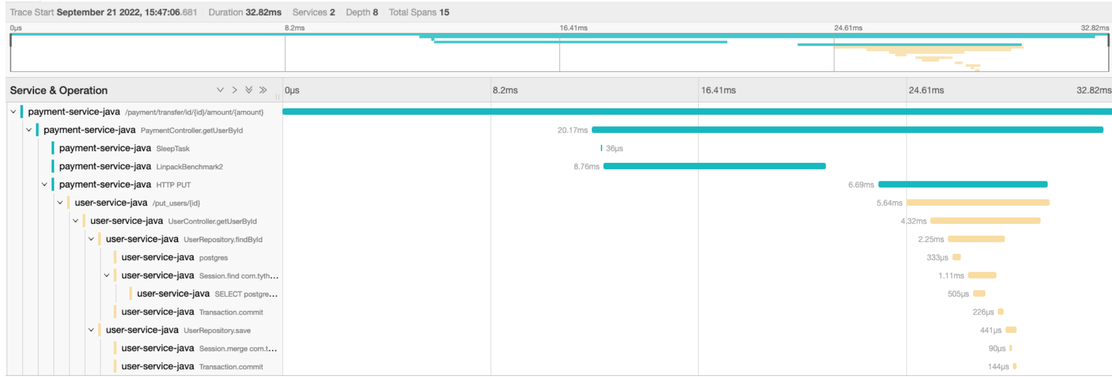


FIGURE 5.1: Request flow for payment-service, including the *SleepTask*

As can be seen, *SleepTask* is the third operation, from the code Listing 4.1 , where for this case the sleep is configured as zero seconds. In this example there is still some minor execution time recorded by our code instrumentation, which is measured as 36 μs. Two reasons can be given for this. First, the *SleepTask* that has been defined is still executed, and the while loop simply immediately ends. Additionally, there might be some inaccuracies here, for instance due to clock drift, which we try to isolate in the upcoming measurements.

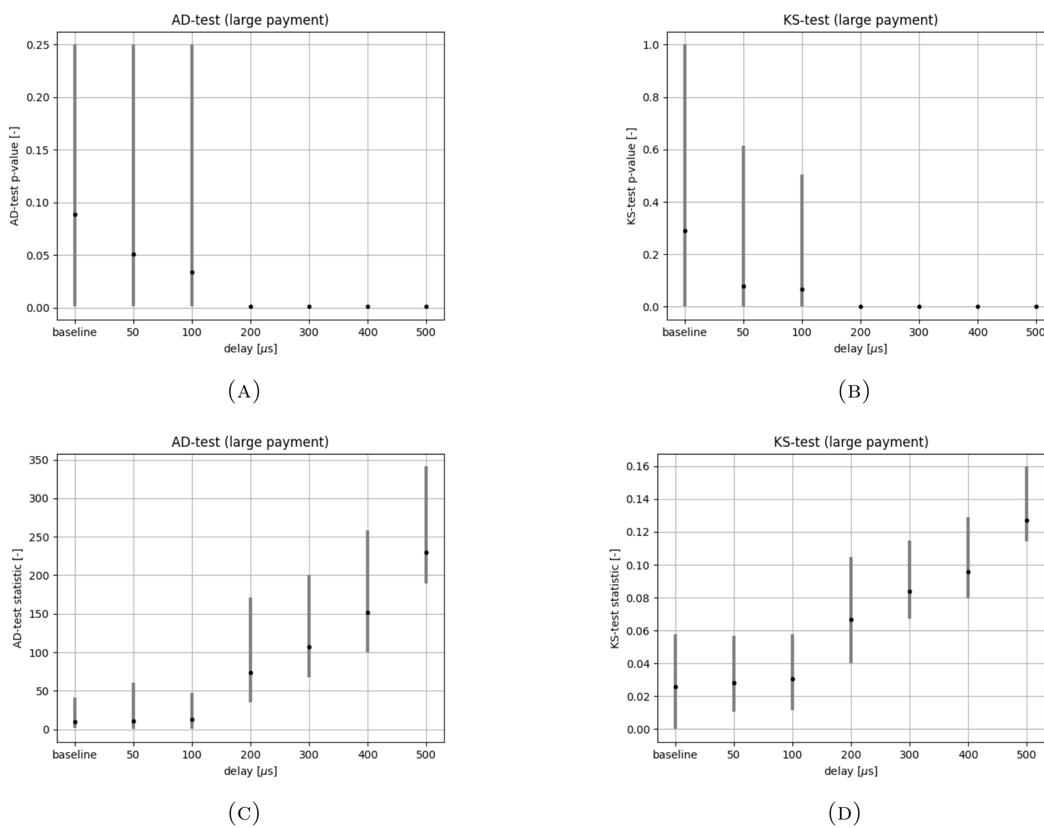
First, to compare statistical tests, p-values can be indicative. As discussed, in the current context, they can be viewed as a measure of how sure we an be whether two sample sets come from the same distribution. In this research, the focus lies on the Kolmogorov-Smirnov (KS-test) and Anderson-Darling test (AD-test). These tests have the best properties for our use case, according to existing research discussed in Chapter 3.

Looking at the calculated p-values for the baseline, we can see, for example, in Figure 5.2a, that the p-values are between 0.001 and 0.25. All sleeps are compared to the ten baseline latency distributions. One thing to note on the implementation of the AD-test used in this research, is its cap on p-values. The SciPy package implements this test with a two-sided cap on the p-value. Its minimum and maximum value will always be respectively 0.001 and 0.25. This means that the most left-hand line represents the p-values of a baseline latency distribution compared to the nine other baseline latency distributions from different runs.

The black dot is the average, while the line represents the range of values of ten runs. This means that with this range, following the p-values, the AD-test does not in all cases fall above any significance level (e.g. 0.01). As discussed in Section 3, the null hypothesis H_0 is that both CDFs come from the same underlying distribution. In this analysis, the null hypothesis is at least rejected for one or more runs. Therefore, based on p-values,

we cannot conclude that all runs come from the same distribution. Consequently, we cannot use p-values to determine if other permuted traces, either by a *SleepTask* or different environment, are coming from the same distribution. We conclude this a the baseline runs, on bare-metal with no sleeps, sometimes result in a lower-bound p-value of 0.001. Unfortunately the same analogy holds for the KS-test which does not always result in a p-value above our significance level of 0.01.

The result for both the KS-test and AD-test can be seen in 5.2a and Figure 5.2b, showing the p-values. These values represent the probability that our null hypothesis is not true, being whether two compared latency sample sets come from the same distribution.



(A) (B) (C) (D)

FIGURE 5.2: The results of the accuracy tests for the larger request flow (*large_payment)*. It shows the p-values for testing the hypothesis whether request flows with a introduced delay in μs come from the same distribution as the baseline (0 μs).The complete trace span measured is the call to the *payment service*.

It is important to be aware that, for these accuracy measurements, on bare-metal, two cases are considered. For ease of wording, we call these *large_payment* and *small_payment*. Both requests are similar to the one depicted in Figure 4.1. However, for *large_payment*, the Linpack benchmark is introduced, which trace takes on average approximately 25 ms. The entire call to the *payment service* takes approximately 27 ms. Now *small_payment* does not include this Linpack benchmark and thus takes approximately 1.3 ms. This

is done to see what granularity of operations (in terms of duration) will be possible to detect any discrepancies with the appropriate statistical tests.

With the information from our accuracy measurements, it can be concluded that p-values of the chosen statistical tests are not suitable for performing the planned measurements on traces. Nevertheless, in Figure 5.2c and Figure 5.2d we can see that the test statistics of both tests give us more insights in where a difference can be detected. For both tests it shows that for sleep tasks with a duration of 200 μs and above, the null hypothesis is rejected for all runs when comparing to the baseline with a sleep of 0 μs. This indicates that the test statistic of both the AD-test and KS-test are able to detect shifts in trace span duration of 200 μs and above.

However, these discussed accuracy measurements depict an optimal scenario. First of all, these tests are ran on bare metal servers. From Section 3 we know that the variability can increase as virtualization layers are added. Therefore, this accuracy could still get worse for the actual virtualization measurements. For the second point, we look at another scenario tested for accuracy; the *small_payment*. The same tests are ran, but then for the second scenario with only a very short Linpack benchmark. The same results for the smaller request flow can be seen in Figure 5.3.

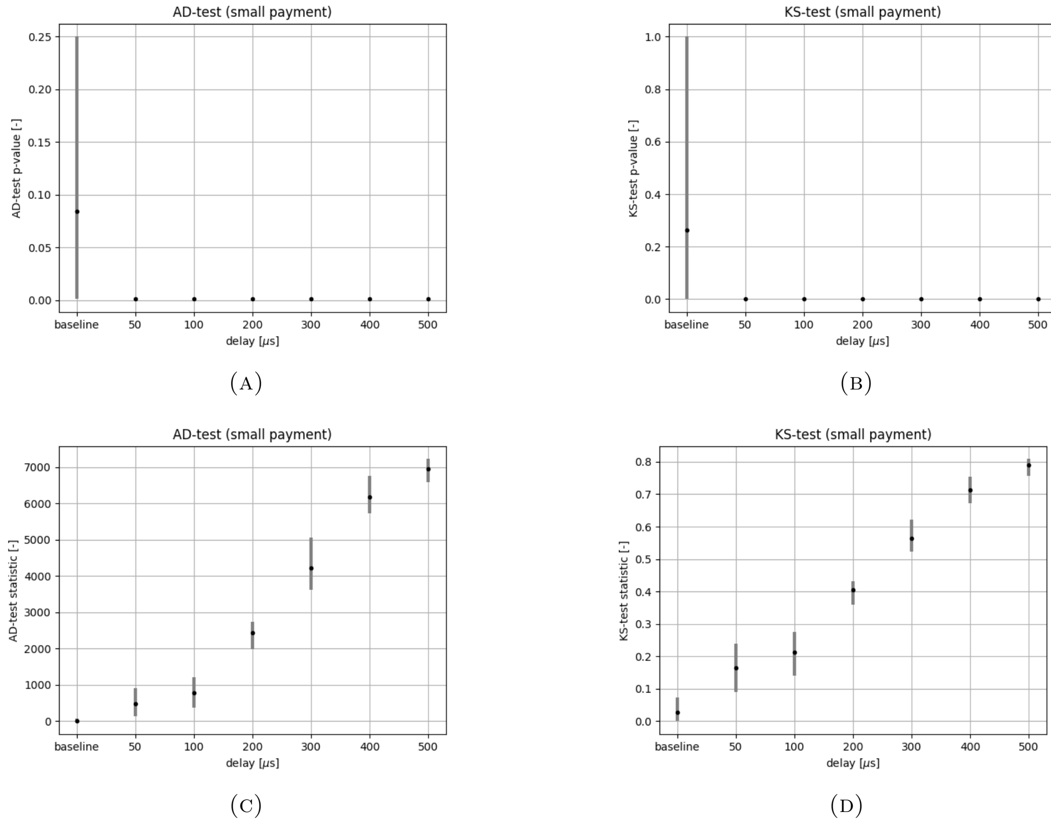


FIGURE 5.3: The results of the accuracy tests for the smaller request flow (*small_payment)*. It shows the p-values for testing the hypothesis whether request flows with a introduced delay in μs come from the same distribution as the baseline (0 μs). The complete trace span measured is the call to the *payment service*.

As can be seen in these figures, the p-value seems to be more accurate and only shows p-values above our significance level of 0.01 in the baseline comparison. This means that for the introduced sleeps, on the x-axis 50 to 500, the tests claim that no run comes from the same distribution as the baseline runs with 0 μs sleeps. Additionally, the test statistics also turn out to be much higher, in the range of 10^3. As the static tests measure the distance between the latency CDFs, this is intuitive. As an example we look at the sleep of 50 μs. This delay becomes more visible in the CDF if the complete request flow (of the payment service) only takes 1.3 ms instead of 27 ms on average. The relative difference introduced by the sleep task is greater, and therefore the statistical test is more sensitive to this change.

The latter proves our second point of why these accuracy measurements are merely specific to these trace spans. There is no appropriate way to set a significance level on the test statistic, as it is highly dependent on the operations contained in the trace span. This means that if actual changes in trace span durations were to be measured, the significance level of the test statistics should be determined dynamically, based on the baseline measurements.

5.1.2 Data exploration

Being more aware of the accuracy that distributed tracing can offer us, we now look at the actual trace spans that were collected from our SUT. For the measurement of the complete distributed system its performance, we take a similar approach as before. Just like the benchmarks above on accuracy of the statistical tests, we again perform 10 runs of tests with 10000 requests each. The latter number excludes the 5000 warm-up requests for each run. This flow is identical to the flow described earlier in the methodology (Figure 4.6).

Before looking at the statistical tests and performance of different virtualized environments, some data exploration is done to see whether the visualizations correspond to the set expectations. Latency distributions, for instance for calling a single service that interacts with a database, are expected to look like Figure 3.7. This means looking similar to a log-normal distribution but with a clearer minimum and a heavily skewed shape due to its long tail. The figure represents the duration of calling the *user service*, that only interacts with a local database, obtained as approximately 10 000 tracing samples.

As discussed in Section 2, it is not useful to summarize a set of latency measurements by commonly used characteristics for distributions. As a result of the skewness, the mean and standard deviation are less meaningful. This also includes common histogram plots of the latency values obtained, as shown in Figure 5.4. Although it clearly depicts the shifted average value, it introduces the risk of discarding information about the data. Specifically related to the longer tail, which is not properly visualized in this case. The values between 25 and 30 ms are barely visible in this figure.

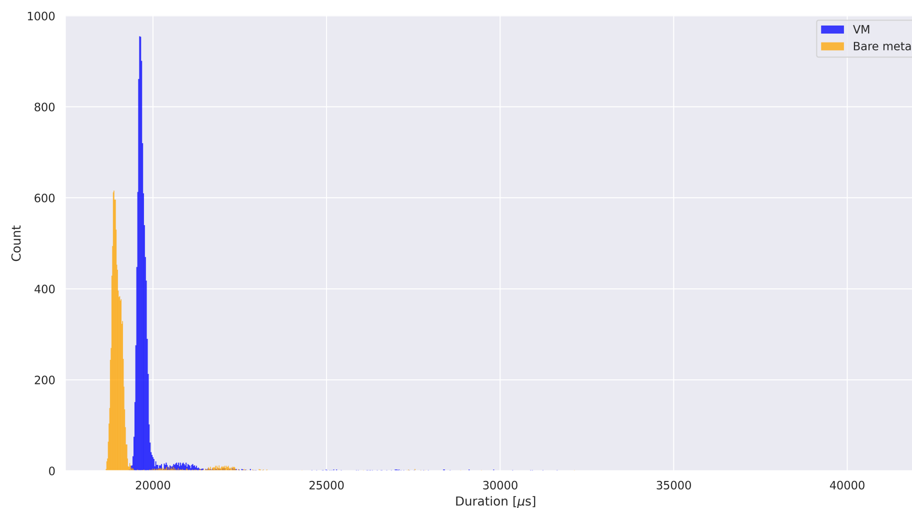


FIGURE 5.4: Example of a typical trace duration distribution plot for one run in a virtual machine and one on bare metal.

As it is more interesting to look at percentiles, we propose a different way of visualization, where the x-axis is a logarithmic axis showing percentiles. For instance, all values indicated on the left side of the 90th percentile fall within this percentile. The same two runs shown in Figure 5.4 are shown as percentiles in Figure 5.5 below. So, these two figures show the exact same data samples.

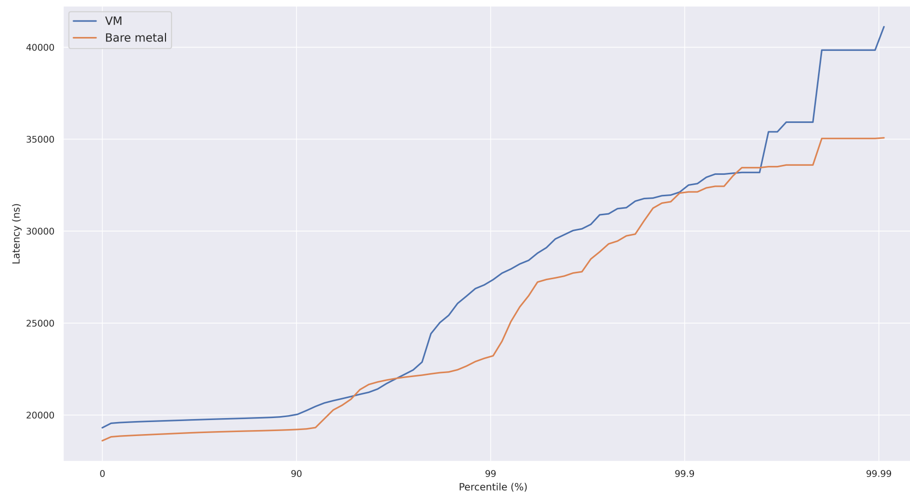


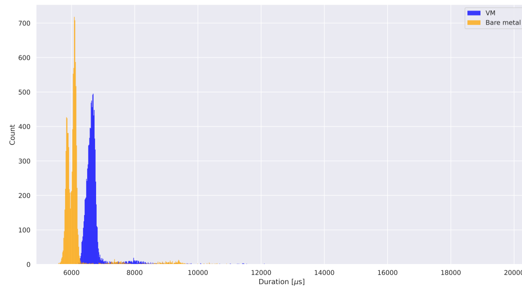
FIGURE 5.5: A plot representing the same data samples as Figure 5.4, focusing on percentiles.

In both figures, it is visible that the bare metal run results in lower latency on average. Furthermore, both figures do show that, between 20 and 25 ms, bare-metal has a heavier tail. However, what Figure 5.5 clearly reveals, contrary to Figure 5.4, is that the maximum outliers for VMs are higher than the maximum for bare metal. The highest values are approximately 35 and 42 ms, for bare metal and VM respectively. For the purpose of this research, to determine performance across all measured values (and not a percentile), the plots showing percentiles are more relevant.

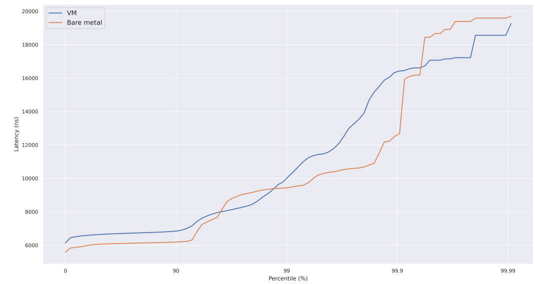
As mentioned above, the plots in Figure 5.4 and Figure 5.5 show the duration of a user-service call, measured from the server where this service runs. This part only performs a small Linpack benchmark, and a single local database transaction.

However, the first thing that came forward during data exploration is that distribution shapes can be different when measuring operations that involve remote networking. This can be seen in Figure 5.6a and Figure 5.6b. In Figure 5.6a, it can be seen that two clear maxima appear in the distribution plot for the bare metal server. This is also visible in the percentile plot in Figure 5.6b, but requires careful interpretation. Our assumption is that these maxima represent two common latency averages affected by the Linpack

benchmark and user service call, both contained in the request flow. On the contrary, the data for the VMs did not show this behaviour. The expectation is that this is due to the higher variability in VMs, as we have seen in Section 3, pointed out by existing works. Therefore, the peaks might fade into each other.



(A) Distribution histogram.



(B) Percentile distribution plot.

FIGURE 5.6: Two plots depicting how two peaks of sampled data occurences fade into each other when running the same benchmark on virtual machines.

5.1.3 System benchmark

In this section, we will provide a deeper dive into the performance tests on the distributed system. First, another validation on the consistency of the results will be established. Afterwards, a broader view on the test statistics will be given. Finally, we look at the actual analysis of the distributed systems, and the performance differences that have been found, if any.

Consistency

We take a quick look on the consistency of our measurements, this to verify whether we can establish some certainty of our results and draw meaningful conclusions. A percentile distribution plot of 5 runs for both the bare metal and VM + container scenario can be seen Figure 5.7 below.

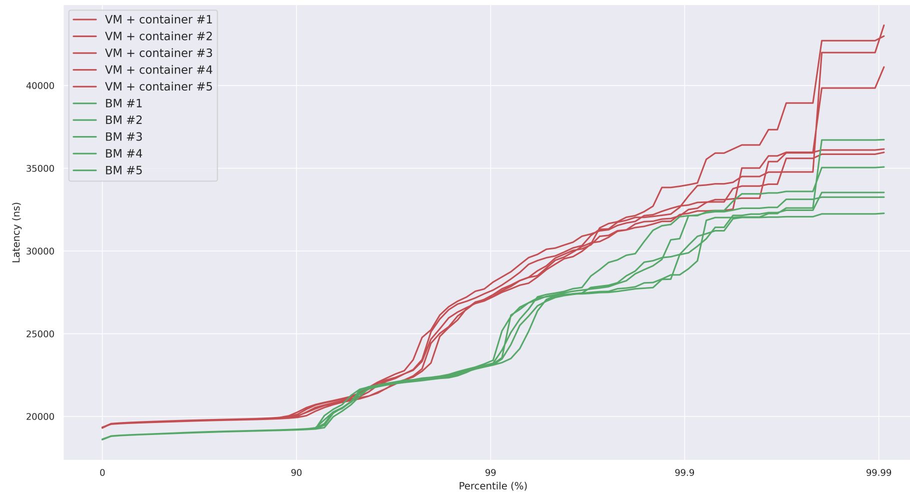
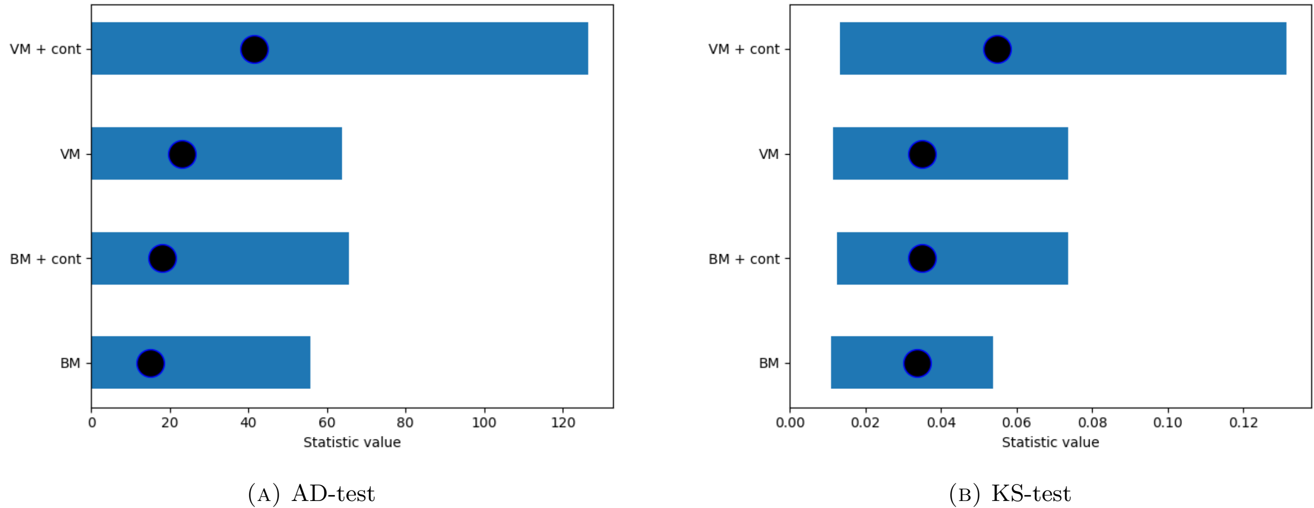


FIGURE 5.7: Example of 5 runs for both Bare metal and Containers for the *payment service*. This figure shows the consistency in lower percentiles, but also the variability in higher percentiles.

As can be seen, specifically in the 90th percentile, there is significant similarity between runs. This implies that the results are reproducible and somewhat consistent. Stating 'somewhat consistent', as generally latency measurements lack consistency and have high veriability. However, taking a closer look between the 90th and 99th percentile, it does look like out fourth scenario, a container running on top of a KVM, does have more variability amongst different runs. This is supported by the statistical values of the Anderson-Darling test and the Kolmogorov-Smirnov test in Figure 5.6a and Figure 5.6b. These figures indicate the consistency of the measured environments for a call to the payment service. Both statistical tests were performed against all other samples from the same environment. So, every run is tested against 9 other runs for that environment (e.g. KVM + container). The resulting test statistics are aggregated, representing the minimum, maximum, and average values. Again for these results, we have found that



(A) AD-test

(B) KS-test

FIGURE 5.8: The test ranges of the test statistic value for the four scenarios. Each run is tested against all other 9 runs in the same catagory, depicting the distance between their CDFs.

Regarding the interpretation of these graphs, the higher the statistic for a given scenario, the higher the variability between runs in the same environment. It is good to emphasize that these graphs purely present variability for four scenarios and thus how consistent their results are. In case the p-values of these tests were not too low to work with, the higher the p-value would be, the lower the statistic. A simple analogy is that the higher the test statistic, the more distance there is between percentile plots (CDFs), like the ones shown in Figure 5.5.

In Figure 5.8a and Figure 5.8b the black dot shows that Bare Metal machines, on average, show the least variability between different runs. For the bare metal + container scenario and KVMs, this gradually increases. Remarkably, for the dual virtualization scenario the variability increases significantly. The maximum statistic is about twice the number compared to other scenarios, relative to its own average. Both of these findings correspond to the findings of Kozhirbayev et al. They show that variability in networking/CPU increase significantly with the "thickness" of virtualization layers added. This implies that web applications running in containers on top of virtual machines suffer from a significant increase in variability. Finally, another thing that stands out again from the figures above is that it seems that the Anderson-Darling test is more sensitive to differences between runs. Comparing the maximum statistic to the average statistic, between the two tests, it shows that the AD-test is more sensitive to changes in the trace span duration for this specific case.

These figures also confirm the conclusion drawn in the previous chapter. There is no one-size-fits-all when it comes to selecting a significance level for the test statistic. This

even differs amongst our four scenarios, measuring the same trace span.

SUT measurements

Now that the reader is more familiar with the statistical approach to detect systematic differences in the measured trace durations, we look at the tests performed in the distributed system defined in Figure 4.1. The first way we compare all traces in different environments is by looking at the average shift in latency on different percentiles. For the SUT desribed in Section 4, we consider eight traces, also listed in Subsection 4.1.5.

We present the result of the trace analysis here, showing tabulated percentiles. This section will discuss these traces step by step, presenting our interpretations and drawbacks of the results.

First, we show the trace span data of the *order service*, which is the entry point of our distributed web application. The duration of these spans represents the complete call to the web services, from the server its point of view. Table 5.1 shows the duration of this trace span, with values per percentile. The trace durations are from the best-fitted run, determined by Algorithm 1. Additionally, the test statistics are included for both the KS-test and AD-test. Note that the AD-test statistic goes to -1.31 due to the implementation, if identical sample sets are compared.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	33206	33411	37499	107844	693440	1963993	-1.31	0.0
Container	34650	35161	40888	107906	837359	1904501	6284.89	0.8532
Virtual Machine	61989	68409	86429	120939	617645	2362824	8167.42	0.9386
VM + Container	64797	72265	126538	143088	638826	2328454	8189.18	0.9395

TABLE 5.1: Overview of percentiles and AD test for the order service span in μs.

First of all, we'd like to make an important note on the results shown here, which affects most of the tracing data. There's a significant difference between the scenarios with and without a virtual machine, which is also depicted in the percentile plot in Figure 5.9. It can be seen that in those cases, these scenarios result in almost twice the latency, where the P_{50} latency for VM and VM + Container increase by 87 % and 96 % respectively. It is good to be aware that our SUT, in the *order service*, involves heavy disk IO,. As discussed in Section 3, the disk IO performance is heavily effected by the virtualization layer of a virtual machine. Therefore, it should be taken into account that the absolute difference in latencies is expected to be mainly due to the heavy IO operations in our system. These differences might not be a good representation of a real-world web application having a synchronous flow.

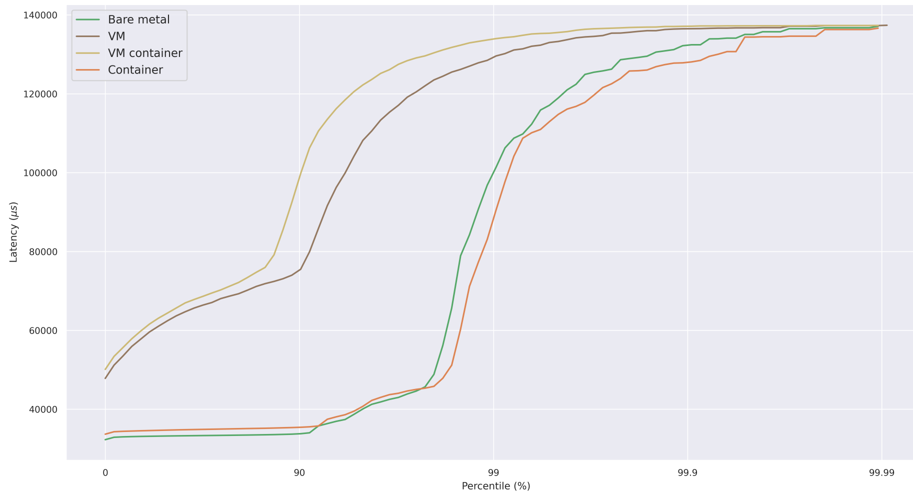


FIGURE 5.9: The percentile distribution plot of trace span durations for the *order service* call.

Some other points that stand out from Table 5.1 and Figure 5.9, are listed below.

- As mentioned before, the P_{50} latency for VM and VM + Container increase by 87 % and 96 % respectively.
- Interestingly, the difference between the VM and VM + Container increases in the higher percentile. While latency increase between VM and VM + Container at P_{50} is only 4.5 %, at the P_{90} this is 46.4 %.
- Compared to bare metal, containers show slightly higher latency across the percentiles, although be it only 6.4 % ($\pm$ 400 μs).
- Finally, the VM and VM + Container scenarios are significantly steeper in the lower percentiles (Figure 5.9). This implies that, besides average latencies, VMs severely increase the latency variability.

In order to break down these results, we look at more fine-grained trace spans, starting with disk IO related traces. Unfortunately, as will be shown, the file interactions of our SUT affect some other tracing measurements. It later follows that some measurements are therefore not representative for actual performance implications.

First, we examine the small files task. As discussed above, within this task, the application writes and reads small files of approximately 140 bytes to the disk. The results of the trace spans are shown in Table 5.2 below.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	1425	1457	1889	2268	4169	655078	-1.31	0.0
Container	1968	2395	2995	3897	114463	1177249	7309.67	0.8749
Virtual Machine	28179	34609	40023	71805	99185	638011	9819.9	0.9941
VM + Container	29577	36864	82472	99299	112708	452075	9816.43	0.9941

TABLE 5.2: Overview of percentiles and AD test for the *SmallFileTask* span in μs.

What can be noticed at first glance is the significant delay that is being introduced when a VM is added to the virtualization layers. This delay here shows how heavily the complete application's latency is affected by disk IO operations. As can be seen in Figure 5.10, the shape of both 'thicker' virtualization layers (VM and VM + container) follows the shape and relative difference of the same latency distributions at the *order service* (Figure 5.9). This implies that this operation might be a good cause of the severe virtualization overhead we have seen for the complete API call (order service). The response time of our complete system increases by approximately 28 milliseconds, where this task can mainly be accounted for. It should also be noted how the test statistics increase with how different the percentiles maximum values are, indicating the tests do function as intended.

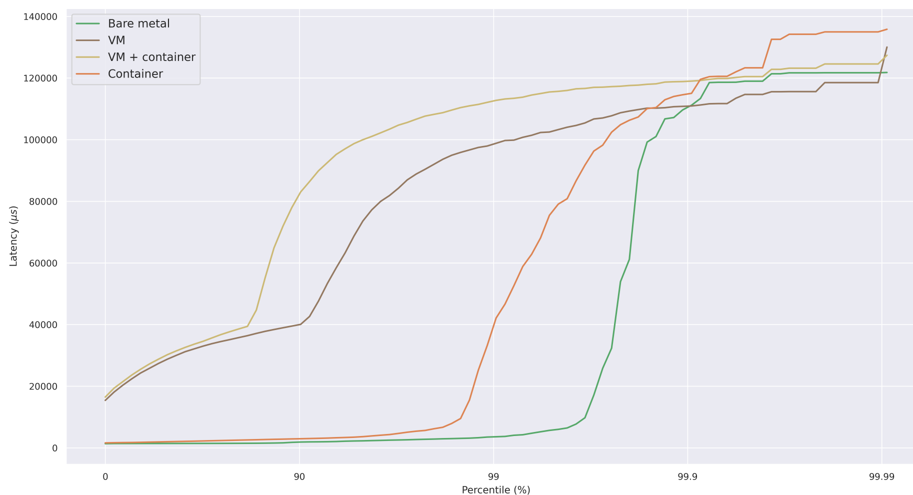


FIGURE 5.10: The percentile distribution plot of trace span durations for the *SmallFileTask* task.

Additionally, a minor difference can be seen between containers and bare metal, with a higher variability for containers. The differences in the 90th percentile seem insignificant, but is still an increase of more than 50 % for the highest values. Specifically in the 99th

and 100th percentile (all values), we see that there are very high outliers within the latency distribution for the container scenario.

We now consider the other two disk IO related tasks, namely for large files. Starting with the task which reads a large file (approximately 2.4 MB) from the disk. The results for this trace span can be seen in Table 5.3 below.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	3725	3748	3786	4074	5505	25987	-1.31	0.0
Container	3337	3352	3409	3671	5625	13700	8578.83	0.9478
Virtual Machine	3755	3780	3856	3997	4794	17131	2056.51	0.3845
VM + Container	3294	3322	3475	3640	5080	10974	8748.62	0.9521

TABLE 5.3: Overview of percentiles and AD test for the *LargeFileTask.read* span in μs.

These results correspond with the conclusions drawn in existing literature. For larger files, the overhead of virtualization is minimal. However, a remarkable result is that even amongst 10 runs of 10 000 requests, the container setup seem to outperform bare-metal. Even when running on top of a VM, the container + VM scenario outperforms a VM interms of trace duration. Nevertheless, this difference is around 400 μs. In the accuracy measurements for our current testing method, it shows that each test can have an inaccuracy up to $\pm 200 \mu s$. This means that the difference between bare metal and containers for this trace span is within our error boundaries (200 for each test).

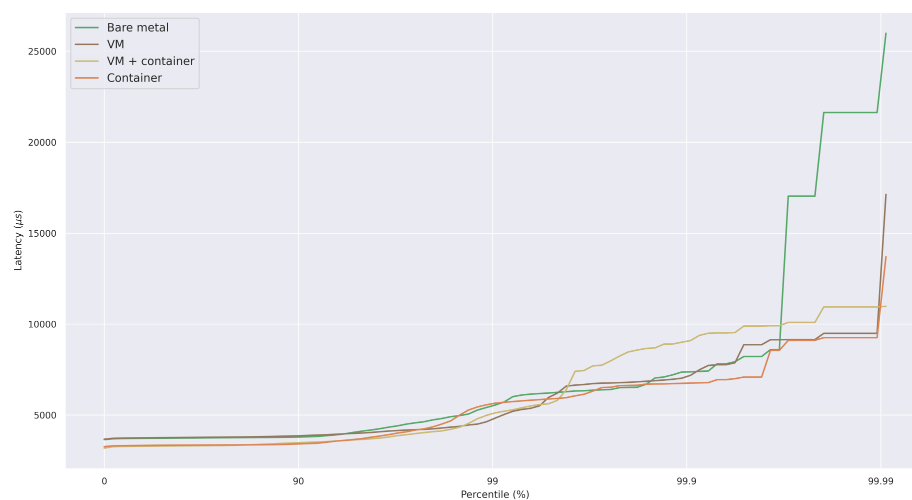


FIGURE 5.11: The percentile distribution plot of trace span durations for the *LargeFileTask.read* task.

Nevertheless, based on other measurements showing significant consistency and the way we select the "best fit" trace to analyze here, blaming this on measurement errors is too blunt. Furthermore, in the latency distribution plot of this task, the 99.99 percentile shows very high outliers for the latency of bare metal. Within this research we have rarely seen such discrepancies in our measurements, presumably due to the many repetitions of experiments / requests. Some other factors might be present affecting these measurements and therefore these discrepancies should not be treated as measurement errors. As discussed before, based on theory, it is unexpected that any scenario might outperform bare metal. Such phenomena should be treated with care, and a more thorough analysis is recommended.

As the last IO heavy task considered, an operation writing a large file to disk is analyzed. This task writes the same file content of approximately 2.4 MB to the disk. The trace span durations and test statistics can be seen in Table 5.4.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	1741	1994	2242	2413	2736	424941	-1.31	0.0
Container	1955	2025	2145	2451	94399	534914	3474.07	0.6295
Virtual Machine	1508	1589	1733	1940	2273	478054	2824.52	0.5145
VM + Container	2084	2214	2551	2764	3046	231683	4570.9	0.656

Table 5.4: Overview of percentiles and AD test for the *LargeFileTask.write* task span in μs.

The container scenario shows a remarkable increase of latency around and above their 99th percentile, being more significant for the container scenario. This is also depicted in the latency distribution plot shown in 5.12. The container scenario show an increase of respectively 4700 %, comparing P_{50} to P_{99}. For bare-metal this is only 57 %. As discussed in Section 3,

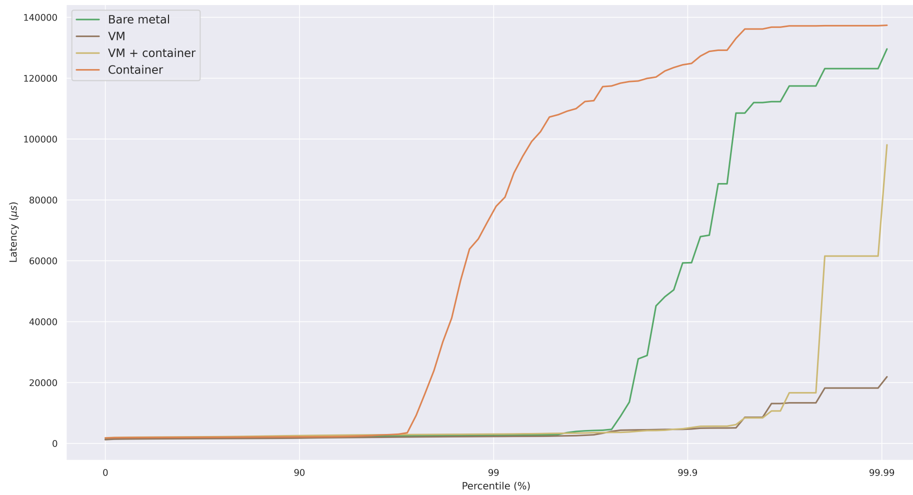


FIGURE 5.12: The percentile distribution plot of trace span durations for the *LargeFileTask.write* task.

Overall, looking at the lower percentiles (e.g. P_{50}) no significant overhead can be seen. Where a VM does not incur any overhead compared to bare metal, the VM + container scenario does. However, we can not draw any conclusions from this given that we use Docker its filesystem in that case. We have not found an explanation for the high variability (P_{99} and above) for bare metal and containers. Even while checking older measurements for verification the same discrepancy occured, which does not correspond with existing research on these scenarios.

Other services

After the previous discussion on disk IO and how it affected the performance of our system as a whole, we now set out the other service trace spans, not directly involving disk IO. The percentile overview for the payment service call can be seen in Table 5.5 below. As discussed before, this complete operation only involves a (local) database call and a call to the remote user service.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	6057	6114	6183	7444	9485	18140	-1.31	0.0
Container	6434	6498	6586	7702	9818	19774	7358.96	0.8428
Virtual Machine	6253	6331	6397	6462	7815	12773	4599.06	0.6116
VM + Container	6614	6702	6839	7835	10054	19489	7976.1	0.9248

TABLE 5.5: Overview of percentiles and AD test for the payment service span in μs.

Here we directly see a remarkable result, the VM scenario appears to be slightly faster than bare metal. This is unexpected, as we know from existing literature that VMs commonly introduce a big performance penalty, due to the virtualized network interface card.

Our expectation is that this result is also affected by the IO operations of another service, slowing down the application. The workload generator tries to maintain a constant number of requests per seconds entering the SUT. However, within this complete request flow significantly more time is being spent within the order service, due to the heavy disk IO. Therefore, we know that the payment service effectively receives less concurrent requests per second within a VM compared to bare metal. Do recall that we saw before that the complete response time of our system was almost doubled from approx 33 to 65 ms. Due to this shift in where time is effectively spent in our system, we know that the host where the payment service is running has a lighter workload during our test. Therefore, a call to the payment service might finish faster in a VM as effectively less requests are received within the same time span. For the VM scenario, this would then result in less CPU interrupts and thus a faster response at the level of the payment service.

This is unfortunate as it negates the way we try to characterize the distributed system, by distributed tracing. This research is bounded by time, which means we haven't been able to adjust the IO workload of the SUT. However, a quick test for solely VM and bare metal was done, where only the payment service was called, which in turn calls the user service. The latency distribution plot of this test can be seen in Figure 5.13 This means the IO heavy order service was left out of the request flow.

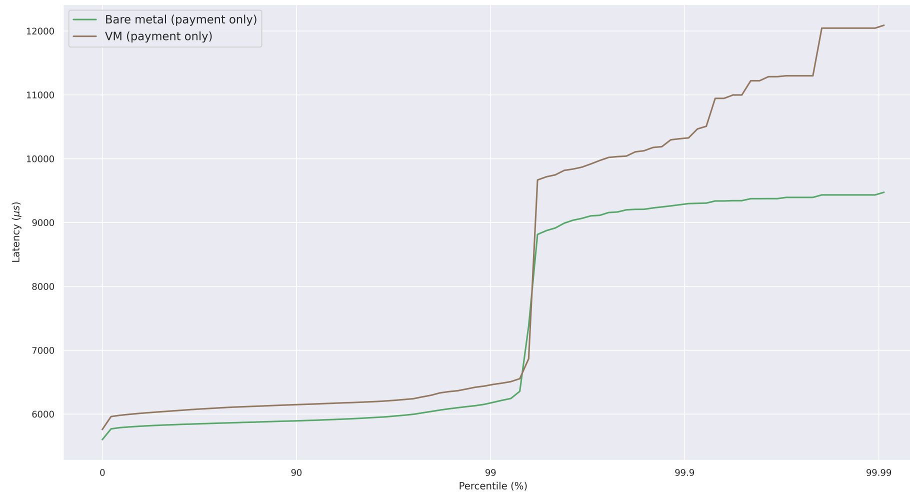


FIGURE 5.13: The percentile distribution plot of trace span durations for the *payment service* call. In this scenario the *order service* was excluded and only the bare metal and VM scenarios are considered.

The most important results of the tracing analysis have been discussed above. We now quickly go over other operations that have been measured. In Table 5.6 the latency percentiles of a PUT call to the user service are shown. At this point, these results seem more intuitive and correspond to the findings in existing research on this topic. For visualization purposes and consistency, the percentile distriubtion plot has been included with Figure 5.14.

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	438	467	486	496	542	791	-1.31	0.0
Container	479	501	522	538	582	1478	1895.91	0.3598
Virtual Machine	519	563	589	605	654	1069	3508.5	0.5489
VM + Container	551	604	638	657	725	2392	4715.23	0.6403

TABLE 5.6: Overview of percentiles and AD test for the (PUT) user service span in μs.

These results show an outcome which was initially expected, also considering at the statistic values for both the AD and KS test shown in the table. The statistical tests are capable of detecting the minor differences between the four scenarios. Furthermore, the (minor) latency increase, becomes higher with the 'thickness' of the virtualization layer. However, we should take into account that we've seen the performance of virtualized services might look better for VMs due to the significant delay introduced at the order-service level. Therefore the corresponding results, as depicted in Table 5.6, only show a

'best case' scenario, where with less IO affecting the same operation, a larger performance penalty might occur. For this specific trace span, it is expected that these trace spancs are less affected by those IO operations due to the short execution time, causing less thread interrupts.

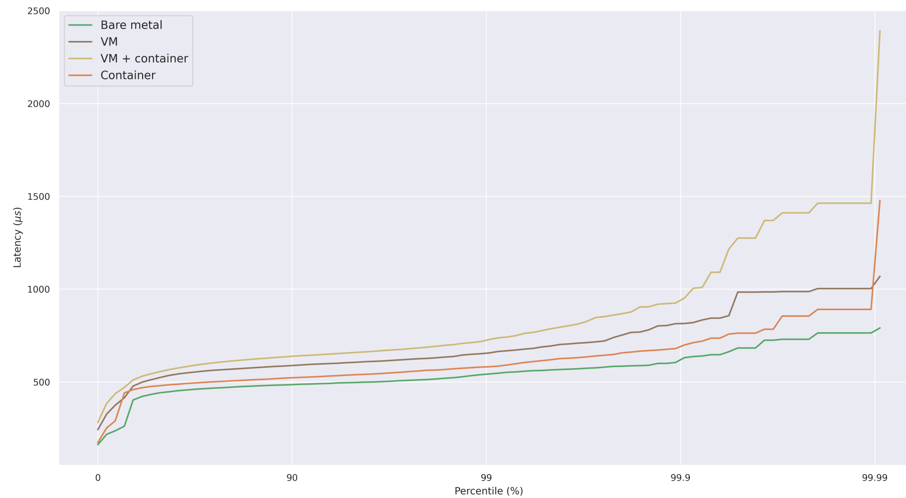


FIGURE 5.14: The percentile distribution plot of trace span durations for the *put_user* call.

Other trace spans mentioned in Section 4.3.2 will not be set out in this chapter, as they would not provide more insights than already presented. The remaining distribution plots and percentile tables can be found in Appendix A. We believe any relevant tracing results have been set out, and leave further interpretation for our discussion in Section 6.

5.2 System-level metrics

In this section we go over the system-level metrics in Section 4.3.3. After presenting all metrics with their average and standard deviation per tested scenario, we discuss the outcomes one by one. An overview of the selected results can be seen in Table 5.7 below. For most metrics, we consider the *payment service* properly representing a common application. It has some small CPU load by the Linpack benchmark, and a call to the *user service*. The latter also performs a small database interaction. However, for some metrics, we include *order service* as well. This serivce, with its heavy IO operations, might be able to reveal different properties of the system.

Metric	Bare Metal	Container	KVM	KVM + container	Service
User space apps memory [GB]	6.48 ± 0.00	6.42 ± 0.00	8.06 ± 0.00	8.83 ± 0.01	Payment
Total used memory [GB]	6.57 ± 0.00	6.52 ± 0.00	8.16 ± 0.00	8.93 ± 0.01	Payment
CPU system [%]	0.96 ± 0.05	1.21 ± 0.06	2.41 ± 0.05	2.58 ± 0.08	Payment
	14.94 ± 0.27	20.25 ± 0.88	42.31 ± 2.51	47.83 ± 4.57	Order
CPU user apps [%]	23.20 ± 1.17	21.07 ± 1.03	24.26 ± 0.55	26.18 ± 0.79	Payment
	19.31 ± 0.86	21.74 ± 1.31	114.09 ± 26.13	145.05 ± 34.24	Order
CPU IO wait [%]	0.04 ± 0.03	0.01 ± 0.01	0.01 ± 0.01	0.03 ± 0.03	Payment
	36.38 ± 5.82	51.4 ± 10.92	23.23 ± 1.08	24.38 ± 1.96	Order
Network received [MB]	0.29 ± 0.02	0.27 ± 0.02	0.30 ± 0.00	0.3 ± 0.01	Payment
Network transmitted [MB]	1.04 ± 0.04	0.98 ± 0.06	1.11 ± 0.02	1.09 ± 0.03	Payment
Context switches [-]	1197.81 ± 80.60	1236.92 ± 61.3543	2504.1 ± 38.89	2730.44 ± 63.32	Payment
	3741.95 ± 53.69	13429.65 ± 891.32	14288.95 ± 606.89	16957.93 ± 1025.38	Order
Thead interrupts [-]	1577.97 ± 88.51	1618.31 ± 71.96	2220.88 ± 29.94	2434.17 ± 67.22	Payment
	6921.10 ± 356.46	12611.56 ± 584.8	12710.17 ± 1263.30	15634.66 ± 1585.25	Order

Table 5.7: Overview of the results from the system-level metrics analysis, exported with Prometheus node exporter. All results are averaged or aggregated over a 1 minute timespan.

The metrics are discussed and listed below one by one.

- **User space apps memory**
  The memory used by applications in the user-space is approximately the same for the bare metal and container scenario. Even so, bare metal shows a slightly higher memory usage. However, this is only 60 MB and we expect other processes running on the system to account for this. Naturally, although we have tried to minimize this effect, background processes in Linux are always present. We should account for this with other metrics as well, and discard minor differences. Nevertheless, it can be concluded that the Docker runtime does not claim a significant amount of memory. Contrarily, both VM scenarios show a significant increase in memory usage of up to 2 GB of memory. This shows that the virtualization layer causes more workload on the system for the same operations, leaving less room for the actual applications running on it.

- **Memory system total**
  In every scenario, approximately 100 MB of memory is in use by applications in the kernel space. A very consistent difference can be seen across all four scenarios.

- **CPU system**
  For this metric, two servers are considered. Where either the *user service* or *order service* is running. This choice was made because the CPU is more challenged during IO operations. For the payment service, we only see minor differences in system CPU usage. We expect the higher percentage of CPU usage, and thus the number of cycles, to simply come from processes managing the virtualized

environments. However, we still have a maximum difference of approximately 1.5 %, which is negligible, especially considering possible inaccuracies. We conclude that even if this indicated a minor number of extra CPU cycles, it would not have a noticeable effect on the application's performance.

For the heavier IO operations within the *order service*, this is different. Here it is clearly visible that IO load heavily increases on a VM, which also holds for the container scenario. For the VM and VM + container scenario, we see that the CPU percentage increases from approximately 15 % to over 45 %. This clearly shows the overhead of the virtualization layer when performing IO operations. Note that this CPU usage only shows operations that are performed in the kernel space.

- **CPU user space apps**
  We won't go into more detail about the *payment service* host its CPU usage in the user space. As can be seen, the results arq quite similar, with a slightly higher trend of a few percents for the VM scenarios. However we consider this negligible. However, again for the *order service* a very clear difference can be seen. The CPU percentage in this space increases from approximately 20 % to 114 % for KVM and even 145 $ for KVM + container. Now this clearly shows the CPU experiences quite a number of extra CPU cycles to perform the same operations in those scenarios as on a bare metal machine. We devote this to the fact that every system call has to go through the hypervisor, resulting in far more CPU cycles before the same operation of a file can finish up. On a final note, we point out that the percentages above 100 % are due to multiple cores running within the CPU, where every core can go up to 100 % in a given time span.

- **CPU IO wait**
  We again do not consider the host running the *payment service* showing negligible IO wait times for the CPU. For the *order service*, we notice that higher IO wait times are present for bare metal and container scenarios. This corresponds to our earlier findings, that effectively more disk IO operations occur in the same timespan compared to VMs. This is due to the two scenarios actually performing better.

- **Network received / transmitted**
  We do not consider received nor transmitted network bytes, showing negligible differences for all scenarios. This makes sense as adding a virtualization layer would not incur extra network IO.

- **Context switches**
  For the *payment service* see that the number of context switches per minute are

approximately the same for containers and bare metal, taking their standard deviation into account. However, the VM scenarios do result in apprximately two times the number of context switches. We expect this is due to more switches between calls living in user space / kernel space, coming from the hypervisor. Regarding the *order service*, this follows approximately the same trend. However, containers seem to suffer from a significant increase in context switches. We have not found a specific reason for this, but it does correspond to the higher variability of the IO tasks, which could be a direct cause of these context switches.

- **Thread interrupts**
  We see that thread interrupts follow the same trends as context switching, with no remarkable relative differences.

Chapter 6

Discussion

In this section the results observed from the benchmarks will be discussed, to try and answer the research questions stated in the introduction. In addition, the measurement methods are discussed, including their limitations, and potential for future research.

System under test
First, we discuss the limitations of our tested system. The initial idea was to use a small set of microservices to mimic a lightweight distributed web application. The intention was to apply measurement techniques that might be able to uncover performance bottlenecks that arise within these types of application. This resulted in research on the potential of using tracing for performance measurements of distributed systems.

During the design and extension of these microservices, it was decided to include some typical workloads. This includes operations on different files, short CPU bursts and interactions with a local database. Initially, we figured that this would be representative of a typical web application. However, as realized during the analysis, the IO workloads within a synchronous request flow heavily affect the overall response time of our system. This is a limitation for this research as it negates some potential of our approach to measuring performance.

Accuracy of statistical tests
First of all, this research has carried out measurements to gain insight into what precision can be achieved using code instrumentation. We investigate how powerful the chosen statistical tests can be in detecting differences between actual sets of recorded trace durations.

The results of these measurements show that the calculated p-values cannot be used to measure the extent to which the trace samples come from the same distribution. First, it seems that the results contain too much variability, meaning that there will always

be a slight difference between different runs and their CDF. This means the p-value quickly drops to zero. Additionally, this is not the approach for which p-values are meant. Generally, a p-value describes the probability of rejecting the null hypothesis H_0 that two sets of samples come from the same distribution. This is not a suitable approach in this research. Therefore, it has been decided to use the test statistic of the AD-test and KS-test to measure the distance between two CDFs of different runs. With this approach, we show that the test statistic can be used as a measure of the extent to which two sample distributions are different from each other. Therefore, we conclude that this approach offers the potential to detect differences in the performance of a distributed web application based on trace durations.

Furthermore, we show that the generalizability of the results is limited. The statistic is heavily dependent on the trace span its operations considered and the environment (scenario) in which it was executed. Further research on this could look into a way of generalizing this, by dynamically determining a significance level for the test statistic. This could be done by comparing the baseline bare metal measurements with each other and choosing a safe significance level based on the consistency of the results within the baseline measurements.

For our system, we have seen that in most cases the test statistic was able to detect a difference by a sleep in the source code, with a duration between 200 and 300 μs. However, it should be taken into account that this depicts an optimal scenario. First of all, the benchmarks are conducted on a bare metal server. Adding virtualization layers or IO heavy operations can introduce variability amongst runs within the same test scenario. We have seen that this can make the test statistic less accurate.

Trace span consistency

Prior to diving into the analysis of our distributed web application, we explored some data to establish a basis on how to interpret these results. Typically, latency is presented as a histogram or a moving average. These approaches often leave out important data, such as high latency peaks. We propose a different way by plotting the values of a sample set as percentiles. We show that this reveals more about the actual latency in higher percentiles and does not leave out any data.

Initially, we look at the consistency among the results for a specific call to the *payment service*. In that analysis, we consider 10 runs per scenario. Hereby, we visualize for each scenario the range of test statistics while comparing every run with its other 9 runs for that scenario. In this comparison, we show that the differences between runs of the same scenario increase with each layer of virtualization being added. This corresponds to the findings of the existing literature. It shows that the four scenarios show comparable consistency, although affected by the increase in variability. Furthermore, we confirm

that the AD-test is slightly more sensitive to this increase, based on the average test statistics.

Finally, we should note that these measurements are applied to one specific trace within our SUT. Although it provides a good indication of the level of consistency, this could still differ for other trace spans container other operations.

System under test

For the actual measurements on our SUT, we have presented some insightful percentile tables and percentile distribution plots. During this analysis, the limitations of our system became clear, whereas the measurements were heavily affected by disk IO. The trace span duration of the *order service*, the entrypoint service, has an increased latency at P_{50} of approximately 87 %. We have concluded that most of this increase is due to the IO operations in our system.

Algorithm 1 proposes a way to select the best fitting run within the combination of scenario (e.g. bare metal) and a trace span (e.g. GET call *user-service*). This gives us an easy way of analyzing our results. The algorithm simply selects the run for which the CDF is closest to all other runs, based on the AD-test statistic. We are aware of its drawback, which is that if some runs for a given scenario are significantly off, it is not taken into account. An avenue for future research includes improving this approach to take into account all measurements for the final analysis.

In the following, we present our main reflections on the gathered tracing results.

- Due to the large influence of the IO operations, specifically for small files, not all trace spans are actually useful for measuring performance. However, we can conclude that interactions on small files are heavily affected by running applications in a VM. At P_{50}, the complete request flow takes 33 ms on bare metal and 62 ms in a VM. The difference is approximately 29 ms, where the *SmallFileTask* can be accounted for approximately 26.5 ms of this difference.
- Additionally, we have seen a significant increase in variability of the trace span durations for containers. Specifically, in its higher percentiles. This implies running IO heavy applications in containers, writing and reading small files, would have less reliable performance.
- The credibility of the performance penalties we see in our results is negated by these IO operations. We acknowledge that these types of operations are not representative of a distributed web application that serves synchronous request flows.
- Trace spans that represent operations on large files are more consistent across scenarios. Both reading and writing of large files show a minor difference in span

duration. Remarkably, VMs seem to outperform containers and even bare metal for the same operations in both cases. We've seen this in both the higher and lower percentile ranges, indicating higher variability. In the next bullet point, we discuss a plausible cause for this.

- If we look at other trace spans, we prefer the call to the *payment service*. Its trace is a good representation of a common part within a distributed system, doing some CPU intensive work and interacting with a database. While analyzing the trace span duration, we have seen remarkable results in which the VMs again slightly outperform the container scenario, by approximately 200 μs on a total of 6434 μs. Our conclusion here is that this is due to the limitations of the IO operations. Because *SmallFileTask* takes significantly longer to perform the same operations within a VM, the *payment service* is effectively called less per time unit. Therefore, we reason that more resources are available to the process, per processed request at the payment service. This results in a lower response time for a specific operation, while, in fact, the overall response time of the system increases.

- In order to confirm our conclusion above, we ran a few tests on both VMs and bare metal, excluding the *order service*. This means that we sent requests directly to the *payment service*, excluding heavy disk IO operations. In that case, KVMs turned out to have significantly longer trace spans for this service. This confirms our statement that the disk IO affects the performance of other services. Unfortunately, due to time limitations, we have not been able to expand these measurements to other scenarios.

- Finally, we show that in other trace spans analyzed, the same phenomenon occurs. Unfortunately, this invalidates the trace date collected from our SUT for drawing further conclusions about performance. Nevertheless, we believe to have shown that there is potential in using this way of measurement on a distributed system.

Initially, round-trip response time (RTT), measured from the load generator, was also taken into account. However, due to the heavy impact of disk IO on our system its performance, it has been decided to leave this out of scope. These measurements did not provide us with any new information. Comparisons of RTTs with the duration of the complete *order service* calls have been considered. This could still have given us insight into how virtualization layers affect RTT. However, Jmeter only measures the response time in milliseconds, which would result in too many measurement errors.

System-level metrics

Finally we consider the system-level metrics, measured with Prometheus node exporter. The tool is generally used for monitoring only; however, we use it by aggregating a selection of metrics over time. We use the average value per minute and included the standard deviation of the complete run. The latter is included to see if there is a significant number of spikes.

In the following, we present our main reflections based on the system-level metrics of our SUT.

- We have not seen containerized applications that use significantly more memory within the user space. However, VMs result in a memory overhead of up to 2 GB, related to the hypervisor.
- No significant differences were found in the usage of system memory, which considers processes running in the kernel.
- For the *payment service* we did not see any significant differences in the CPU usage of the system. However, for the *order service*, with heavy disk IO, the CPU usage increases severely with 'thicker' virtualization layers. For VM + container this increases to up to 45 %. This corresponds to our tracing measurements, where significantly more time is spent on the IO operations. The same analogy holds for CPU usage in the user space, where this percentage can even go up to 145 %. Again, we know from the existing literature that every call to the host system its kernel, has to pass through the hypervisor. This includes accessing on-disk files.
- CPU IO wait times shows the same trend as above, except for containers. They showed a significant increase for the host running heavy disk IO. This corresponded to conclusions drawn from tracing measurements, where effectively more requests are handled in the same time span. This means more IO operations and therefore also more IO wait times, which increases per file that is being handled.
- Network bytes received and transmitted did not show any significant differences, which is expected with the nature of our SUT. Only relatively request bodies are

being sent over the network. These sizes should neither increase nor decrease in different environments.

- It was shown that VMs cause about two times more context switches compared to the service running on bare metal. This increased from approximately 2000 to 14000 context switches on average per minute. We expect that this is due to the increased number of calls between the hypervisor and the OS, resulting in more calls from the user space, where the hypervisor is running, to the kernel space. Containers also showed a significant increase up to 13000 context switches per minute. This could be a reason for why we have seen an increase in variability with container scenarios.

Altogether, we have presented the main findings of this research, including its limitations and drawbacks. Eventually, this study has not been able to clearly expose the overhead of virutalization layers within a distributed sytem. Nevertheless, we believe our work provides valuable insights for future research, showing that there is potential in measuring performance by using distributed tracing. The system-level metrics have provided us confirmation on earlier drawn conclusions from the tracing data. Additionally, it can give an idea into what increase of resource usage can be expected by adding different virtualization layer. The next chapter will summarize our conclusions and provide any avenues for future work on these topics.

Chapter 7

Conclusion and future work

This section provides a final overview of the main findings of this study. This includes any recommendations for further research required to establish answers to our research questions. We also discuss whether and how we answer these questions in our study.

In the following, we reiterate our research question and discuss how these were approached.

- **What is the effect on performance when moving web applications from bare metal servers to containers in a highly distributed system?** This study aimed to describe how distributed web applications perform in different virtualized environments compared to a bare metal baseline. The choice has been made to approach this using system-level metrics and code instrumentation. For these tests, four scenarios were considered. Bare-metal as a baseline, where the other three scenarios included container on bare metal, KVMs and containers on a KVM. Unfortunately, the setup of our tested application introduced some complications that have exposed certain caveats when applying this method. We will set out any drawbacks/findings within this chapter.
- **What metrics can be a good indication of how the performance of a distributed is being affected?** Based on existing work on this topic (Section 3), we have listed a set of system metrics that are often considered for measuring the performance in different virtualization scenarios. These have proven to provide valuable insights in what performance measurements can be expected while transitioning towards a virtualized environments. Nevertheless, we reason that conventional benchmarks are not suitable for the distributed manner in which most applications are implemented with the rise of cloud native systems. This

is where the next research question will provide a different angle on measuring performance differences in a distributed system.

- **Does code instrumentation (tracing) offer a suitable way to measure the performance of distributed web applications?** A non-conventional approach was chosen and a distributed application was benchmarked, using distributed tracing. We propose this different approach to allow companies to take into account the properties that come with these types of application. We have been able to demonstrate the potential in measuring the performance of these systems using tracing. However, the results should be interpreted carefully due to certain ill-suited properties of the tested system.

In the following, we recapitulate some of the conclusions described in the discussion (Section 6) and summarize the possibilities for further research.

- This study shows that distributed tracing can offer an effective way to benchmark distributed systems. However, as shown, the effectiveness of these benchmarks is heavily dependent on the type of application. When the differences between two scenarios become too significant, this can affect the outcome of other services due to the nature of a distributed system. For further research, we recommend reiterating this research on production-grade distributed web systems, to unlock the full potential of our methodologies. Operations that contain a significant amount of disk IO should be carefully considered.

- Unfortunately, the SUT tested in this research incurred some complications in the effectiveness of our measurements. Although our review of the literature revealed what to expect when moving these types of applications to virtualized environments, our own results were less insightful. From existing studies, we conclude that it is advisory for any enterprise moving to virtualized environments to verify their system performance. Specifically, with 'thicker' virtualization layers (e.g. VM + container), both network and disk IO might be heavily affected. For a single instance the performance penalty of networking latency might seem negligible. However, it should be noted that, in a microservices-like application, these effects can stack up and become quite significant.

- This study shows that disk IO severely affected our SUT performance within a VM. Due to the IO operations in one of our services, our application its response time doubled. These differences have been largely attributed to the reading and writing of small files. This corresponds to the literature review, as we know that system calls to the host OS kernel are significantly more expensive in these scenarios.

- As noted above, due to these severe differences in disk IO performance in a single service, other services are affected. Therefore, it seemed some parts of the application were performing worse, while in fact they simply had a higher throughput of operations scheduled. We acknowledge that this heavy IO might not be representative of a production-grade synchronous request flow. We would like to inspire further research, performing the same tests on services with fewer disk IO involved.
- This study proposed a way to analyze these traces, using different approaches, such as percentile visualizations and statistical tests, to determine the goodness of fit. We have shown that the AD-test and the KS-test are good potential candidates for detecting small differences between the trace span durations, while proving our results to be consistent. Although the tests seem a bit too sensitive to minor alterations among runs, the AD-test shows to be slightly more sensitive to small changes in our distributions. The fact that the tests are too sensitive indicates the need for a different interpretation, not using p-values.
- An algorithm was proposed to select the best fit experiment to simplify the analysis. However, we recognize that this approach may neglect important information from other experiments. For future work, we would like to inspire for better ways of including all experiments and dynamically estimating a cut-off (significance) level for the test statistics we have used.
- Before accepting these measurement methods, it is important to gain more insight into the effect of code instrumentation on performance of the application. Even if the average latencies do not increase, it should be verified whether both the averages and variability are not affected.
- Finally, we present results on system-level metrics within the four tested scenarios. Again we can conclude that some metrics were heavily affected by IO operations, and we should take into account that the same effect can occur as for traces. It could be that a server appears to be less busy due to the lower effective throughput of another service that excludes disk IO workloads.
- During the analysis of these metrics, it was concluded that the hypervisor itself occupies almost 2 GBs of memory. Furthermore, the CPU turns out to be have more cycles per minute on a VM than on bare metal / container, for the same workload. However, this is simply due to waiting for the 'longer' traversed path to open files on the host OS disk. This is confirmed by the number of thread interrupts or context switches that we have seen for IO heavy workloads. Finally, we have also seen the number of interrupts increase with each additional virtualization layer added. We assume that this shows the performance penalty introduced by the VM, which is not aware of all the underlying host CPU its properties.

Altogether, we believe this research its contributions could be an inspiration for other research. It would be interesting to apply the used methodologies to a more representative system. By benchmarking distributed web applications in this less conservative way, more properties of the distributed system its nature are taken into account. Additionally, this study can provide enterprises with a guideline of how to measure performance implications in these systems. Traditional ways of benchmarking, like isolated CPU benchmarks or measuring P_{90} response times could be too conservative to mitigate all risks.

Appendix A

Remaining system traces

In this appendix we present the remaining measured trace spans durations, by depicting their percentile distribution plots and percentile tables.

Please add the following required packages to your document preamble: booktabs graphicx

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	20575	21564	22235	22370	28232	32963	-1.31	0.0
Container	19275	19389	19491	21515	23417	33624	4343.54	0.6991
Virtual Machine	19130	19229	19322	19426	23858	31236	6679.47	0.7552
VM + Container	19668	19773	20014	21076	26897	41642	3224.21	0.6289

Table A.1: Overview of percentiles and AD test for the *user service* call (GET) in μs.

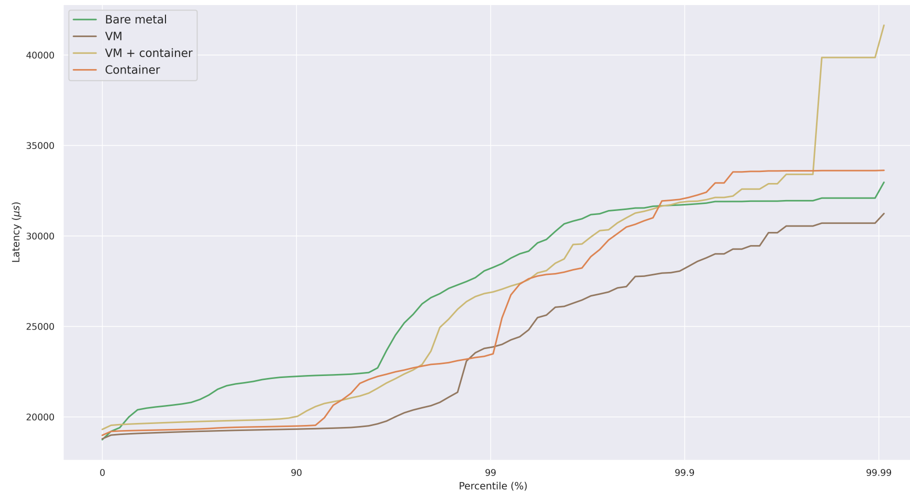


FIGURE A.1: The percentile distribution plot of trace span durations for the *user service* call (GET).

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	20229	21212	21924	22072	27874	32400	-1.31	0.0
Container	18916	18949	19003	21471	23075	33360	4472.32	0.7157
Virtual Machine	18655	18709	18773	18891	23350	30577	8640.47	0.8686
VM + Container	19111	19161	19414	20471	26564	40723	3614.52	0.6398

TABLE A.2: Overview of percentiles and AD test for the *LinpackBenchmark1* task in μs.

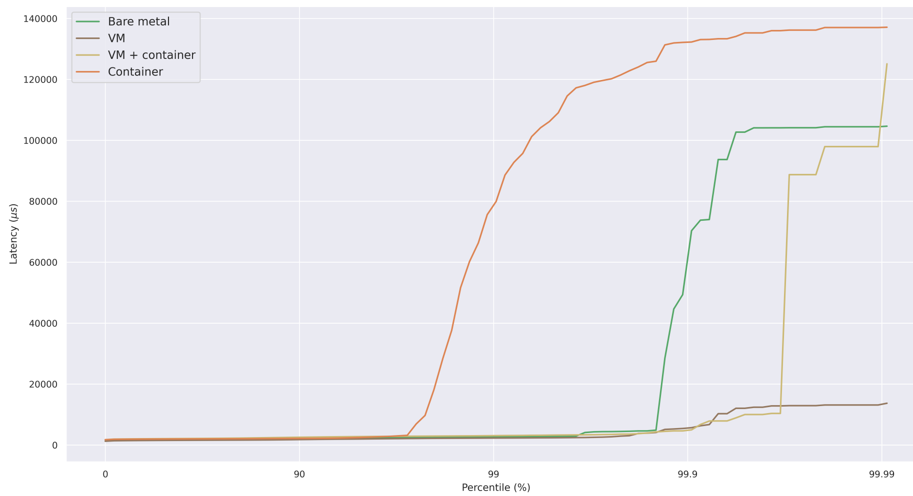


FIGURE A.2: The percentile distribution plot of trace span durations for the *linpack-benchmark1* task (*user service*).

	P_{50}	P_{75}	P_{90}	P_{95}	P_{99}	P_{100}	**AD statistic**	**KS statistic**
Bare metal	6366	7470	8090	8528	9983	19290	-1.31	0.0
Container	5207	5229	5260	6703	8575	15102	3297.35	0.6422
Virtual Machine	4993	5020	5050	5088	6682	18584	4434.88	0.7069
VM + Container	5202	5239	5319	6454	8720	19544	3441.96	0.624

TABLE A.3: Overview of percentiles and AD test for the *LinpackBenchmark2* task in μs.

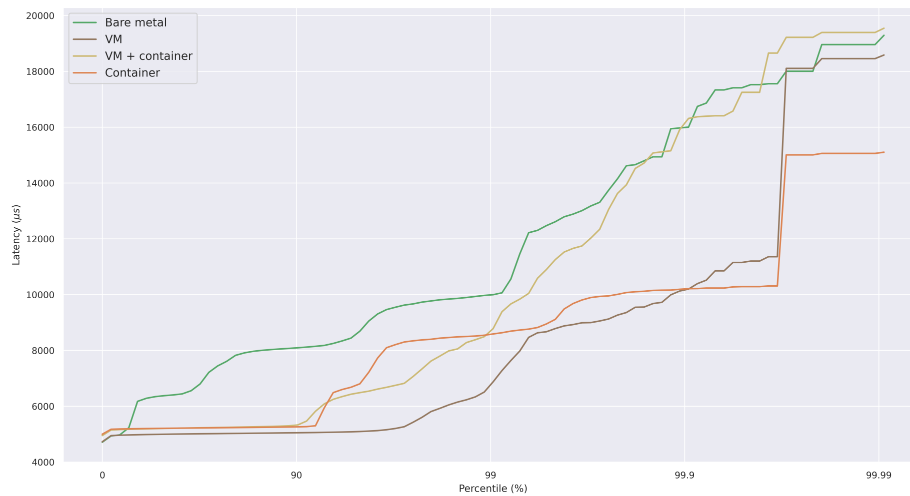


FIGURE A.3: The percentile distribution plot of trace span durations for the *linpack-benchmark2* task (*payment service*).

Bibliography

[1] Sultan Abdullah Algarni et al. "Performance evaluation of Xen, KVM, and proxmox hypervisors". In: *International Journal of Open Source Software and Processes (IJOSSP)* 9.2 (2018), pp. 39–54.

[2] Abhineet Anand, Amit Chaudhary, and Arvindhan Mu. *The Need for Virtualization: When and Why Virtualization Took Over Physical Servers*. Aug. 2020, pp. 1351–1359. ISBN: 978-981-15-5340-0. DOI: 10.1007/978-981-15-5341-7_102.

[3] Theodore W Anderson. "On the distribution of the two-sample Cramer-von Mises criterion". In: *The Annals of Mathematical Statistics* (1962), pp. 1148–1159.

[4] Betsy Beyer et al. *Site Reliability Engineering: How Google Runs Production Systems*. 2016. URL: http://landing.google.com/sre/book.html.

[5] Janki Bhimani et al. "Understanding performance of I/O intensive containerized applications for NVMe SSDs". In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. 2016, pp. 1–8. DOI: 10.1109/PCCC.2016.7820650.

[6] S. M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[7] Jeffrey P Casazza, Michael Greenfield, and Kan Shi. "Redefining server performance characterization for virtualization benchmarking." In: *Intel Technology Journal* 10.3 (2006).

[8] Humble Devassy Chirammal, Prasad Mukhedkar, and Anil Vettahu. *Mastering KVM virtualization: Dive in to the cutting edge techniques of linux KVM virtualization, and build the Virtualization Solutions Your Datacenter demands*. 2016.

[9] G.W. Corder and D.I. Foreman. *Nonparametric Statistics: A Step-by-Step Approach*. Wiley, 2014. ISBN: 9781118840313. URL: https://books.google.nl/books?id=hYVYAwAAQBAJ.

[10] Mariela Curiel and Ana Pont. "Workload Generators for Web-Based Systems: Characteristics, Current Status, and Challenges". In: *IEEE Communications Surveys Tutorials* 20.2 (2018), pp. 1526–1546. DOI: 10.1109/COMST.2018.2798641.

[11] Christoffer Dall and Jason Nieh. "KVM/ARM: The Design and Implementation of the Linux ARM Hypervisor". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 333–348. ISBN: 9781450323055. DOI: 10.1145/2541940.2541946. URL: https://doi.org/10.1145/2541940.2541946.

[12] Sonja Engmann and Denis Cousineau. "Comparing distributions: the two-sample Anderson–Darling test as an alternative to the Kolmogorov–Smirnov test". In: *Journal of Applied Quantitative Methods* 6 (Sept. 2011), pp. 1–17.

[13] Dominik Ernst and Stefan Tai. "Offline trace generation for microservice observability". In: *2021 IEEE 25th International Enterprise Distributed Object Computing Workshop (EDOCW)*. IEEE. 2021, pp. 308–317.

[14] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: vol. 00. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[15] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[16] Nikolas Herbst et al. "Quantifying Cloud Performance and Dependability: Taxonomy, Metric Design, and Emerging Challenges". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3 (Aug. 2018). DOI: 10.1145/3236332.

[17] Ann Mary Joy. "Performance comparison between Linux containers and virtual machines". In: *2015 International Conference on Advances in Computer Engineering and Applications*. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[18] Jonathan Kaldor et al. "Canopy: An End-to-End Performance Tracing And Analysis System". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 34–50. ISBN: 9781450350853. DOI: 10.1145/3132747.3132749. URL: https://doi.org/10.1145/3132747.3132749.

[19] Zhanibek Kozhirbayev and Richard O. Sinnott. "A performance comparison of container-based technologies for the Cloud". In: *Future Generation Computer Systems* 68 (2017), pp. 175–182. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2016.08.025. URL: https://www.sciencedirect.com/science/article/pii/S0167739X16303041.

[20] Jack Li et al. "Performance Overhead among Three Hypervisors: An Experimental Study Using Hadoop Benchmarks". In: *2013 IEEE International Congress on Big Data*. 2013, pp. 9–16. DOI: 10.1109/BigData.Congress.2013.11.

[21] Zheng Li et al. "Performance Overhead Comparison between Hypervisor and Container Based Virtualization". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. 2017, pp. 955–962. DOI: 10.1109/AINA.2017.79.

[22] Jiuxing Liu et al. "High Performance VMM-Bypass I/O in Virtual Machines." In: *USENIX Annual Technical Conference, General Track*. 2006, pp. 29–42.

[23] Tao Lu. "Optimizing Virtual Machine I/O Performance in Cloud Environments". PhD thesis. Nov. 2016.

[24] Frank J Massey Jr. "The Kolmogorov-Smirnov test for goodness of fit". In: *Journal of the American statistical Association* 46.253 (1951), pp. 68–78.

[25] Soheil Mazaheri et al. "Cloud benchmarking in bare-metal, virtualized, and containerized execution environments". In: *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*. 2016, pp. 371–376. DOI: 10.1109/CCIS.2016.7790286.

[26] Scott McCarty. *Architecting Containers Part 2: Why the user space matters*. Sept. 2015. URL: https://www.redhat.com/en/blog/architecting-containers-part-2-why-user-space-matters.

[27] Richard McDougall and Jennifer Anderson. "Virtualization Performance: Perspectives and Challenges Ahead". In: *SIGOPS Oper. Syst. Rev.* 44.4 (Dec. 2010), pp. 40–56. ISSN: 0163-5980. DOI: 10.1145/1899928.1899933. URL: https://doi.org/10.1145/1899928.1899933.

[28] Nornadiah Mohd Razali and Bee Yap. "Power Comparisons of Shapiro-Wilk, Kolmogorov-Smirnov, Lilliefors and Anderson-Darling Tests". In: *J. Stat. Model. Analytics* 2 (Jan. 2011).

[29] Susan Moore. *Gartner forecasts strong revenue growth for global container management software and services through 2024*. June 2020. URL: https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co.

[30] Susan Moore. *Gartner says more than half of enterprise IT spending in key market segments will shift to the cloud by 2025*. Feb. 2022. URL: https://www.gartner.com/en/newsroom/press-releases/2022-02-09-gartner-says-more-than-half-of-enterprise-it-spending.

[31] Karthik Nagaraj, Charles Killian, and Jennifer Neville. "Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 26.

[32] Kay Ousterhout et al. "Making Sense of Performance in Data Analytics Frameworks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 293–307. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[33] A. Parker et al. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, Incorporated, 2020. ISBN: 9781492056638. URL: https://books.google.nl/books?id=fgfIyAEACAAJ.

[34] Maxim Polenov, Vyacheslav Guzik, and Vladislav Lukyanov. "Hypervisors Comparison and Their Performance Testing". In: *Software Engineering and Algorithms in Intelligent Systems*. Ed. by Radek Silhavy. Cham: Springer International Publishing, 2019, pp. 148–157. ISBN: 978-3-319-91186-1.

[35] Matthew Portnoy. *Virtualization Essentials*. 1st. USA: SYBEX Inc., 2012. ISBN: 1118176715.

[36] Raja Sambasivan et al. "Diagnosing performance changes by comparing request flows". In: Oct. 2011.

[37] Raja R Sambasivan et al. "Principled workflow-centric tracing of distributed systems". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 401–414.

[38] Raja R Sambasivan et al. "Principled workflow-centric tracing of distributed systems". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 2016, pp. 401–414.

[39] Fritz W Scholz and Michael A Stephens. "K-sample Anderson–Darling tests". In: *Journal of the American Statistical Association* 82.399 (1987), pp. 918–924.

[40] RJ Serfling. "The Wilcoxon two-sample statistic on strongly mixing processes". In: *The Annals of Mathematical Statistics* (1968), pp. 1202–1209.

[41] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[42] Malcolm J Slakter. "A comparison of the Pearson chi-square and Kolmogorov goodness-of-fit tests with respect to validity". In: *Journal of the American Statistical Association* 60.311 (1965), pp. 854–858.

[43] Zeyi Tao et al. "A Survey of Virtual Machine Management in Edge Computing". In: *Proceedings of the IEEE* 107.8 (2019), pp. 1482–1499. DOI: 10.1109/JPROC.2019.2927919.

[44] Roberto Vitillo. *Why you should measure tail latencies*. May 2020. URL: https://robertovitillo.com/why-you-should-measure-tail-latencies/.