Universiteit
Leiden

Master Computer Science

Towards a Testbed for Evaluating Microservice Architecture Performance

Name:	Stephan van der Putten
Student ID:	s1528459
Date:	30/08/2024
Specialisation:	Data Science
1st supervisor:	Kristian Rietveld
2nd supervisor:	Alexandru Uta

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

The microservice architecture has gained increasing attention in both industry and academia as a design pattern for complex, distributed applications. However, there has been a lack of reproducible and representative scientific research on microservice architecture performance due to the lack of benchmarks and tools. To address this gap, we developed a minimal open-source testbed that includes a proof-of-concept setup for deployment, environment setup, workloads, and experiment scripts for use in CloudLab. Using this testbed, we conducted a comprehensive evaluation of three popular container orchestration engines that are typically used to deploy microservice applications. Kubernetes, Nomad, and Docker Swarm. The knowledge we gained during the implementation to create a consistent environment for the testbed will help other researchers when conducting similar research. Our experiments used benchmarks from the DeathStarBench benchmark suite to investigate the performance of these engines under scaling scenarios, with a focus on comparing the performance differences between the orchestrators. Our findings revealed significant performance variations, for which we provide insights and recommendations based on our observations. Our work contributes to a better understanding of the performance characteristics of container orchestration engines in the context of microservice architecture and enables researchers and organisations to make informed decisions about tool selection and deployment.

Acknowledgements

I would like to express my sincere gratitude to my thesis supervisors, Alexandru Uta and Kristian Rietveld, for their guidance and support throughout the duration of this research project. I am grateful for the time, constructive feedback, and effort they invested in helping me develop and refine my thesis. I would also like to thank Alexandru Uta for his patience, support and encouragement, as well as my friends, family, and study advisor Alexandra for their motivation and emotional support. In addition, I would like to express my appreciation to the reviewers who provided valuable feedback on my work. All of these contributions were invaluable in the completion of this thesis.

Contents

Chapter 1

Introduction

The microservice architecture (MSA) is a term commonly associated with Service-Oriented Architecture, Virtualization, Cloud, DevOps, Serverless and many current-day industry technologies, as described by Hamzehloui et al. and Eyk et al. [34, 47]. Originally, microservices were an answer to monolithic applications that suffer from multiple issues, such as maintainability, multiple dependencies, and scalability [16]. This led to the adaptation of microservices acknowledged by numerous companies, such as Netflix [38] and Amazon [29]. Microservices enable the industry to revolutionise its services in a more automated way, such as DevOps [3]. Meanwhile, interest continues to increase as the cloud microservice market is expected to grow to a market of 3 billion in 2026 [39]. Although interest in the industry has been high, academic analyses on microservice performance have been lagging behind. Current research from the industry only primarily provides principles and guidelines or experience reports, without an in-depth academic point of view.

Hamzehloui et al. [34] identified a lack of the tools and experience required as one of the main reasons why microservices are not yet fully understood. Franesco et al. [21] found that only one of the 71 published studies checked until 2017 provided an open-source test system to benchmark microservice-based systems. In recent years, there has been a noteworthy but limited effort to increase the availability of benchmarks. This is important as the lack of open-source microservice benchmarks is a significant limitation for future MSA research, as it makes it difficult to compare results between studies and replicate findings. Furthermore, most microservice systems are proprietary or not easily accessible to the research community [2]. There are currently a limited number of open-source microservice reference applications available, but these applications often lack the necessary depth and complexity for realistic comparison due to their limited number of microservices. An overview of microservice applications that have been created for benchmarking in cloud and/or microservices: TeaStore [48], TrainTicket [51], SocketShop [42], MusicStore [35], Spring Cloud Demo [6], ACME Air [1], μSuite [44], Sirius [23], TailBench [28], CloudSuite [19], DeathStarBench [22], μBench [14]. With the inclusion of DeathStarBench, we now have a benchmarking suite that consists of multiple reference applications with a sufficient number of microservices to accurately represent realistic complex microservice systems. The open-source benchmark suite has released three of a set of five end-to-end microservice-based applications suitable for cloud system experimentation [13]. This benchmark suite enables researchers to focus on a more diverse and open landscape of microservices, as it provides a higher level of depth and complexity compared to other available benchmarks.

In the landscape of microservice architecture and cloud providers, there are numerous open-source frameworks and tools, each with its own implementation and performance trade-offs. Each tool influences the microservice architecture, as we have different categories such as container engine (Docker, LXC), container orchestration engines (Kubernetes, Docker Swarm, Nomad), application architecture (DeathStarBench), public cloud platform (Google Cloud, Azure, AWS), or other infrastructure platforms to consider [26].

1.1 Research goals and questions

To date, there has been a lack of reproducible academic research focused on comparative systems studies on microservice architecture (MSA) with representative benchmarks for the complexity of microservice applications.

This sets our overlaying research goal to enable reproducible and relevant research for the MSA field through 1) the design and implementation of an (experimental/prototype) testbed that utilizes the state-of-the-art benchmark suite DeathStarBench; 2) contribute with experiments on the performance of different container orchestration engines, workloads, and variables in the context of the microservice architecture.

We have identified the following research questions to guide our thesis.

> *RQ T.1: How to design and implement the DeathStarBench into a testbed?*
>
> *RQ T.2: How do we implement consistency/reproducibility of our testbed? What is required to set up our testbed/experiments?*
>
> *RQ T.3: Using our testbed, how can we ensure consistency in our experiments? What pre-tuning is required to conduct experiments, and what performance characteristics can we find?*
>
> *RQ T.4: Using our testbed, are there performance differences in the orchestration tools in various scaling scenarios?*

1.2 Our Contributions

With this thesis we contribute the following:

- **Design & Implementation of a minimal open-source testbed for CloudLab to experiment**: Following the implementation of our testbed, we had to create a configuration of our experimental setup. This includes how we configured the nodes in Cloudlab (or any cluster of accessible nodes with root access) on which we run our experiments and also how we configured the experimental setup to perform the experiments. This will help future researchers decide how to set up our testbed for their research. Please refer to Chapter 4 along with the deployment files found on GitHub [1].
- **Comparative study on Containers Orchestration Tools**: The first part of our experiments involves an initial setup with Docker Swarm to establish the load generator settings and set a baseline for subsequent experimental

[1] https://github.com/Stvdputten/Orchestration

comparisons. This part can help future researchers save time by deploying our tools for similar research. We have implemented DeathStarBench on top of this testbed, please refer to Chapter 5, Chapter 6.

- **Lessons learned during our configuration**: To create our proposed testbed, we were faced with multiple challenges in implementing the applications for different orchestration tools, and as such reflect that work in this study in the discussion. Please refer to Chapter 7.
- **Dataset of our Experiments**: In our study, we performed extensive research on our testbed. Our results include a dataset which has all our configuration settings of the load generator, experiment setup parameters including horizontal-, vertical-scaling and orchestration tool, and results measured in tail latency and throughput available for any researcher to compare it against. Please refer to our GitHub[2].

1.3 Thesis Organisation

The chapters in this thesis are arranged as follows. The second and third chapters introduce the background of our testbed and the related work. We continue the fourth chapter, Design & Implementation, by discussing the system design choices to modify existing benchmarks to run with the infrastructure provided in Cloudlab. In this fourth chapter, we also talk about how to set up our proof-of-concept testbed and the included scripts with examples to extend the testbed. Our testbed is then used to perform the experiments on the MSA. We focus on exploring the performance of the microservice architecture through scaling parameters (horizontal, vertical, high availability on/off), different orchestration tools, and exploring the performance limits in the setup. In our preliminary experiments, we only use Docker Swarm as a baseline to get an initial configuration setup using some tests presented in Appendix A and Chapter five. The second part of the experiments, presented in Chapter six, continues with an extensive series of initial experiments to create a comparison of the performance of the different orchestrators, to explore the scaling parameters and to stress the benchmarks to their limits. The results are then discussed and evaluated. In Chapter seven we discuss the pros and cons of our testbed, we propose the value of this research for others, how our research can be extended, and the limits of our approach to extend this testbed to other benchmarks as our lessons learnt. Chapter eight is the conclusion of our study and proposes future work.

[2] https://github.com/Stvdputten/hdr-plot & https://github.com/Stvdputten/Orchestration

Chapter 2

Background

In this chapter, we provide the background of the technology that enabled the adoption of the microservice architecture. This is followed by a deeper look at the current tools used in our study. We first look at the origin of the microservice architecture compared to the monolithic architecture. We continue with a look at containers and their comparison with virtual machines. What are container orchestration tools, and which are we using in this study? Which physical testbed (Cloudlab) are we building on for experimentation and research? Finally, what applications are in the benchmark suite (DeathStarBench) and what is included in it?

2.1 From Monolith to Microservices

Monolithic architecture design was primarily used in the earlier days of application design. It was characterised by a single unified software application with tight coupling between components, which can become unmanageable and inflexible as the complexity of a system grows. Usually, the developed application(s) run on a single large virtual machine.

As business needs grow due to an increase in demand, the complexity of applications increases with time, for which monolithic application design becomes increasingly a burden to scale and maintain. With the introduction of containers and lightweight environments to run applications, the concept of microservice-based applications was realised. Compared to monoliths, microservices-based architectures offer greater flexibility and scalability through the use of loosely coupled and independently deployable components.

Microservice architecture (MSA), has no one singular definitions, while some see it as a set of characteristics defined by Lewis and Fowler [31], other see it as the realisation of another concept called service-oriented architecture. Service-oriented architecture is an earlier architectural pattern that splits a software in a set of service components that are similar to their business logic. Compared to MSAs, service-oriented architecture is more coarse-grained, the philosophy about sharing data across services and also a focus on service orchestration to name a few. MSAs are typically implemented as lightweight applications written in various modern programming languages, each with its own specific dependencies, libraries, and environmental requirements. To ensure that a microservice has everything it needs to run successfully, it is packaged together with its dependencies in a container, which can be easily deployed and managed in a production environment. In the last decade, the microservice architecture for application design has increased in popularity through wide adoption in the industry through public cloud platforms such as Google, AWS, and Azure.

2.1.1 Containers and VMs

Containers are broadly used as the backbone of many current day software applications. Containers are a powerful tool for delivering high-performing, scalable applications on any infrastructure, as they provide a means of encapsulating microservices and their dependencies in isolated, portable virtual environments. Containers do not run microservices directly but rather operate on container images, which are executable environments that bundle the application along with its runtime, libraries, and dependencies. These images serve as the source for containers deployed on various platforms, including workstations, virtual machines, and public clouds. The portability and flexibility of containerization make containerization an attractive approach for the deployment of microservices in diverse environments. This enables the deployment of microservices in a consistent and predictable manner, without interference from other running applications in a single (virtual) environment.

Virtual machines (VMs) are a software-based abstraction of a physical computer that, with the need of some built-in hardware support. VMs create a standalone, isolated environment in which an operating system and its applications can run. Virtual machines are useful for a variety of purposes, such as testing and development, providing a consistent runtime environment for applications, and enabling the deployment of legacy software on modern hardware. However, they also have some limitations and overhead compared to other approaches, such as containers.

Containers, compared to VMs, are a lightweight and more efficient form of environment virtualization that allows applications to be packaged together with their dependencies and run in isolated environments. Unlike virtual machines, which create virtual copies of the entire hardware and operating system, containers share the same kernel of the operating system as the host and only include the libraries and dependencies needed for the specific application. Containers, similar to VMs, are portable and resource-efficient, as they can easily moved between different hosts and environments without the need for re-installation or configuration. Containers are more efficient in this regard, as they do not have to completely duplicate the entire OS kernel, offering lower costs and faster deployment time.

In conclusion, virtual machines provide a complete standalone environment for running multiple operating systems and applications, while containers offer a more efficient and portable way of packaging and deploying applications within a single operating system. Both approaches have their benefits and trade-offs, and the choice between them depends on the specific requirements and needs of the application.

2.1.2 Container Orchestration

Container orchestration is the process of automating the deployment, scaling, and management of containers in a distributed environment. Container orchestration involves coordinating the life cycle of containers across a cluster of (virtual) nodes, including tasks such as deploying and scaling containers, distributing workloads across the cluster, and ensuring the availability and resiliency of the system.

Microservices are typically implemented using containerization. However, full utilization of containers requires additional tools, such as container orchestrators. Container orchestration is an important aspect of modern distributed systems, as it helps developers and operations teams to manage and deploy complex applications built using microservices. By automating the management of containers, orchestration tools enable developers to focus on building and developing their applications, rather than worrying about the underlying infrastructure.

2.2 Container orchestration tools

At the time of writing, various container orchestration tools are available. Within the scope of this thesis, we use *Nomad*, *Docker Swarm*, and *Kubernetes*, as they are the most commonly used and widely supported by an open-source community. We will now briefly discuss the orchestration tools and their configuration for those unfamiliar with them.

2.2.1 Nomad & Consul

Nomad [36] is a container and workload orchestrator developed by HashiCorp, which utilizes the HashiCorp Configuration Language (HCL) to deploy applications. As it does not natively support automatic service discovery, Nomad is often paired with **Consul** [12], a service mesh and service discovery tool that manages the sharing of data and service discovery among microservices.

Both Nomad and Consul operate as services on pre-allocated nodes, as illustrated in Figure 2.1. Nodes can be configured as either servers (managers) or clients (workers) nodes. Nomad and Consul both use the Raft consensus algorithm and Serf gossip protocol for communication. Servers employ the Raft consensus algorithm to ensure log replication and consistency between servers, as shown in Figure 2.1. The gossip protocol is used as shared communication by all nodes (server and client) to perform actions such as job placement on client nodes and communicate the state of nodes in the cluster. The RPC protocol is used for all direct and targeted communication between the server and the client nodes. To ensure high availability, it is recommended that you have at least three or five manager nodes. In Nomad, applications are executed using jobs, which are specifications consisting of task groups. Task groups are workloads that must be run on the same node, while tasks are the smallest unit of a job and specify the work to be performed (e.g., a Nginx service).

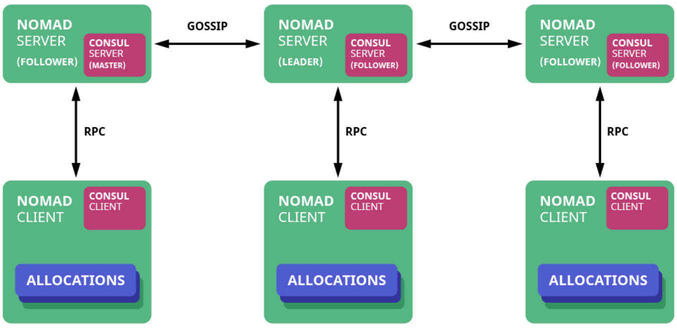


Figure 2.1: Overview of the Nomad & Consul setup. Managers (servers) communicate and allocate workloads to the Nomad workers (clients). Consul takes care of the communication between services [18].

2.2.2 Docker Swarm

Docker Swarm [15] is a container orchestrator provided by Docker as part of the Docker Engine, a standard tool for deploying containers. To set up Docker Swarm, multiple nodes need to run the Docker Engine, for which each node needs to be configured as either a worker or manager. Managers, which use the Raft consensus protocol to select a leader, perform orchestration and cluster management to maintain the desired state of the applications. The worker nodes execute tasks and report any changes to the managers. To ensure high availability, it is recommended to have at least three manager nodes, as can be seen in Figure 2.2, managers delegate tasks (e.g., a Nginx service) to worker nodes.

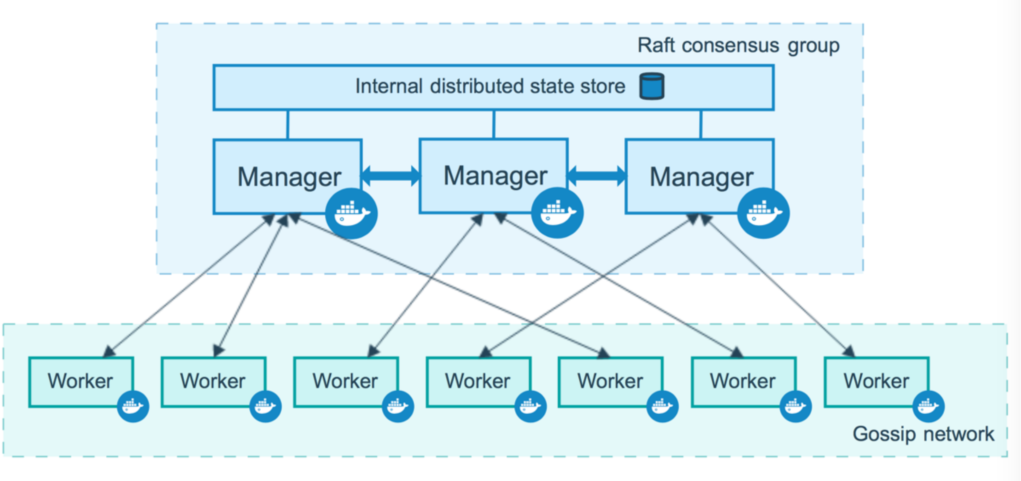


Figure 2.2: Overview of the Docker Swarm manager-worker relationship [25].

2.2.3 Kubernetes

Kubernetes [30] is an open-source container orchestration platform that automates the management, deployment, and scaling of containerised applications. It was originally developed by Google and is now maintained by the Cloud Native Computing Foundation (CNCF).

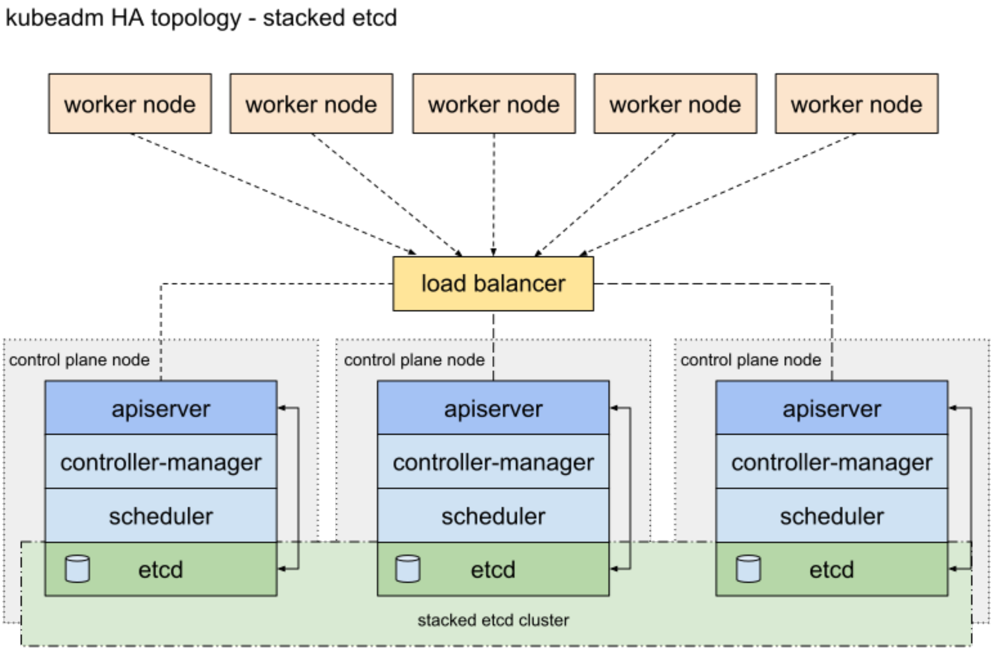


Figure 2.3: Overview of the Kubernetes manager-worker relationship [37].

Kubernetes operates a control plane consisting of one or multiple control plane nodes (also known as managers), which supervise the worker nodes that run the workloads assigned by the control plane, as illustrated in Figure 2.3. The control plane stores critical data in etcd, a key-value storage system based on the Raft consensus protocol. Furthermore, both manager and worker nodes use a container engine (such as containerd) to create containers for executing tasks. Kubernetes also supports the use of multiple container networks with their Container Network Interfaces to create overlay networks for communication between services. Kubernetes deployment can be complex, and tools such as Kubeadm have been developed to facilitate the creation of a cluster. Kubernetes is widely used in the industry and has become a de facto standard for container orchestration.

2.3 Testbeds and benchmark suite

Testbeds will be frequently discussed throughout this thesis, and therefore we dedicate a few paragraphs describing the definition of the testbed and its importance.

A **testbed** is conceptually speaking an experimental setup with included tools that are used to rigorously test and monitor the performance of a (complex) system. To study the behaviour and characteristics of a system, researchers use a testbed to design and conduct experiments under controlled conditions.

Testbeds are important for academic research because they provide a controlled and repeatable environment for studying the performance and behaviour of complex systems. By using a testbed, researchers can systematically vary different parameters (e.g., workload, network conditions, hardware configurations) and measure the effects on the system's performance using desired metrics such as throughput and tail latency. This allows researchers to gain a better understanding of the underlying mechanisms and patterns of complex systems, and to identify potential issues and challenges that may arise in real-world deployments. By conducting experiments on a testbed, researchers can validate their hypotheses and theories and provide empirical evidence to support their findings and conclusions.

2.3.1 CloudLab: flexible, scientific infrastructure for research

Cloudlab [17] is one of the core components of our research. CloudLab is a platform for conducting experiments and deploying testbeds in a cloud-like environment. It provides a standardized, reproducible, and scalable infrastructure for researchers and developers to study and evaluate the performance and behaviour of complex distributed systems, such as cloud computing platforms, networked systems, and distributed storage systems.

CloudLab consists of a network of physical and virtual machines, connected by high-speed networks, that can be customized to create a wide range of experimental environments. Users can configure the hardware, software, and networking aspects of their experiments and can use CloudLab to deploy and manage a variety of applications and workloads.

CloudLab is designed to be user-friendly and easy to use, with a web-based interface and a range of tools and libraries for building and deploying testbeds. It has been widely adopted by researchers in academia and industry and has been applied in a range of fields, including computer science, engineering, and biology.

2.3.2 DeathStarBench: a benchmark suite

The **DeathStarBench** [13] (DSB) is an open-source benchmark suite that currently consists of three applications, built out of microservices, that can be used to benchmark a system under test. Originally, it was used to analyse the latency and throughput between client and server systems, producing the percentile latencies and data transfer per second for all the HTTP requests. To allow comparisons between systems, each system is benchmarked using the workloads of a load generator in an isolated network environment. The three applications are designed to function as a social network, a media review system, and a hotel reservation plot system on Google Maps which can be deployed on the target system. Each application contains a dataset to load and has one or more predefined workload generators. All applications make use of Jaeger to monitor and trace communication between microservices.

Benchmark 1: The Social Network

The **Social Network** (SN): The application includes 28 microservices. The application is a social media site in which users can follow each other, create posts, search engines, and see each other's profiles. The front-end uses Nginx to communicate with multiple microservices in the back end which store their data in several database and caching microservices (Redis, MongoDB, and Memcached). Furthermore, it uses Thrift-like microservice components to implement the logic. The dataset can be loaded with a Python script with around 962 users. Users follow each other according to a graph from the Facebook Reed98 dataset [40], but other social network graphs can easily be applied. The included workload generator has three workloads for the supported HTTP requests: reading the user timeline, home timeline, and composing user posts. The workload can also send mixed requests.

Benchmark 2: The Media Review System

The **Media Microservice** (MM) allows users to browse film information and review movies. Similarly to the social network, the application can be accessed through a Nginx web service to back-end microservices, which also use several database microservices, and includes a container-level network DNS resolver as a microservice. The dataset uses a Python script that loads the film data and registers the users and movies in the application. It has only one workload, which composes film reviews. The application includes around 31 microservices.

Benchmark 3: The Hotel Reservation System

The **Hotel Reservation** (HR) application is based on a Go microservice example of a Hotel Reservation application [49]. The application allows the user to make hotel reservations, get profiles and ratings of nearby hotels during a given time period, and recommend hotels based on user input. The application includes a front-end that is accessible through HTTP using the gRPC for inter-service communication. The application also uses a container-level DNS resolver microservice called *Consul* (similar software use case to our previous mention of Consul and Nomad, which does node orchestration instead of inter-service communication). The dataset is included in the application and does not require any further steps to configure. It includes a workload that sends a mixed variety of requests to reserve, recommend hotels, and plot hotels on Google Maps. The application includes 19 microservices.

Chapter 3

Related Work

This chapter addresses related work on microservice architecture. What challenges do we face when benchmarking microservice applications? In addition, we discuss performance-related studies to evaluate microservices and discuss the state of reproducibility.

3.1 Benchmarking for microservice applications

Microservices have been around for a while and emerged in 2014 [31]. After those initial years, Hamzehloui et al. [34] identified the lack of research in that area due to the tools and experience as one of the main reasons why microservices were not that well understood performance wise. The lack of open-source microservice benchmarks is a significant limitation for microservice architecture research, as it makes it difficult to compare the results between studies and replicate the findings. Furthermore, most microservice systems are proprietary or not easily accessible to the research community [2]. As this is still an open challenge, to enable open-source and reproducible research in this area, we will focus our study on creating a tool that can support this.

In 2017, Aderlaod et al. evaluated several benchmarks for microservice systems, most of Acme Air, Spring Cloud Demo Apps, Socks Shop, and MusicStore [1, 43, 42, 35]. An overview of relevant microservice applications that have been created for benchmarking in cloud and/or microservices include TeaStore [48], TrainTicket [51], SocketShop [42], MusicStore [35], Spring Cloud Demo [6], ACME Air [1], μSuite [44], Sirius [23], TailBench [28], CloudSuite [19], μBench [14], DeathStarBench [22]. With the inclusion of DeathStarBench, we now have a benchmarking suite that consists of multiple reference applications with a sufficient number of microservices to accurately represent a realistic microservice architecture application. The open-source benchmark suite has released three of a set of five end-to-end microservice-based applications suitable for cloud system experimentation [13]. DSB is promising for enabling researchers to study microservice performance, as it provides a higher level of depth and complexity compared to other available benchmarks. It is still an experimental suite of applications and tools that will require some tweaking to meet our needs. Still, we will include DeathStarBench in the experimental testbed to find if we can enable a comparative study tool for the microservice architecture.

3.2 Performance and reproducibility

Microservices present a unique challenge for analysis due to their decentralised and independent nature. In 2022, Bushong et al. [8] provided a systematic mapping study where understanding performance issues was still perceived as an open problem. In a study in 2017, Heinrich et al. [24], performed a study that showed that microservice performance testing can be challenging for testing, monitoring, and modelling performance. The challenge arises due to the scale of microservices and the autoscaling policies employed by orchestrators like Kubernetes, necessitating simpler models to comprehend performance issues. Our experiments will therefore start with a fixed set of microservices and be limited to explicit scaling of our resources in containers, to keep it simple and see what we can learn from that approach. The DeathStarBench in our study will be the core benchmark suite of our tool to enable research in this field on microservice performance; not much research has been done on performance in realistic environments. Furthermore, performance variability is an acknowledged problem in the cloud and performance engineering field, as stated by Uta et al. [46]. The study goes into more depth about reasons for a lack of performance reproducibility, such as a lack of sound experimentation due to a lack of using enough statistics and at the same time the variability introduced by cloud platforms. Our thesis will include a more stable environment, comparing systems using CloudLab [10] which is open to the research community, making it possible to use similar hardware to perform our experiments. Furthermore, we will look at a broad range of experiments to move toward reproducibility. Efforts are being made to improve reproducibility within the research community. For example, a detailed study on the Alibaba cloud [32] has been published, which includes a dataset. Nonetheless, reproducibility remains difficult due to insufficient transparency in the tools used. Therefore, we will ensure that both our tools and dataset are accessible online.

Existing research directions in the microservice architecture performance domain are dispersed, focused on microservices versus monolithic performance characteristics [7], examining the scaling policy performance [9, 50] or evaluating varying (auto)scaling [5, 4] policies and performances for Kubernetes. Kubernetes stands as a leading orchestrator in microservice performance research, whereas numerous other orchestrators have appeared and disappeared throughout the years. Thus, we believe that it is crucial to conduct comparative research on the performance of various container orchestrators, including Kubernetes. However, we can only find research of multiple orchestrators on functional characteristics [27, 33], which focused on characteristics such as provisioning time, security, learning curve, and more. Therefore, we will incorporate additional orchestrators into our study. We include Kubernetes, Docker Swarm and additionally add Nomad as understudied orchestrator to compare and see if that also leads to similar reproducibility between orchestrators. In conclusion, while the recent addition of benchmarks and experiments of representative real-world microservice applications is a step in the right direction, we still find the lack of comparative studies for the characteristics of the microservice architecture and orchestration platforms to be an existing gap.

Chapter 4

Design & Implementation

This chapter presents the design of the testbed and its components. Our goal is to create a reproducible and representative tool for use in a controlled environment to experiment and benchmark microservice applications. During the creation of our tool (testbed deployment files, automation, implementation of DSB, etc.), we designed and implemented the testbed in parallel to the DeathStarBench (DSB) implementation on the testbed on which to conduct experiments. We will begin with a visual overview of the testbed design, consisting of multiple layers that we logically divided when creating this design for our testbed tool. Each component consists of a step-by-step guide to recreate our testbed. These steps include the setup in CloudLab and (re)implementation of the applications from DeathStarBench to ensure compatibility with the testbed for the container orchestrators: Kubernetes, Docker Swarm, and Nomad.

One of the recent additions to the research community is the DeathStarBench, a microservice benchmark suite. This part represents, in terms of time spent, the majority of this thesis. To reflect that work, we have created a separate chapter here to help guide future research to be able to reproduce our research. At the start of this thesis, we first explored DSB to explore the applications inside. From there we continued designing and building a testbed and ported the remaining DSB applications to our current product.

To motivate our approach and to perform reproducible research, the option of a testbed checks all the marks. During our design, we have looked at the landscape of microservice architecture and the tools available to deploy these applications. As a popular choice for cloud-native/microservice design, there are numerous open-source frameworks and tools, each with its own implementation and performance trade-offs. Each tool influences the microservice architecture, as we have different technologies such as the container engine (Docker, LXC, containerd), container orchestration engines (Kubernetes, Docker Swarm, Nomad), application architecture/benchmark suites (DeathStarBench, TeaStore) and/or public/private/federated cloud platform (Google Cloud, Azure, AWS, CloudLab) or other infrastructure platforms to consider [26]. In this thesis, we focus on the most popular container orchestrators, Kubernetes, Nomad and Docker Swarm. Our final design is shown in multiple layers; see Figure 4.1. The next section will discuss each layer shown in the figure.

4.1 Design of the Testbed

In this section, we are answering our research question:

> *RQ T.1: How to design and implement the DeathStarBench into a testbed?*

As we mentioned in Section 2.3, a testbed is conceptually speaking an experimental setup with included tools that are used to rigorously test and monitor the performance of a (complex) system. To study the behaviour and characteristics of a system, researchers use a testbed to design and conduct experiments under controlled conditions.

Before we start discussing the details to reproduce the implementation design choices we took in our thesis, we will take a top-down approach to discuss the components of our testbed and the final contribution. Starting with a complete overview and then discussing each layer separately.

Figure 4.1 is an overview of the testbed we designed and the underlying tools used during our thesis. The complete testbed consists of five layers, each layer is implemented primarily with a set of scripts in Bash to automate the process of deploying and running the experiments where possible. As mentioned before, our design considerations are both *reproducibility* and *representativeness* to support the research community. Our selection of the number of layers is based on the distinct phases of investigation required during the development of this testbed and its parts. Let us go through the process.

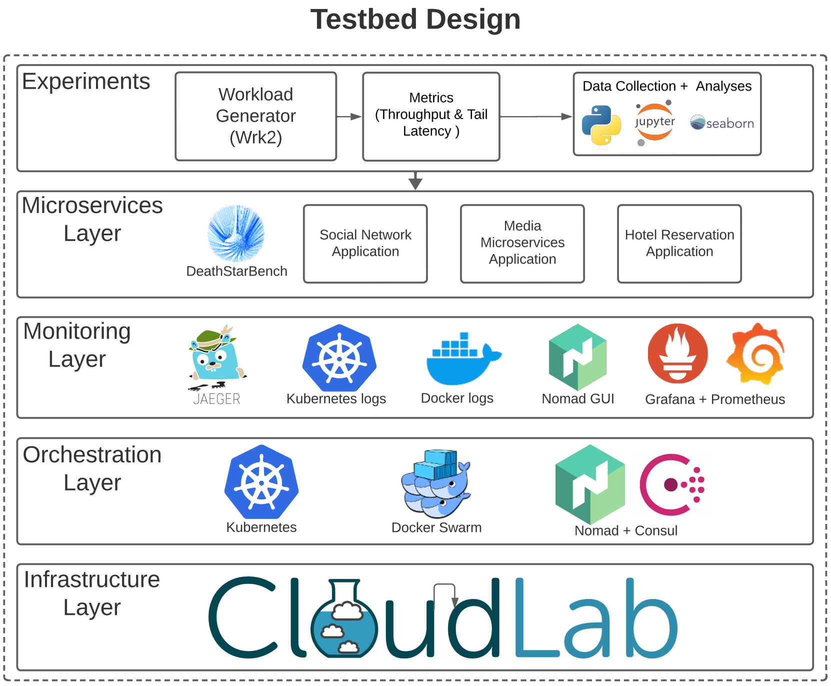


Figure 4.1: An overview of the testbed and each layer that is deployed to set up the experiments. The experiments are run benchmarking the applications in the microservice layer. The testbed is run starting from the bottom layer and ending with the top layer.

The *infrastructure layer* deploys the hardware and/or virtual machines on which we can run our orchestrators and microservices. It runs on the CloudLab platform[1], which is available to the research community with hardware that

[1] https://cloudlab.us/

supports reproducibility. It is of great value that CloudLab is available to us. Alternatively, we could have considered the public cloud, but it is less affordable, adds an additional layer of complexity to our research due to development time, and is also constantly changing, as mentioned by Uta et al. [46] in the case of single cloud usage due to underlying (commercial) changes, which causes a reproducibility problem. In comparison, CloudLab is a more controlled environment; as such, we will use CloudLab throughout this thesis. Furthermore, this step also includes the choice of OS to deploy and the networking between each node in the infrastructure.

The *orchestration layer* deploys the container orchestrator platform on the nodes, which manages the deployment, scaling, and networking of microservices. After our infrastructure is deployed, we ensure that each node is configured to communicate with each other and includes the necessary environment including packages. Depending on the number of nodes available and the type of experiment, the script will choose the number of managers and use the remaining as workers. For example, with 9 available nodes, the straightforward choice would be three nodes as managers for high availability and the remaining five nodes as workers. The last node is set as the client from which to run the workloads on the remaining cluster.

The *monitoring layer* deploys all the tools to monitor performance and collect logs from the microservice layer and the orchestration layer. Monitoring tools include Grafana and Prometheus for node and container monitoring, Jaeger for tracing and overseeing container workloads, and the built-in orchestrator monitoring tools. These monitoring tools have been helpful and necessary during the debugging of applications to support each orchestration engine.

The *microservice layer* deploys the applications, in our case the DeathStarBench, and ensures that each application can be correctly deployed. The DeathStarBench has three applications available at the time of writing. Each of the three applications consists of multiple microservices. Each application, although available, still required extensive effort to rewrite to run and be ported in our testbed environment and to fit each of the orchestrators underlying design choices, such as DNS resolving, networking design choice, etc.

The last top layer in Figure 4.1 consists of the *experiment layer* that includes workload generation, data collection, and scripts (Python) with Jupyter Notebook to aggregate and visualise the results. The experiments run the workloads to stress the applications, which we visualise with an arrow on the microservice layer. After DSB has been rewritten and deployed on the testbed, we run the experiments. The workloads can be configured, such as the number of requests. The workloads in DSB run using a singular service call, such as the composing of social media posts using the social network application. We ran those workloads on the applications, which returned tail latency and throughput. Using our testbed implementation, we input the results to evaluate the final performance with figures.

The remaining sections will discuss in detail the implementation of each layer in more detail.

4.2 Implementation

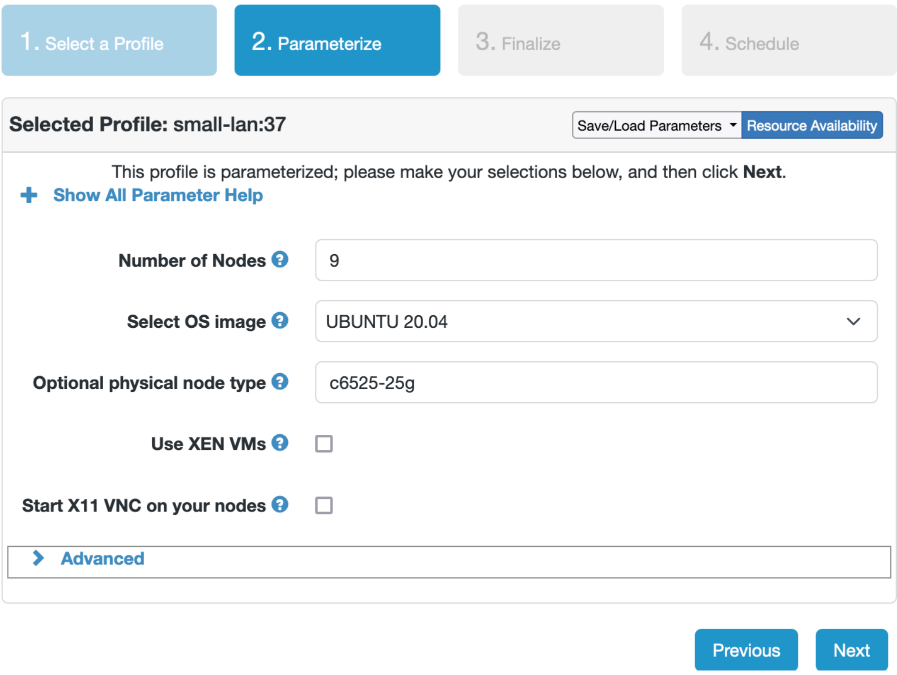
In this section, we are guided by our research question:

> *RQ T.2: How do we implement consistency/reproducibility of our testbed? What is required to set up our testbed/experiments?*

In this section, we will go through our testbed implementation and include the workflow to run, with extra details where necessary. Each layer, shown in Figure 4.1, will be discussed. Our current tool is implemented as a set of scripts in Bash that can be found on GitHub[2]. Each layer reflects a step in our tool written in Bash to prepare our cluster for experimentation.

4.2.1 Infrastructure: Preparation of the Cluster in CloudLab

To run our testbed, we first (manually) deploy our node cluster in CloudLab, which is our infrastructure layer. Our choice of CloudLab is straightforward, as the platform supports our need for reproducible research and is available to the research community. To set up CloudLab, we reserve bare metal nodes and choose the profile of the small LAN [11], as shown in Figure 4.2, which allows us to choose the number of nodes we require, the operating system we want to install (Ubuntu 20.04) and the type of node (c6525-25g). When all nodes return to the ready state of the platform, we can access the nodes using ssh. CloudLab prepares a Virtual Private Network to create a Local Area Network setup.



Selected Profile: small-lan:37

This profile is parameterized; please make your selections below, and then click **Next**.

Number of Nodes	9
Select OS image	UBUNTU 20.04
Optional physical node type	c6525-25g
Use XEN VMs	☐
Start X11 VNC on your nodes	☐

Figure 4.2: Example to configure infrastructure through the CloudLab website. Usage of nine nodes with Ubuntu and the physical node c6525-25g.

After the deployment, we (manually) copy the names of the nodes in our script and explicitly assign the roles of the nodes. We assume either three or one node to support high or low availability in the control plane. One node is set aside to be used later in the experimentation layer as a workload generator for the experiments, and we also call it the test client. The remaining nodes will be used as a worker or server node for the remaining cluster, see Figure 4.3 for the setup.

[2] https://github.com/Stvdputten/Orchestration

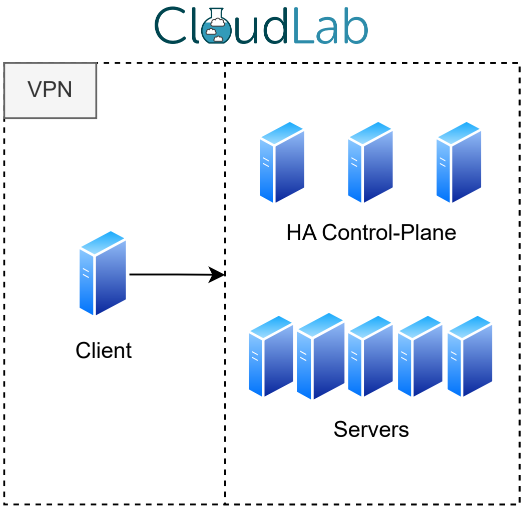


Figure 4.3: Architectural setup of the infrastructure. In total 9 nodes are used. We run 1 client node to run the workload from. The control plane is run in high availability mode, which means at least 3 nodes are used. The remaining five nodes are used as the servers. Each node is identical.

4.2.2 Orchestration: Deployment of the Orchestration Engine

This step consists of three scripts that each deploy a different container engine. We deploy only one because each orchestrator needs unique configurations to be set up, and they should not interfere with each other. First, a generic configuration script will do the initial setup as follows per node:

1. Disable the firewall
2. Install the Docker engine (version 20.10)
3. Download the benchmark suite from our GitHub repository
4. Install packages to run the benchmark tools (e.g., workload generator) and build any additional tools

To do this, we first specify which nodes will be managers, workers, or clients. The cluster uses our pre-specified configuration file to determine which node has which role and configures them accordingly. In the case of 9 nodes that would result in Figure 4.3. The next step to this is to set up our container orchestration engine.

Setting up the container orchestration engine

Our setup assumes that nine nodes are ready to be used. Each container orchestration engine, Docker Swarm, Kubernetes, and Nomad are set up using separate deployment scripts available in our GitHub repository.

Docker Swarm requires the least effort of the orchestrators to deploy. The nodes have previously been deployed with Docker in the previous step. First, we deploy our managers using the command:

```
docker swarm init --advertise-addr ip
```

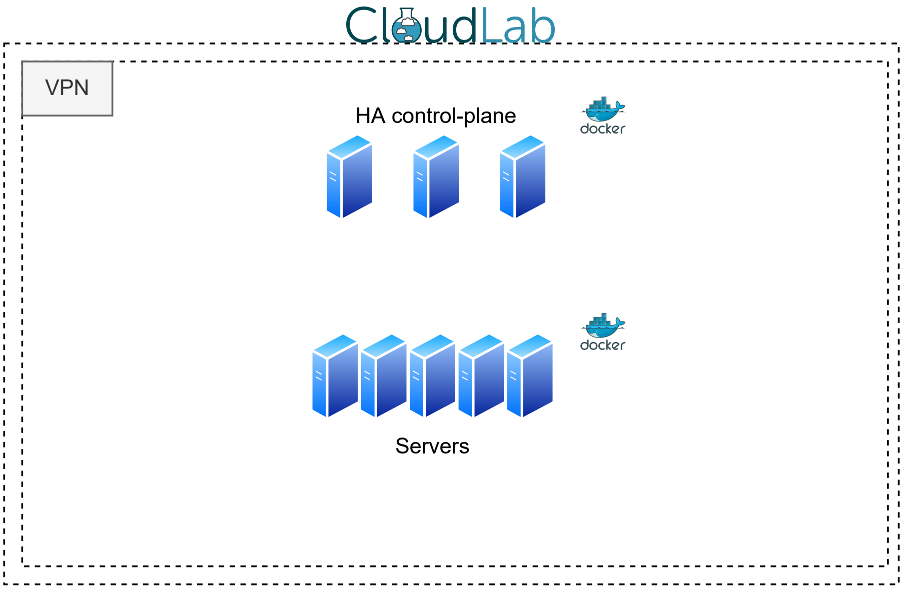


Figure 4.4: Architectural setup of the Docker Swarm infrastructure.

and advertise their private addresses in the network, which are accessible from our other nodes. The output of the first manager node can be copied to the remaining nodes to assign a manager or worker role to the cluster. Now we can use the Docker command-line tool to deploy the benchmarks and interact with the cluster. This final result can be seen in Figure 4.4 for Docker Swarm.

Kubernetes is arguably more complex to deploy. Our approach follows the documentation chapter 'Creating a cluster with kubeadm'[3]. We first deploy the manager nodes and the remaining nodes will join afterwards. The steps are as follows:

1. Installing kubelet, kubectl and kubeadm (same versions), version 1.31
2. Disable swap memory (required by kubeadm)
3. Configure networking to open ports on nodes to listen and advertise their private IP

The container network interface for container-to-container communication is set to Flannel [20]. The manager node runs the following command:

```
kubeadm init --control-plane-endpoint='ipmanager' --apiserver-advertise-address='ipmanager'
 ↪ --upload-certs --apiserver-cert-extra-sans='ipmanager' --pod-network-cidr=10.244.0.0/16
```

Kubectl is used to interact with the cluster and deploy the benchmarks. The setup is shown in Figure 4.5. Workers and managers use their internal networks to communicate, as shown in the figure.

[3] https://Kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/

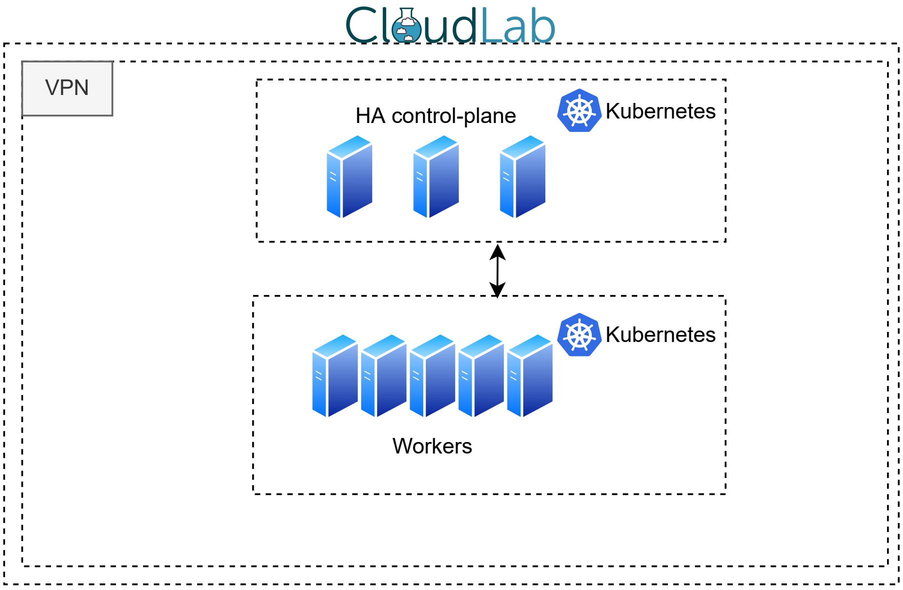


Figure 4.5: Architectural setup of the Kubernetes infrastructure.

Nomad & Consul is deployed using a tool called hashi-up[41], which allows us to easily configure our nodes to our preference. In contrast to the previous container orchestrators, we first set up the consul cluster and then deploy the nomad cluster on the node level. We install the service consul as follows:

```
hashi-up consul install --ssh-target-addr ipmanager1 --ssh-target-user user  --server
  ↪ --client-addr 0.0.0.0 --bootstrap-expect 3  --version consulversion --connect
  ↪ --retry-join ipmanager1 --retry-join ipmanager2 --retry-join ipmanager3 --bind-addr "{{
  ↪  GetInterfaceIP "device"}}"
```

When set up correctly, Nomad will identify Consul and can be installed as follows:

```
hashi-up nomad install  --ssh-target-addr ipmanager1 --ssh-target-user user  --server
  ↪ --version nomadversion --bootstrap-expect 3  --advertise "{{ GetInterfaceIP "device"}}"
```

Nomad also requires an extra configuration step that is required to enable Docker to be fully supported with the configuration 'docker.volumes.enabled'. Furthermore, the container network required extra networking configurations. A manager node is set up using a Consul DNS server so that containers can use that as a DNS resolver to find other deployed containers on other nodes. Kubernetes and Docker Swarm support container service discovery out-of-the-box.

Both Consul and Nomad have a GUI that can be used to monitor the benchmark application resources. Nomad and Consul have their separate command line interface which allows interaction with the cluster and deployment of the applications. Figure 4.6 shows the setup for Nomad and Consul where the control plane also includes the Consul DNS server to support the containers looking up other services when services are deployed on a different node.

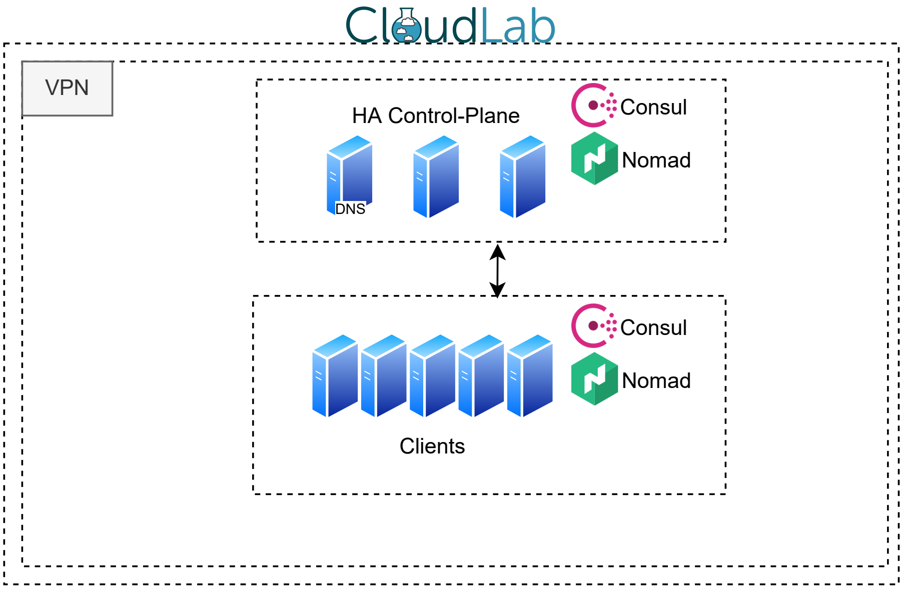


Figure 4.6: Architectural setup of the Nomad and Consul infrastructure.

4.2.3 Monitoring: Node and Container

Our testbed contains scripts for deploying monitoring solutions, which have been used as a support to debug and observe the correct usage of resources and service calls of the applications during experimentation. The monitoring tools are included in a separate script if not already built-in. As Figure 4.1 shows in the monitoring layer, Prometheus and Grafana with cAdvisor and Node Exporter can be deployed to log additional information about the nodes or containers, which we had to create ourselves. Furthermore, Jaeger is a service that is deployed as a microservice additionally to the applications to trace the API calls by the workload generator. Our Kubernetes setup incorporates the extra logs and monitoring features provided by Kubernetes, enabling us to monitor the containers. Our Docker Swarm services can be monitored using the ‘docker logs‘ command for additional information, also we used a service based on the tool swarmprom[4] to visualize the node and container CPU and Memory resource usage. The Nomad orchestrator has a built-in dashboard to monitor deployments, which is useful for monitoring resource usage for both nodes and containers.

4.2.4 Microservice: The Benchmark Suite

As a major challenge in supporting the DeathStarBench for our testbed, we had to build a wide range of deployment files. The DeathStarBench benchmark suite has been expanded to support multiple orchestrators, which was not supported at the beginning of this thesis. As an experimental benchmark suite, we had to include a long building and testing phase to prepare the benchmarks for experiments. Before we started our work, DeathStarBench only included the deployment files for Kubernetes/OpenShift and Docker. We extended each benchmark to support Nomad, Kubernetes, and Docker Swarm. In total, our work includes the creation of 36 separate deployment files. Supporting the three benchmarks with three orchestrators and four experimental configurations (baseline, vertical, horizontal, and unlimited resources). Furthermore, additional tedious and time-consuming debugging and testing, which we will briefly discuss in Chapter 7, had to be performed for each of the separate 36 deployment files to ensure a working state

[4] https://github.com/stefanprodan/swarmprom

before we could experiment. We will continue with a more in-depth explanation of the considerations of creating the deployment files and tweaking the benchmark applications that we have created.

Implementation considerations of the deployment files for the orchestrators

First, to the best of our knowledge, we tweaked and configured the benchmarks to be representative and fair use for comparative research for the container orchestration engines. We achieved this by tuning our scripts and/or deployment files for consistency where possible. For example, images are set to a specific version by (re)building container images for each orchestrator setup, explicitly specifying the available resources to the containers. In each deployment file, the resources have been explicitly defined, to limit usage of resources only to 1 CPU and Memory to 1 GiB. Each deployment service file includes explicit Docker image versioning (e.g. stvdputten/media-microservices:swarm) that we copied and/or built separately on a separate Docker Hub account to ensure the reproducibility of the work. Docker Engine (20.10), Kubernetes (1.31), Nomad (1.1.6) and Consul (1.10.3) have been versioned, as shown in Table 4.1. We use identical node types, which is typical. Our testbed tool benchmarks were forked from the version of DeathStarBench on 27 May 2021.

Table 4.1: Overview of orchestrator specifications.

Specifications				
Orchestrator	Kubernetes	Nomad	Consul	Docker Swarm
Version	1.31.1	1.1.6	1.10.3	20.10
Deployment tool	Kubeadm	Hashi-up	Hashi-up	Bash scripts
Container runtime	containerd	Docker	Docker	Docker

Second, the deployment files were created to match the DeathStarBench original papers' [13] microservice architecture as closely as possible. Our new Docker Swarm files are based on the original Docker Compose files. Our new Kubernetes files are based on the available OpenShift, Docker Swarm, or Docker Compose implementation. Our Nomad job files have been newly created to match the Docker Swarm or Docker Compose setup as closely as possible. Each orchestrator has many components to take into account, such as service discovery, DNS resolution for internal container-to-container communication, type of networking, storage choices, and all other general default options that might differ between orchestrators. Our contribution to extending the number of supported orchestrators in DeathStarBench is shown in Table 4.2.

Table 4.2: An overview of the orchestrators that were supported by the DeathStarBench suite per application. The orchestrator that had to be introduced in this work per application.

Benchmark	Original Container Orchestrators	Introduced Container Orchestrators
Social Network	Docker Compose, Docker Swarm, OpenShift	Kubernetes, Nomad
Media Microservices	Docker Compose, OpenShift	Docker Swarm, Kubernetes, Nomad
Hotel Reservations	Docker Compose, Kubernetes, OpenShift	Docker Swarm, Nomad

The general approach to port the different benchmarks to our testbed is shown in Figure 4.7. The figure presents the process we took to create each deployment file. For each of the three benchmarks, we first create the Docker

Swarm YAML files if it did not exist yet. If not, we would mostly manually create the Docker Swarm files from the existing Docker Compose, otherwise some regular expressions and conversions tool could help along. Second, we check whether all container resources have been explicitly set. If not, we explicitly set the number of cores to 1, and the RAM to 1 GiB. Only the front-end service of the benchmarks, where we run our workloads, uses 4 cores and 4 GiB to not provide a bottleneck for the experiments. We continue to define each service to only have one instance/replica and ensure the Docker image is set to explicit versions found on the public repository of Docker Hub. We also use the local node volume when we are required to specify a storage location for the different orchestrators.

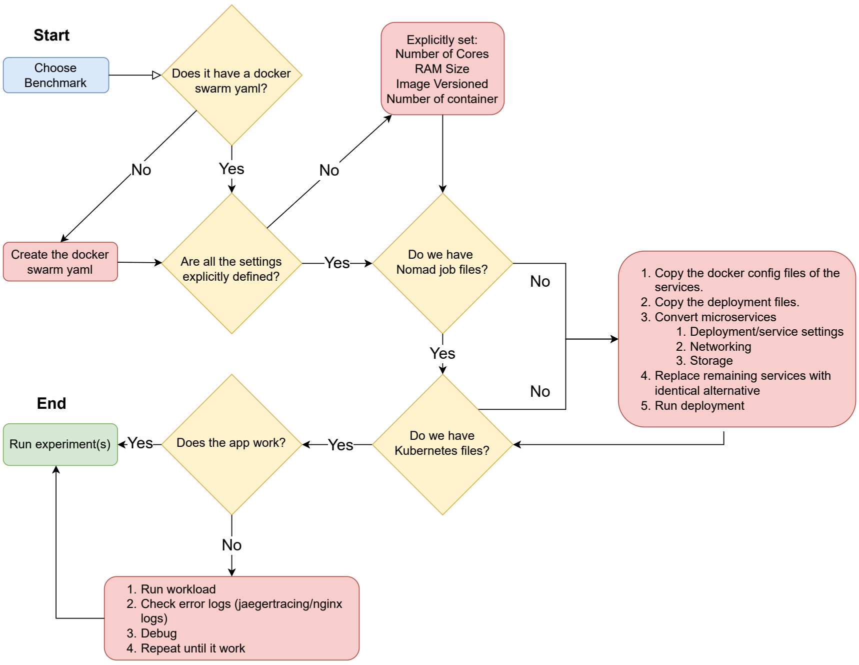

Start
Choose Benchmark
Does it have a docker swarm yaml?
Explicitly set:
Number of Cores
RAM Size
Image Versioned
Number of container
No
Yes
No
Create the docker swarm yaml
Are all the settings explicitly defined?
Yes
Do we have Nomad job files?
No
1. Copy the docker config files of the services.
2. Copy the deployment files.
3. Convert microservices
1. Deployment/service settings
2. Networking
3. Storage
4. Replace remaining services with identical alternative
5. Run deployment
Yes
No
End
Run experiment(s)
Yes
Does the app work?
Yes
Do we have Kubernetes files?
No
1. Run workload
2. Check error logs (jaegertracing/nginx logs)
3. Debug
4. Repeat until it work

Figure 4.7: The general approach to creating the deployment/service files for all three orchestrators: Nomad, Docker Swarm and Kubernetes.

As Nomad deployment/job files are similar syntactically to Docker, we first manually create their files. At this point, we ensure that everything is working with the Docker Swarm files by deploying and testing the applications. We continue and create the Nomad job/deployment files. Nomad with Consul requires an extra step to support services to communicate by using a Consul manager node as the DNS server for inter-container communication. We also constrain the front-end web server (usually Nginx) to the first worker node.

For the Kubernetes deployment files, we use kompose[5] to convert Docker Swarm files to Kubernetes. We confirm that all benchmarks work as intended with a minimal stress test and by exploring the microservice logs.

The next step requires a separate deployment file for each of our four experiments (baseline/high availability on, vertical, horizontal, high availability off), which will be discussed in Chapter 5. Vertical and horizontal scaling are straightforward. Our solution is to copy the original deployment files and change the required settings. In the case of vertical scaling, we double the CPUs and Memory resources (e.g. 1 core becomes 2 and 1 GiB becomes 2 GiB). With horizontal scaling, we change the number of replicas from 1 to 2 for all microservices. The configurations of some

[5] github.com/Kubernetes/kompose/releases

containers have also been modified to ensure consistency between services, such as Nginx, Memcached, and MongoDB. Usually, the tinkering involved library calls or explicit service discovery renaming. For a complete overview of the exact specified resources per microservice, see Appendix D. In Appendix F snippets of the orchestrator and benchmarks are shared. The Docker images that we built are publicly available on Docker Hub[6]. The built Docker images only include the ones that are exclusive to the DeathStarBench (e.g. stvdputten/media-microservices:swarm). The exceptions are the openresty-thrift image built by the original author, a few that could not be built from source, and the other open-source public images; see Appendix F for the overview. A final note, we do try to limit the resources of the tracing service because that might interfere with correctly reporting and monitoring the API calls. The overhead introduced by the Jaegertracing service was assumed to not interfere with the measurements, since the resources required by Jaeger, as observed during preliminary experiments, were not in any form close to starving the nodes for resources.

4.2.5 Experiments: Preparation, Experiments, and Results

The final step in our process is to prepare the test client to perform our experiments, which we again include in a script. This is straightforward once the application has been deployed correctly. Each benchmark is stressed using the wrk2 load generator and traced using Jaeger. Each application has a front-end service that is exposed using the orchestrator tools and that we made accessible to the test client.

Wrk2 [45] is a load generator included in the DeathStarBench that provides constant throughput with accurate and correct latency. The DeathStarBench researchers have made slight modifications to wrk2 to integrate it into the DeathStarBench suite. Wrk2 sends a fixed number of requests per second for a defined duration. The latency measurements are +/- 1ms granularity, which the original developer mentions due to the behaviour of the OS sleep time. The specific loads generated for each benchmark are defined in Lua. No modification has been made to the type mixed workload in the DeathStarBench suite for our experiments. The workload scripts have been modified to fit our parameterisation. The load generator allows the user to define the concurrent connections, the amount of requests, the duration of the runs, and the threads. The load generator outputs the latency for different percentiles and reports statistics: maximum, mean, standard deviation of the latency, input throughput/requests, and returned throughput in requests per second. Our results focus on the 99^{th}/ P99 percentile tail latency and throughput, though it is possible to use other percentiles as well.

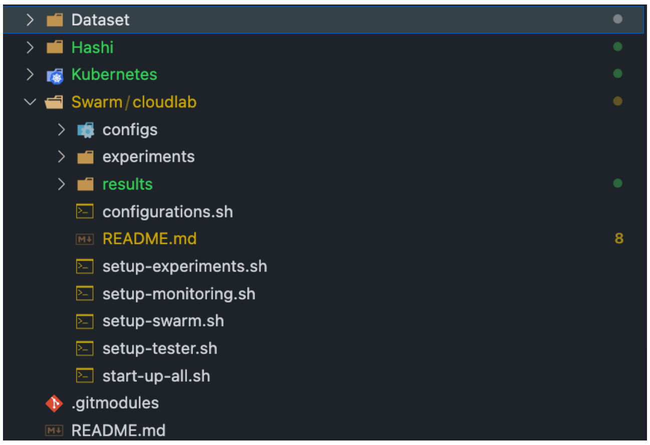
Our testbed is capable of performing experiments in three modes for running our applications: high availability (on/off), horizontal scaling (on/off), and vertical scaling (on/off). Our baseline is defined as having high availability enabled while the other parameters are disabled. There is also an option to assign limited resources or allow unlimited resources for the application. The vertical and horizontal parameters cannot be enabled simultaneously; in such cases, our scripts will activate one mode and skip the other. However, it is possible to disable high availability and still run the vertical or horizontal scaling deployment files with the applications. Moreover, our current tool permits users to enter predefined parameters into scripts or command lines, which will then carry out the specified range of requests for the experiments with the selected workload and generate output accordingly. Appendix C shows the code that implements the workloads, and an example of the output after the experiment has been run at Section A.2.

[6] hub.docker.com/repository/docker/stvdputten/social-network-microservices

We extended a simple Python script[7] to extract multiple variables and create multiple datasets from the results. The data can be visualised with the previous Python tool or by using our included Jupyter Notebook scripts.

4.2.6 Workflow: How to perform research in our Testbed

This section explains the DIY setup of our testbed. For a more detailed explanation of the creation of the testbed and implementation details of our testbed, see our previous sections.



```
> Dataset
> Hashi
> Kubernetes
v Swarm / cloudlab
    > configs
    > experiments
    > results
      configurations.sh
      README.md                              8
      setup-experiments.sh
      setup-monitoring.sh
      setup-swarm.sh
      setup-tester.sh
      start-up-all.sh
  .gitmodules
  README.md
```

Figure 4.8: The repository containing the scripts to run the testbed.

After the infrastructure is operational, our testbed tool allows users to enhance the scripts to execute the benchmarks, choosing the experiments to conduct. Figure 4.8 presents the setup of our testbed tool once the user has downloaded it. The directories to our testbed are used as follows:

- *Dataset*: The dataset contains the throughput and tail latency measured in this thesis with additional parameters for the experiments, including the Python scripts to visualize the results.
- *Swarm—Nomad—Kubernetes/cloudlab/configs*: input worker/manager nodes in the IPs file and the test client in the remote file. Input the roles of each node into the roles file.
- *start-up-all.sh*: Will run configurations.sh, setup-swarm.sh, setup-monitoring, and setup-test.sh in that order. Similar for the other directories named ‘Kubernetes/cloudlab‘ and ‘Nomad/cloudlab‘.
- *Experiments*: contains the experiments.sh, which are a set of experiments used in Chapter 5 and Chapter 6.
- *Results*: Will contain the results generated by the experiment the user has run, sorted by benchmark, experiment, and additional information on which parameters have been used during the run in the file description.

We have included all our experiments, four experiments per orchestrator, and eleven additional configuration experiments for Docker Swarm to perform some preliminary experiments for creating a similar baseline configuration. The experiment files can be used by researchers as a template to run experiments, having the option to choose which orchestrator to use by writing the environment variables inside the experiment scripts. The generated results can, for example, be compared with our dataset for comparative analyses. Furthermore, the directories are structured in a way that

[7] github.com/Stvdputten/hdr-plot

researchers can build on this testbed by copying existing deployment scripts and rewriting components to fit their unique infrastructure, orchestrator, monitoring tools, microservices, and/or experiments.

To give an example of how one would approach extending this. Let us discuss two use-cases:

1. "I would like to run an Swarm cluster on Azure for the DeathStarBench": The user would duplicate Swarm/-cloudlab, see Figure 4.8, to 'Swarm/azure'. Starting from here, we suggest approaching the deployment and configuration files one by one. If you would like to include the *infrastructure* deployment of your resources in Azure, you could change the 'configuration.sh' file to ensure that the resources have been created and are accessible. The 'configs' folder should include your ssh keys or other secrets that are required by the general scripts and specify the roles of your noes. Then, deploy the orchestrator on your Azure resources by extending 'setup-swarm.sh'. Furthermore, 'setup-monitoring.sh' is optional if it makes sense to add additional monitoring tools to your cluster. Then 'setup-experiments.sh' is the layer that deploys the benchmark to your cluster and configures the environment to be able to benchmark the applications. The experiment workloads need to be changed or added in the 'experiments' folder to fit. These steps are just some general guidelines, and additional tweaking or testing per script might be required to ensure everything is running correctly. If everything is working, 'start-up-all.sh' should be able to run your benchmark with your new resources.

2. Another use case would be to add an additional benchmark, assuming that it is in CloudLab, one can change the 'setup-experiments.sh' to include another parameter which branches to your benchmark that includes the deployment of your benchmark on docker swarm. It would also require manual extension of the workload and experiments scripts in the 'experiments' folder. One would copy any of the 'experiments.sh' and change the experiments to fit their new benchmark.

Ideally, both examples would be able to run using only the 'start-up-all.sh' by being generic enough and providing the parameters and/or configurations through the command line, e.g. '*./start-up-all.sh -benchmark DSB -Requests 1000 -NumberOfRuns 5 ...*'.

Chapter 5

Performance Exploration with Docker Swarm

The experiments are divided into two chapters. In this chapter i) exploring and tuning the testbed, w.r.t. the workload parameters for experimentation, and ii) assessing the limits w.r.t. *tail latency* 99^{th} *percentile and throughput* of the current setup for a baseline with Docker Swarm as a reference. In the second chapter, Chapter 6, we further experiment with performance benchmarking, for which we are running the experiments with all orchestrators.

5.1 Tuning the Testbed

In this section, we are guided by our research question:

> *RQ T.3: Using our testbed, how can we ensure consistency in our experiments? What pre-tuning is required to conduct experiments, and what performance characteristics can we find?*

With the help of our testbed, we can carry out rigorous and extensive research on performance. However, the breadth of benchmarking the applications is still costly and time-consuming. This is due to the unavoidable large configuration parameter space to choose from for each of the three applications for each orchestrator when running multiple runs. To keep the time to run the experiments realistic and practical, we dedicate our first part to finding a realistic baseline and parameter scope using Docker Swarm. In the second part of our experiments, we will look at the impact of various configurations, such as the performance of Kubernetes, Docker, and Nomad under different conditions, including horizontal/vertical scaling and the need for high availability.

5.1.1 Experimental Setup

All our experiments are run on CloudLab infrastructure using nine c6525-25g[1] nodes, five workers, and three managers with one client node for testing purposes to run the workloads from. Each node has identical hardware, shown in

[1] https://docs.cloudlab.us/hardware.html

Table A.2, is configured with Ubuntu 20.04 and is connected to a LAN network. Each node has 16 cores, 128GB of memory, and two 480GB of SSD storage. The container orchestrator versions were set to the most recent version at the time of running our experiments; see Table 4.1. For more information, see also Appendix A. Our experiments use the *social network (sn)*, *media microservices (mm)* and *hotel reservation (hr)* applications from the DeathStarBench benchmark suite. The applications are deployed *without limiting* the resource usage, unless otherwise stated. The orchestrator is scoped to Docker Swarm for tuning. The workloads are run with wrk2 included with the DeathStarBench, only modified to work with our testbed, which produces a constant load per second and measures latency and throughput. The workloads simulate user actions for each of the benchmarks. The social network uses a mix of requests: 60% of the requests read the homepage, 30% read user timelines, and 10% composed user posts. For the media microservice, the workload consists only of composing reviews. The hotel reservation consists of 60% searching for a hotel, 39% recommending a hotel, and 1% reserving a hotel. Each experiment has a workload running for a duration of 30 seconds. Each experiment is run only once, unless otherwise stated. In the case where we run multiple runs, we report the mean latency and the error bar using the standard deviation.

5.1.2 Evaluation Methodology

For evaluation, we mainly consider the 99^{th} *percentile of latency* as the performance metric; other percentile tail latency metrics can also be easily used from the results. Our evaluation is interested in the 'breaking point', the point when the applications show significant performance reduction (e.g. going from milliseconds to seconds for the requests to be processed). Unless otherwise stated, our experiments are run by deploying each application in Docker Swarm, waiting until the application is ready (the front end is deployed and reachable using a curl command from the test client). The application deployment files are not bound to resource limits at this stage of experimentation. The workload is applied for a steady 30 seconds, per run. Whenever we refer to requests, it implies requests per second unless otherwise specified. The range of our workload requests is 500 up to 20.000 per second, if necessary. During that time, we measured the tail latency and throughput. To collect data, we use *wrk2*. The experiments are automated using our experiment scripts.

Four experiments are conducted, each exploring the effect of performance and assessing whether consistency exists or extra tuning is required in our next phase. All have a focus on exploring variance in our applications. Experiment A focusses on exploring the parameter space of our workload generator. Experiment B looks at running multiple test clients. Based on Experiments A and B, Experiment C stresses the benchmarks to their limits. Experiment D looks at the impact of a redeployment, if any, between experiment runs.

A comprehensive overview of these experiments, including descriptions of the configurations and supporting figures, can be found in Appendix A. For a complete overview of all the configurations used, refer to Appendix A Table A.1 which gives an overview of all the parameters used in all experiments. Our initial parameters are the "initial" parameter settings in the benchmark suite, as shown in Table 5.1.

5.2 *Experiment A.* Investigating the Workload Parameters

Experiment A has the goal of exploring the initial performance metric using tail latency to find a standard set of workload parameters. As the original paper lacks the initial experimentation design for the workload generator [13], we will define a new set of workload parameters. The test client, uses the wrk2 workload parameters shown in Table 5.1 and include *threads*, *connections*, *requests*, and *duration* as parameters. The initial values are found as default in the DeathStarBench. We will explore increasing the range of the previously mentioned parameters, except the duration.

Table 5.1: Overview of the workload configurations for the test client.

Parameters	**Description**	**Initial**	**New**
Threads	Number of threads used to execute workloads	4	8
Connections	Open connection to the web server	8	512
Requests	Amount of requests to the application per second	200	500
Duration	How long do we run the experiment	30s	30s

The tail latency starts at: 12.7ms for hotel reservation, 9.29 ms for media microservice, and 7.79 ms for social network under a load of 200 requests per second (req/s) and with the initial parameters, as shown in Figure 5.1. It seems that all the benchmarks work with our current load and settings.

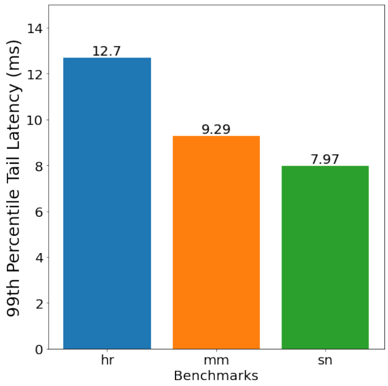


Figure 5.1: First tune, exploring tail latency of each application, using Docker Swarm.

Our next experiment, as visualized in Figure 5.2, includes a wide range of requests, threads, and connections while keeping the remaining parameters fixed to our initial parameters. The first left figure has an increasing number of requests from 500 to 3.000 req/s for each application. The range was incremented until we noticed a 'breaking point', which was reached first by the media microservice, then by the social network. Social network and media microservice both have a noticeable effect on latency starting from 2.000 req/s. The hotel reservation remains capable of handling the loads throughout. The hotel reservation benchmark has some unusual latency in the first run, but it does not show any noticeable difference after the initial start. The second figure in the middle of Figure 5.2 shows the relationship between threads and latency, run under 200 req/s. Again we vary the number of threads per worker with 200 req/s, we have no noticeable effect when changing the thread count. The last figure on the right, which considers the connection, does not show any noticeable tail latency.

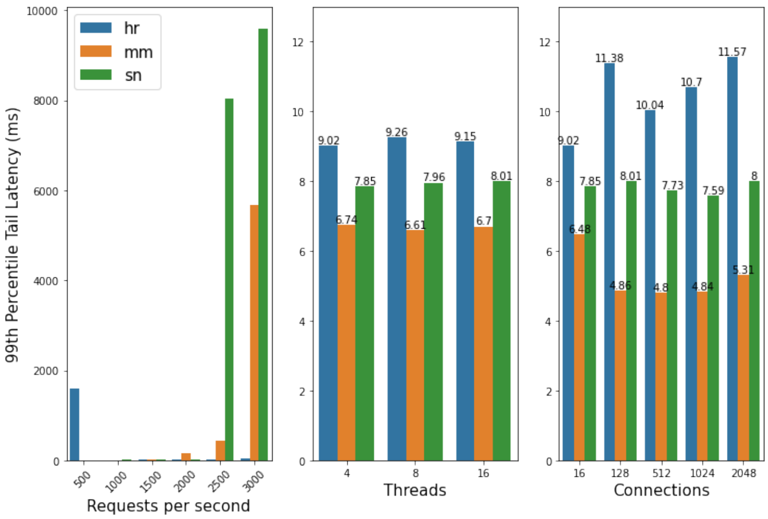
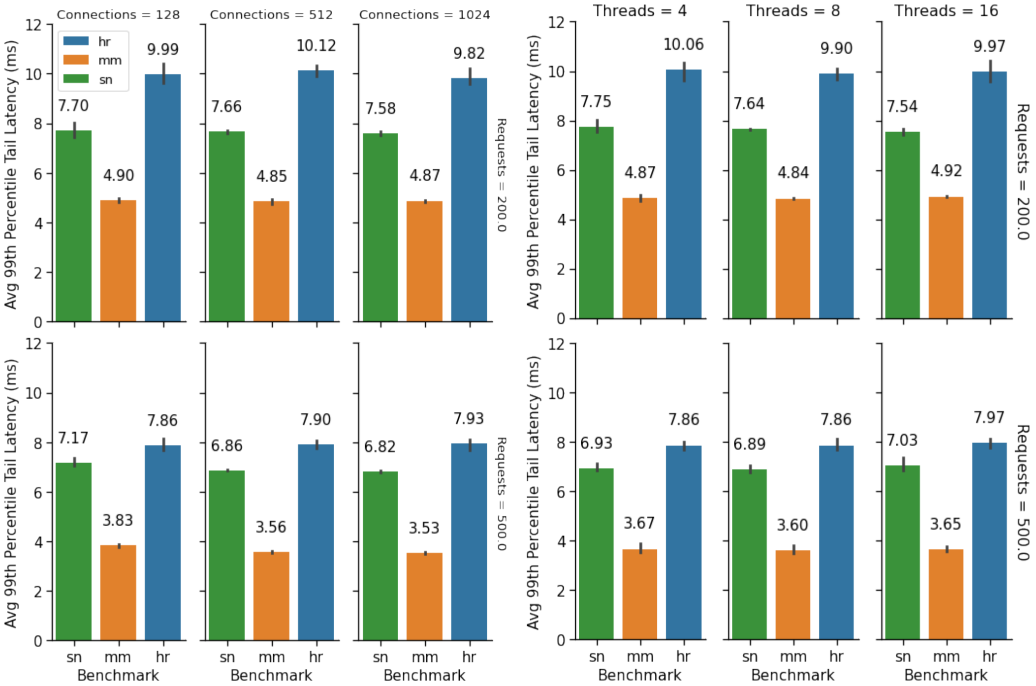


Figure 5.2: Effect of requests, connections and threads on latency per benchmark. The left requests experiment is run for 30 seconds per load. The threads and connections experiment are run with 200 req/s for 30 seconds.

Based on our previous results from Figure 5.2, we can see that only the request parameters primarily affect the tail latency, as the performance for threads and connections are stable in comparison. We rerun the previous experiment increasing the load slightly, from 200 req/s to 500 req/s, to see if minor load changes will lead to changes for connections and threads on tail latency. This time we explore only the connection, 128, 512 and 1024 as these seem closer to what we expect to be realistic to use for further experimentation.



(a) Connections ranged from 128 to 1024.

(b) Threads range 4 until 16

Figure 5.3: Effect of connections, threads on latency with 200 req/s and 500 req/s load on P99 tail latency.

Figure 5.3 shows us varying threads and connections under two load levels. This time we ran the experiment 3 times. We use three values of connections and threads but use two load levels, 200 req/s and 500 req/s as top and bottom row. The figures include the mean of tail latency measurements per benchmark and the standard deviation. In Figure 5.3a our connections range from 128 to 1024 and in Figure 5.3b our threads range from 4 to 16 threads.

When comparing the load of 200 req/s of the top row the connections and threads are stable as they stay the same and the error bar remains small, indicating that threads and connections do not affect tail latency. When comparing 500 req/s, in the bottom row, we see a similar trend where the latency stays consistent. We conclude that the threads and connections, do not really matter at this point, so we change the default workload parameters to 8 threads, 512 connections, and 500 req/s for now.

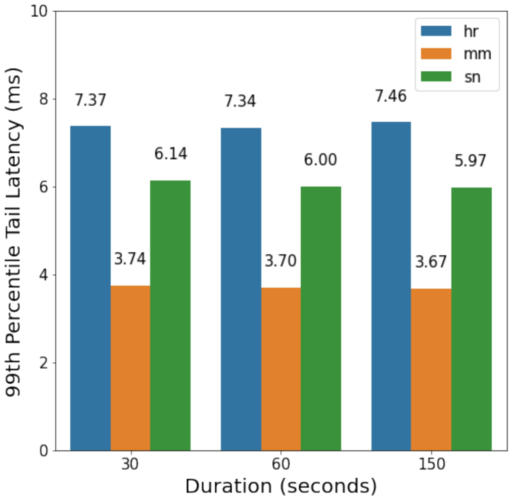


Figure 5.4: The effect of running the workload for a duration of 30, 60 and 150 seconds on the tail latency.

Our last experiment for the workload parameters is the duration, as shown in Figure 5.4. So far we assumed that 30 seconds is enough time to measure the tail latency, and to compare we will run a load of 500 req/s for a duration of 60 and 150 seconds. The tail latency is again consistent across the board, with little variance. To be more time efficient for further experiments and runs, we will continue to use the smallest value of 30 seconds.

Conclusion A: The first few test runs have not shown noticeable differences, but shows that the benchmarks are mostly affected by the input loads, requests per second, which is expected. No noticeable performance anomalies are observed except for the first hotel reservation run, which indicates that a cold-start effect is sometimes applicable.

5.3 *Experiment B.* Examining the Impact of 3 Test Clients on System Performance

Experiment B compares the setup of three test clients when they are benchmarking the same application. In case one of the test clients results in a different tail latency, we could further investigate if there is a single point of failure. A total of nine runs have been performed, three runs per client for each of the three benchmarks

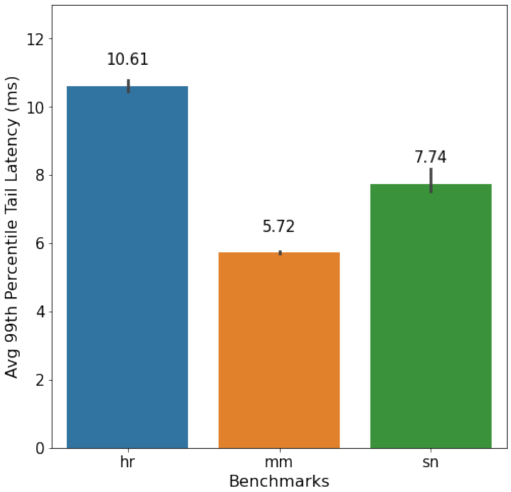


Figure 5.5: Three test clients. Multiple clients on the tail latency, using 500 requests per second for a duration of 30 seconds.

Our results are shown in Figure 5.5, where we report the average tail latency. The results for each benchmark are as follows (in order of social network, media microservice and hotel reservation) shown in Table 5.2.

Table 5.2: Tail latency for each benchmark, req/s is 500

Test Client	Social Network	Media Microservice	Hotel Reservation
Client 1	8.17ms	5.76ms	10.59ms
Client 2	7.52ms	5.69ms	10.79ms
Client 3	7.54ms	5.71ms	10.45ms
Mean	7.74ms	5.72ms	10.61ms
Variance	0.0911	0.000867	0.019467

Table 5.2 indicates similar results across clients, as the results are within 1 ms of each other per application. The margin is within the expected granularity of the wrk2 measurements. The variance is also small, below 1 ms. The only noticeable take-away result is that the media microservice seems to have the lowest latency on average, followed by the social network and then hotel reservation based on all previous experiments and runs.

Conclusion B: The setup of multiple test clients does not affect the performance of our benchmarks (Figure 5.5). We will continue to use one single test client.

5.4 *Experiment C.* Stress Testing the Microservices

In experiment C, our primary goal is to find a breaking point of the applications, when we incrementally increase the load until we see noticeable changes in tail latency. Now we have configured our experimental setup with workload parameters, based on Experiments A and B which resulted in the new workload parameters 8 threads, 512 connection with loads starting from 500 for 30 seconds. This reference will help us find the expected initial limits for the applications. Our results are shown in Figure 5.6. We repeated the experiment five times per benchmark.

In Figure 5.6, we can see that the social network, media microservice application and hotel reservation are run in the range of starting from 500 req/s up to 15.000 req/s. The social network shows performance problems after 6.000

req/s as the tail latency goes into the seconds. The media microservice has a breaking point above 2.000 req/s. Hotel reservation seems to be quite robust against our workloads, but does increase slightly when taken to the 15.000 req/s.

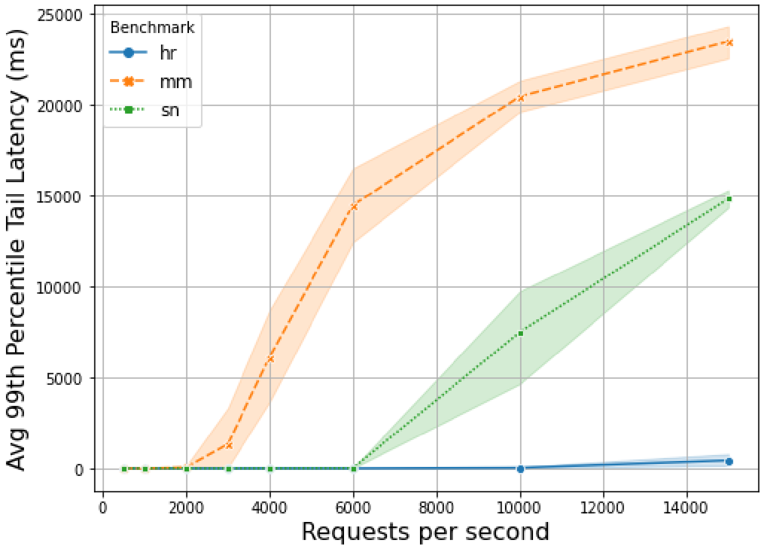


Figure 5.6: The tail latency in milliseconds for the benchmark, running from 500 to 15.000 req/s. Experiment has been repeated 5 times per benchmark. Mm breaks after 2.000 req/s, sn after 6.000 req/s and hr requires more than 15.000 req/s. The shades represent the 95% confidence interval.

The experiment shows the breaking points of our benchmarks based on tail latency. Furthermore, we can see that the media microservice breaks first, then the social network second, and hotel reservation last under increasing loads. This might be due to limitations of the Nginx web service, which serves as a front-end of both the microservices of the media and the social network. In comparison, hotel reservation uses a Go-written front-end web service. We will use the breaking order of this experiment as the reference for the next experiments when comparing orchestrators.

Conclusion C: The applications rank robustness in order of breaking: media microservices, social network and finally hotel reservation in the case of Docker Swarm.

5.5 *Experiment D.* Redeployment and Time

Experiment D aims to evaluate the impact of time and frequent application redeployment on the tail latency. There are multiple reasons to consider this experiment. Firstly, the time between experiment runs and redeployment can introduce variability in the state of the applications. Secondly, we need to determine whether a cold-start effect is a given for the applications to reach a steady state. This could influence the consistency and reliability of our results. To evaluate this, we experimented with ranges similar to those of the previous experiment C. With time/temporal effect, we leave an additional 60 seconds between each repeated stress test. With redeployment, we redeploy the applications between each test using the orchestrator as if it were a completely new experiment.

Our results are visualized in Figure 5.7. We repeat the experiment five times per benchmark. The requests start at 500 req/s and increase up to 15.000 req/s. The column, shows the baseline as shown in Figure 5.6 the previous experiment. The 2nd and 3rd columns show the time and redeployment experiments.

We are once again interested in the point where the tail latency increases and around which load. The social network,

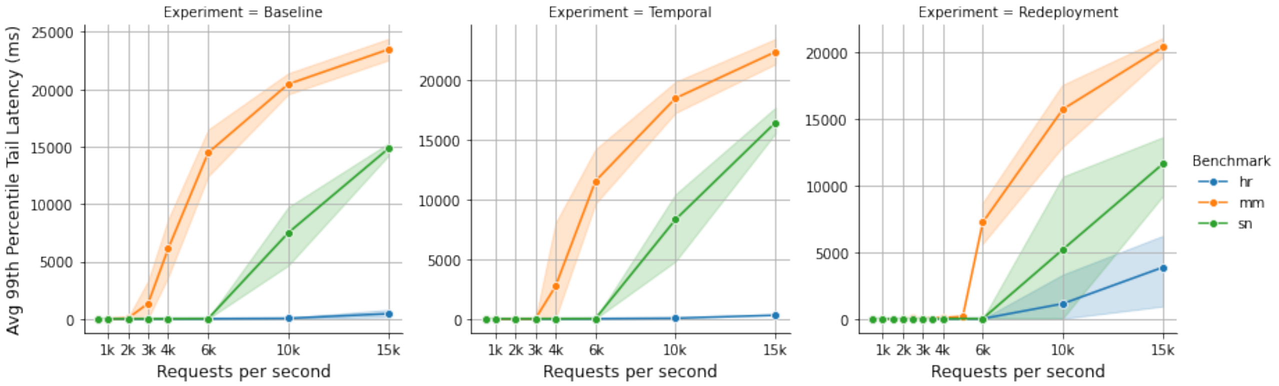


Figure 5.7: The left figure (baseline) is the same as the image of Figure 5.6, the middle image shows the effect of time and the right image shows the effect of redeployment on latency. Each experiment has been run five times, per benchmark. The tail latency P99 is shown in milliseconds.

seems to stay reliable around 6.000 req/s before it breaks in all cases. The results indicate that the breaking point for the media microservice application initially shows a breaking point between at 3.000 req/s based on our temporal experiment. There is also a shift of the breaking point to a load of more than 5.000 req/s considering redeployment, indicating that for media microservices there can be differences between runs. For the hotel reservation application, the baseline breaking point falls between 6.000 and 15.000 req/s for the redeployment, with both the baseline and time experiment exhibiting a similar breaking point starting somewhere between 10.000 to 15.000 req/s. Again, there is a noticeable breaking limit change between runs.

Conclusion D: The results show that the benchmarks respond differently to timing and redeployment. We can derive that the social network is stable in its performance, while media microservices and hotel reservation show a shift both negative and positive to the breaking point. This might indicate that there are factors of variability that we do not yet control. The overal trend to see which applications breaks first seems to remain similar.

5.6 Closing thoughts on the Docker Swarm testbed experiments

When we started these experiments, we were guided by the following research questions:

> *RQ T.3: Using our testbed, how can we ensure consistency in our experiments? What pre-tuning is required to conduct experiments, and what performance characteristics can we find?*

Experiments A-D have been run to establish the baseline performance and exploration of our testbed. Our preliminary results of the experiment led us to choose a set of default workload parameters. Our initial experiments with Docker Swarm show that some cold-start effects can affect the hotel reservation application. We also have observed and empirical evidence of what load the applications can handle and at what point they break comparing each application, when not limited by any resource limits in Docker Swarm. The microservice media is expected to break first, then social networks usually show a performance reduction when stressed, followed by the hotel reservation. Further exploration also showed some effect on performance due to time between experiments and redeployment, which means that breaking limits can change when the state of the applications are affected by time or redeployment. However, we have observed a consistent pattern in the breaking of the applications, which will be taken into account in our subsequent experiments. In the next chapter, we will proceed by comparing the multiple orchestrators.

Chapter 6

Orchestrator Experiments

In this chapter, we are guided by our research questions;

> *RQ T.4: Using our testbed, are there performance differences in the orchestration tools in various scaling scenarios?*

In this chapter, the second part of our experiments, we expand our experiments on the testbed to include all orchestration tools. As variability is a major part of this thesis, we scope our experiments to explore variability between orchestrators. Our experiments compare the various scaling methods (the baseline with high availability on, vertical scaling, horizontal scaling, and high availability off). The experiments will show the performance metrics of each application per orchestrator, comparing Docker Swarm, Kubernetes, and Nomad with the help of our testbed, then we focus on one microservice benchmark where we explore more in depth, and finally we finish with an overall comparison.

6.1 Experiment Setup

The experimental setup is identical to the previous setup, described in Section 5.1, however, where applicable the changes will be explicitly mentioned. The orchestrators on the testbed are all explored: Docker Swarm, Kubernetes and Nomad. Our experiments are run at least once, and where applicable, averages are shown. Our applications in general have been resource limited for each orchestrator and benchmark to make the comparison with equal resources possible.

6.2 Evaluation Methodology

To evaluate, we will run each experiment one time, for four scaling scenarios using performance metrics, tail latency 99^{th} percentile (P99) to find the limits through a breaking point in performance. The default settings for the parameters for scaling are shown in Table 6.1. Our baseline (high availability is on) uses eight nodes, of which three are the control plane and five are the worker nodes with an additional test client to run the workloads from. When high availability (HA-off) is turned off, we only use one node in the control plane instead. Horizontal scaling (HS) scales the replicas

Table 6.1: Overview of the experiments per orchestrator

Experiment	Description
Baseline	Default Container resources have a limit CPU= 1 and Memory = 1GiB, HA is on
Horizontal scaling	Container count is 2 for each service
Vertical scaling	Container resources are scaled 2x of the default
High availability off	Control plane consists of one node, instead of three

per container to two, and vertical scaling (VS) scales the container resources to two times. Our experiments will stress the systems until they break, and we will report the changes in performance for each orchestrator and between each of the orchestrators. We generally run workloads with 8 threads and 512 connections for 30 seconds using workloads as previously mentioned in subsection 5.1.2 using various constant-throughput loads. The overview of tail latency, throughput, and workload input can be found in Appendix D. The three applications have their resources set so that each container has 1 core and 1 GiB of memory except for the front-end service or the Nginx front-end service which gets 4 cores and 4 GiB of memory, excluding changes made when we scale the resources.

At first, 36 runs are performed, 12 per orchestrator, with for 4 scaling scenarios 3 applications. The exact nodes used during the experiments can change, as there is a pool of more than 100 nodes available at any time, but will usually always stay the same once deployed. The nodes are redeployed, when we run the high availability scenarios, changing nine nodes to seven nodes, and when we change from orchestrator as we do not run multiple orchestrator in one cluster. Experiment A has a focus on the Docker Swarm. Experiment B has a focus on Kubernetes. Experiment C looks at Nomad. Experiment D only compares the orchestrator results available from A, B, and C to create a separate comparison between orchestrators. Experiments E, takes the social network benchmark and does 5 runs, for increasing loads per orchestrator to create some statically significant results as validation of our testbed results. In the following sections, we present the most relevant findings from these experiments along with supporting figures.

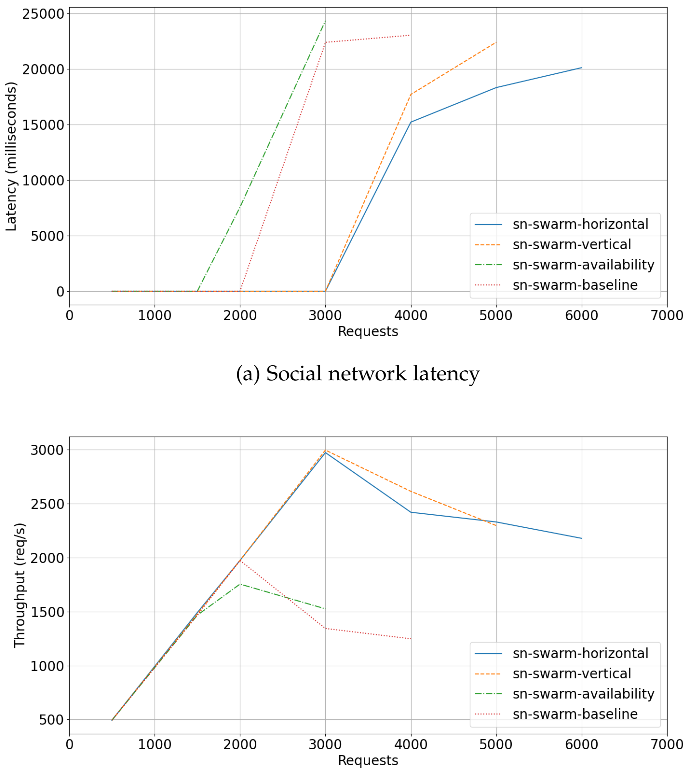
For a complete overview of the performance between the orchestrators, we refer to Section 6.6. Additional orchestrator scenarios are visualised at Appendix B. The social network (sn) container resources can be found in Table D.1, media microservices resources (mm) at Table D.2 and Table D.3 for hotel reservation (hr) resources.

6.3 *Experiment A.* Docker Swarm Performance

In Experiment A, we will look at the performance of Docker Swarm. A complete overview of the results is shown in Section B.1. Our Figure 6.1a, Figure 6.2a and Figure 6.3a show different tail latency P99 results. Our Figure 6.1b, Figure 6.2b and Figure 6.3b show the difference in throughput for applications deployed with Docker Swarm with different applications.

After deploying the cluster using the usual nine node setup. We run a range of workloads for a duration of 30 seconds, running from a single test client to measure the tail latency and throughput. The workload generator has increasing loads for each application to run until we can clearly see a noticeable latency increase; this is done for each application in the different scaling scenario. For the various scenarios, a performance increase would be expected for the vertical and horizontal scaling, while the high availability should realistically show no difference or decrease in performance. Let us see how realistic microservices, with real-world workloads, perform according to our testbed.

Social Network: First, Figure 6.1a shows the tail latency, indicating that the application can handle up to 2.000 req/s, our baseline reference of 100%. Increasing the resources 100% through *VS* and *HS* causes a 50% increase in performance at a load of 3.000 req/s. Furthermore, *HA-off* causes the application to have a performance decrease of 25% to 1.500 req/s.



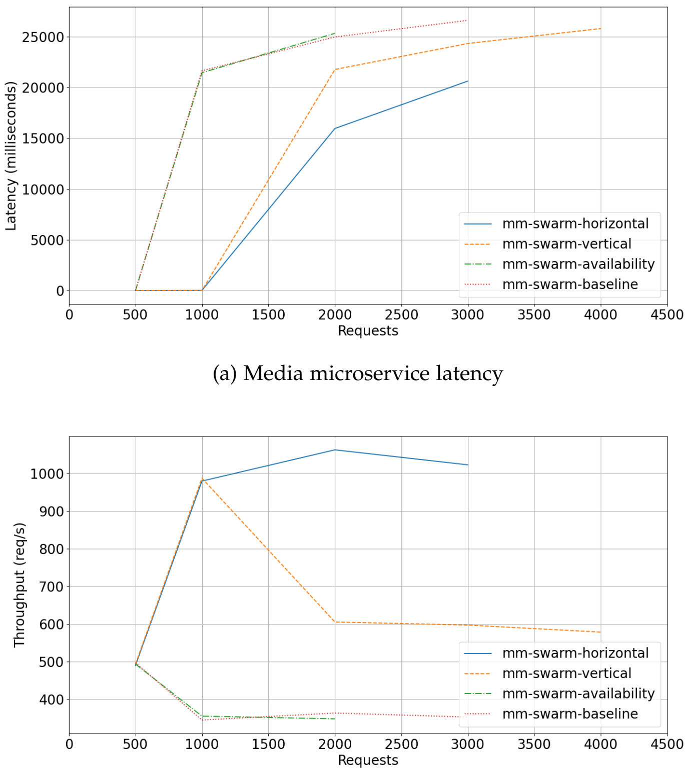
(a) Social network latency

(b) Social network throughput

Figure 6.1: Social Network tail latency P99 and throughput per second compared in the scaling scenarios using Docker Swarm tor compared to requests per second.

The throughput is shown by Figure 6.1b. Here, the baseline shows the break around 2.000 req/s, also results in a measured throughput of 2.000 req/s and decreases as the requests from the test client increases. *VS* and *HS* can handle up to 50% more requests as they break around 3.000 req/s compared to the baseline of 2.000 req/s, while *HA-off* reduces performance between at least $12,5\% - 25\%$. Increasing the load further causes the performance to change. This trend is similar to *HS*, *VS* and *HA-off*. When the load is too high, the performance decreases more and more as the load increases. One thing we can see is that *HA-off* does not immediately decrease, but has a slight increase before also decreasing for increasing intervals.

Media Microservice: Second, we look at the performance of the media microservice. In Figure 6.2a, tail latency P99 starts at a load of 500 req/s (100%). *VS* and *HS*, increases tail performance with 100% up to 1000 req/s. The effect of *HA-off* is not noticeable and shows no difference from the baseline.



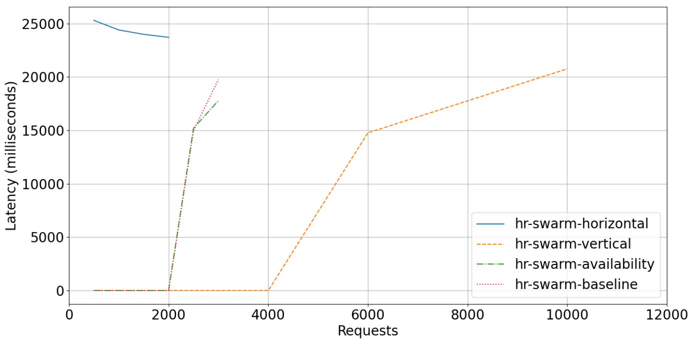
(a) Media microservice latency

(b) Media microservice throughput

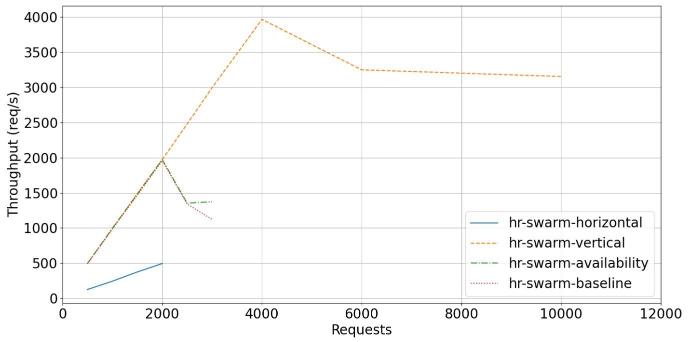
Figure 6.2: Media Microservice latency and throughput compared in the various scaling scenarios with Docker Swarm.

In Figure 6.2b, throughput shows a similar trend to latency for the baseline and *HA-off*. *HS* and *VS* similarly show a 100% performance increase before showing increases in tail latency. Interestingly, increasing the load beyond the 1.000 req/s breaking point causes the *HS* to plateau while *VS* performance decreases and plateaus to around 600 req/s.

Hotel Reservation: Now we look at the performance of the hotel reservation application. Figure 6.3a, represents the tail latency. The baseline starts to break after a load of 2.000 req/s (100%). *HA-off* is similar with no noticeable difference. *VS* increases the performance by almost 100%. For the *HS*, the throughput stops after the initial 2.000 requests per second which indicates that too many errors have been returned. HS has reached a breaking limit with limited resources. Furthermore, Figure 6.3b shows that the throughput shows similar trends to tail latency when comparing the baseline to *VS* and *HA-off*.



(a) Hotel Reservation latency



(b) Hotel Reservation throughput

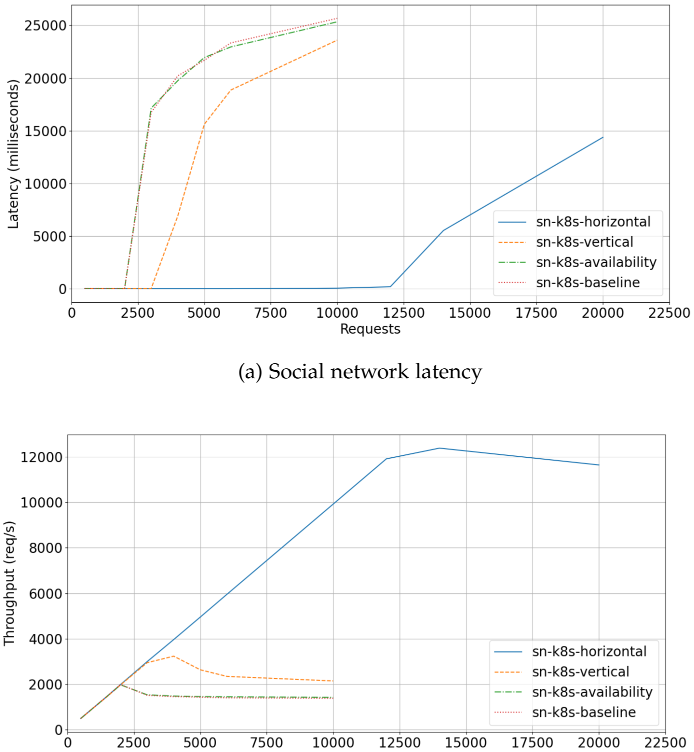
Figure 6.3: Hotel Reservation latency and throughput compared in the various scaling scenarios with Docker Swarm.

Conclusion A: Based on the applications on Docker Swarm, comparing the baseline performance to vertical or horizontal scaling (except for hotel reservation) can increase the performance with 100%. Furthermore, having no high availability does not decrease performance, but in the case of the social network, it decreased performance to 25%. Finally, when the load is too high, the latency changes from milliseconds to seconds, clearly indicating that the application has hit the breaking point and performance decreases.

6.4 *Experiment B.* Comparing Kubernetes Performance

In Experiment B, we will look at the performance of Kubernetes. A complete overview of the results is shown in Figure B.4. Our Figure 6.4a, Figure 6.5a and Figure 6.6a shows tail latency P99 results for the different applications. Our Figure 6.4b, Figure 6.5b and Figure 6.6b show the difference in throughput for the applications deployed with Kubernetes.

Social Network: For the social networking performance we can see in Figure 6.4a that our baseline can handle up to 2.000 req/s (100%), before breaking. *HA-off*, shows no difference compared to the baseline. Taking a look at *VS* shows an increase of the performance by 50%, to 3.000 req/s before reaching a breaking point. Note that, in contrast to Docker Swarm, *HS*, has a performance increase much greater than would be expected by 2x the number of replicas. At this point, we do not have an explanation for this behaviour, and this would require further investigation. In Figure 6.4b the throughput shows a trend comparable to latency.



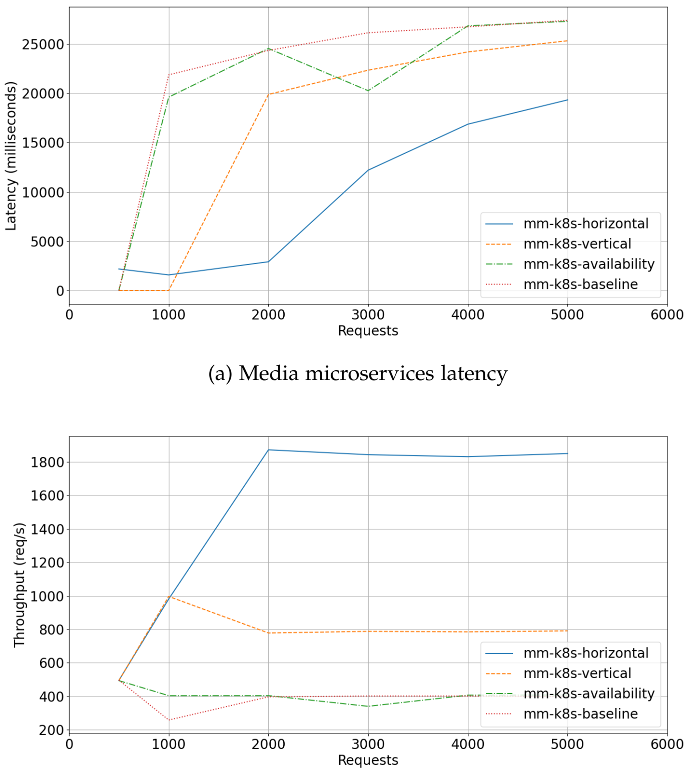
(a) Social network latency

(b) Social network throughput

Figure 6.4: Social Network deployed with Kubernetes. Comparing latency and throughput in the various scaling scenarios.

Media Microservice: Again, we look at the media microservice performance. When evaluating tail latency in Figure 6.5a, the baseline breaks after the initial load of 500 req/s (100%). Again *HA-off* is similar to the baseline. *VS* increases the performance by 100%, to around 1.000 req/s. *HS* again shows unexpected behavior, out-of-the box the tail latency is in the seconds, which indicates a performance decrease compared to the baseline.

When we look at throughput, in Figure 6.5b, we can see that the throughput through *HS* is able to handle up to 2.000 req/s with 1.900 req/s measured, and 280% performance increases more than one would expect from replicas



(a) Media microservices latency

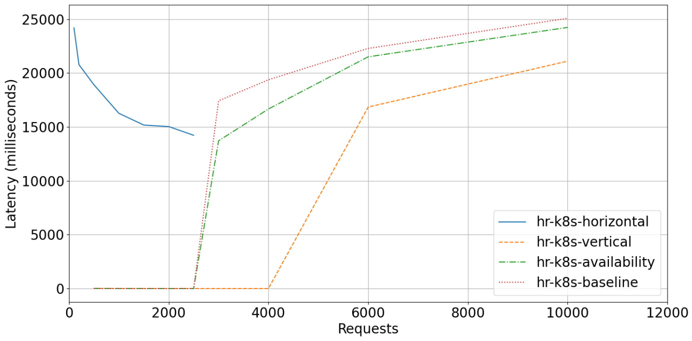
(b) Media microservices throughput

Figure 6.5: Media microservices deployed with Kubernetes. Comparing latency and throughput in the various scaling scenarios.

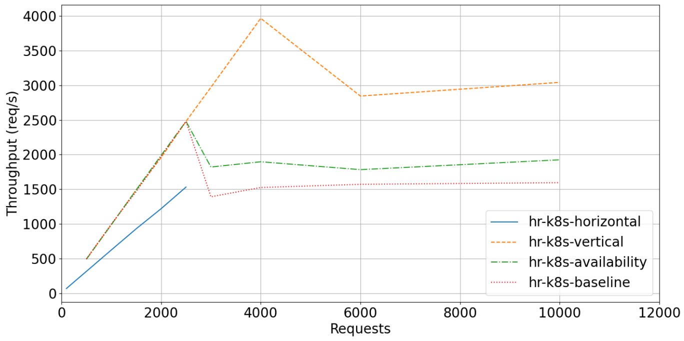
2x. *VS* reaches a limit of around 1.000 req/s a 100% performance increase. We verified these results by taking a closer look at the data with Jaeger, and from the GUI we can see the increased latency in relation to the type of workload per microservice. Specifically, certain requests, using workloads such as composing reviews, take longer to complete than less demanding requests. In short, this suggests that only partial service performance degradation is the reason for our overall higher tail latency P99 for *HS*.

Hotel Reservation: Finally, our performance of the hotel reservation applications. In Figure 6.6a, the baseline and *HA-off* break around a load of 2.500 req/s (100%). *VS* gives us a 60% performance increase and can handle the load up to around 4.000 req/s. *HS* again has a noticeable high latency, this performance can be due to the way how our horizontal scaling is approached within the benchmark. It might be that replicas 2x does not scale and actually breaks the application sooner.

For throughput, we look at Figure 6.6b. The baseline, *HA-off* and *VS* show the expected pattern. Based on the *HS* we can see that performance compared to baseline decreases by at least 60%, when comparing the break point of 2.500 to the 1.500 req/s. One reason for this *HS* being broken here is that the application might simply be more fragile in the horizontally scalable with limited resources.



(a) Hotel reservation latency



(b) Hotel reservation throughput

Figure 6.6: Hotel reservation deployed with Kubernetes. Comparing latency and throughput in the various scaling scenarios.

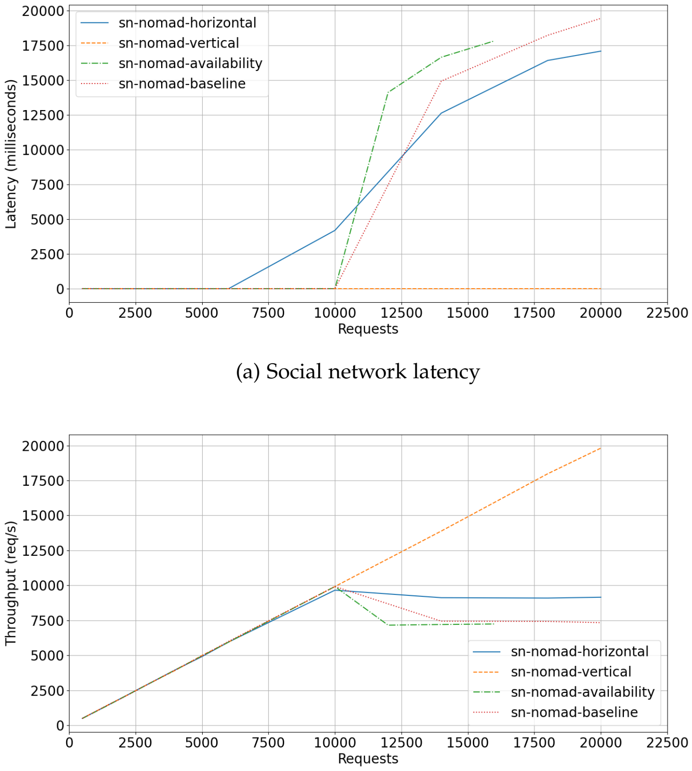
Conclusion B: Based on the applications on Kubernetes, our observation shows that the baseline performance is the same as the *HA-off* performance. Furthermore, *VS* also improves performance around 50% and *HS* can at best improve performance up to 100%. However, we would consider further investigation into *HS* for the hotel reservation application.

6.5 *Experiment C.* Comparing Nomad Performance

In Experiment C, our final experiment, we measure the performance of Nomad. All results are produced are shown in the Appendix, with Figure B.6 and Figure B.7. The figures in this section are about the latency and throughput of our applications for the scaling scenarios. In Figure B.6 we can see that we have found the breaking point of almost all parameters.

In this section we discuss Figure 6.7a, Figure 6.8a and Figure 6.9a which show a slice of figures with tail latency P99. Our Figure 6.7b, Figure 6.8b and Figure 6.9b show the difference in throughput for our Nomad-implemented applications.

Social Network: Our investigation this time focuses on Nomad. In contrast to previous experiments, the baseline performance seems to start noticeably higher. In Figure 6.7a the baseline can handle up to 10.000 req/s (100%) before having a noticeable increase of latency. *HA-Off* and *VS* both show a similar breaking point. Surprisingly, *HS* breaks at 6.000 req/s. This would imply that *HS* has a performance decrease of 40% and *VS* does not improve performance based on tail latency.



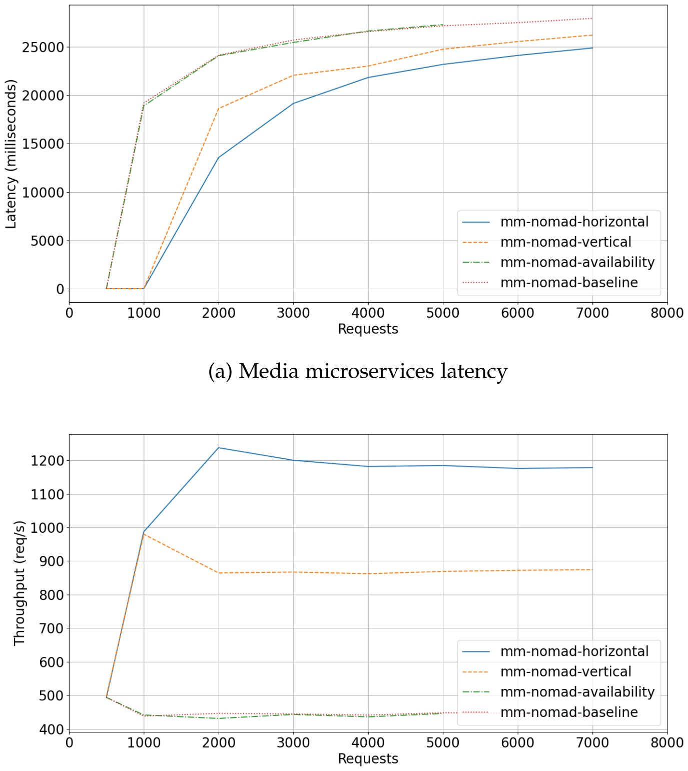
(a) Social network latency

(b) Social network throughput

Figure 6.7: Social network deployed with Nomad. Comparing latency and throughput in the various scaling scenarios.

When we again look at Figure 6.7b to evaluate throughput, we can find that various scaling scenarios break around 10.000 req/s. Except for *VS*, which perfectly handles the load up to 20.000 req/s. We would not expect this, especially as the resources are limited and also as our preliminary experiments indicated that unlimited resources in Docker Swarm reach their breaking point before this range. *HS* however is bottlenecked at the same performance as the baseline, meaning that having two containers does not help in this case. This could mean that in the case of Nomad, the performance with the limited resources is very efficient or that this simply might be an outlier run.

Media microservices: Now let us look at the media microservice performance. In Figure 6.8a the baseline breaks after the initial load of 500 req/s (100%). The *HA-off* shows no difference to the baseline. The vertical and horizontal scaling improves the system 100% to around 1.000 req/s throughput before having a noticeable increase in latency.



(a) Media microservices latency

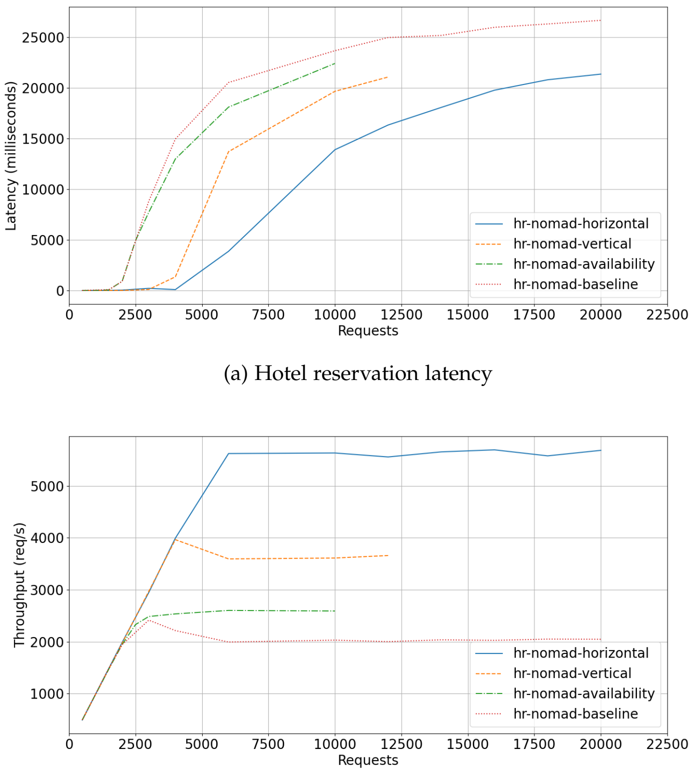
(b) Media microservices throughput

Figure 6.8: Media microservices deployed with Nomad. Comparing latency and throughput in the various scaling scenarios.

When comparing throughput in Figure 6.8b, the initial baseline and *HA-off* throughput drops after the initial 500 req/s and remain identical. This time *VS* and *HS* shows a slightly different story. The results show that *VS* increases the initial baseline throughput almost up to 1.000 req/s, implying an increase of 100%, before dropping in performance to around 900 req/s. *HS* actually is able to go past 1.200 req/s and stays around that number, implying an increase of 120% performance. Showing that *HS* in this case is preferable.

Hotel reservation: Now we can take a look at the performance of the hotel reservation application. In Figure 6.9a the baseline and *HA-off*, again following a similar trend with tail latency, break after a load of 2.000 req/s. The *VS* and *HS*, both handle the load up to 3.000 req/s, implying at least an increase of performance of 50%. The *HS* outperforms *VS* until 4.000 req/s, a 100% increase in performance, after which the performance starts to decline.

From the perspective of throughput, we can observe the results in Figure 6.9b. The baseline falls off around 2.500 req/s, while *HA-off* follows a similar trend. It's clear that *HS* is able to handle up to 6.000 req/s , while in comparison *VS* falls off after 5.000 req/s. This tells us that both *VS* and *HS*, but *HS* goes beyond a performance increase of



(a) Hotel reservation latency

(b) Hotel reservation throughput

Figure 6.9: Hotel reservation deployed with Nomad. Comparing latency and throughput in the various scaling scenarios.

100% in this case. This is interesting, given that our previous orchestrators and hotel reservation observation point to the contrast.

Conclusion C: Based on the applications with Nomad, our applications tell us different conclusions from the previous orchestrators with regards to scaling. Based on Social Network, *VS* is the preferable method, with an increase of 100% at best. However, the other two applications tell us that *HS* is the preferable method where it can actually increase performance beyond 100% compared to the baseline. The *HA-off* in general did not show any difference to the baseline in all cases.

6.6 *Experiment D* Comparing Swarm, Kubernetes and Nomad

In this final section, we will create an overview of the breaking points thus far, comparing tail latency of each orchestrator for the various scaling scenarios. From this overview, we can determine the difference in performance between orchestrator and scaling method. The table is shown in Table 6.2. The overview in the table is based on the figures shown in the previous section and on the complete overview in the appendix, Figure B.1 and Figure B.2.

Table 6.2: Overview of the breaking points based on loads of requests per second. The breaking point implies that the tail latency 99 percentile has a noticeable increase, based on requests per seconds. Bold shows the best performing orchestrator(s) per benchmarked application, if applicable.

	Social Networks				Media Microservices				Hotel Reservation			
	Baseline	Availability Off	Horizontal	Vertical	Baseline	Availability Off	Horizontal	Vertical	Baseline	Availability Off	Horizontal	Vertical
Swarm	2.000 - 3.000	1.500 - 2.000	3.000 - 4.000	3.000 - 4.000	500 - 1.000	500 - 1.000	**1.000 - 2.000**	1.000 - 2.000	2.000 - 2.500	2.000 - 2.500	?? - 500	**4.000 - 6.000**
Kubernetes	2.000 - 3.000	2.000 - 3.000	**12.000 - 14.000**	3.000 - 4.000	500 - 1.000	500 - 1.000	?? - 500	1.000 - 2.000	**2.500 - 3.000**	**2.500 - 3.000**	?? - 500	**4.000 - 6.000**
Nomad	**10.000 - 14.000**	**10.000 - 14.000**	6.000 - 10.000	**20.000 - ??**	500 - 1.000	500 - 1.000	**1.000 - 2.000**	1.000 - 2.000	2.000 - 3.000	2.000 - 3.000	**4.000 - 6.000**	3.000 - 4.000

The range of breaking points of our orchestrators is shown in Table 6.2. Based on this table, we can see that most of the performance is similar, especially when looking at Swarm and Kubernetes. We can see that there are some outliers, e.g. performance difference has drastically increased such that 20.000 req/s for vertical scaling for Nomad does not show a breaking point, that might indicate unknown factors that affected the performance of our benchmarks on orchestrator. Our results have been checked to see if container resources were not correctly limited, but our testbed monitoring tools and logs indicate that it was correctly limited, so further investigation should be done in future work on the application or orchestrator to explain the performance difference. Still we believe some generalisations can be made on the basis of this table. We will summarise the most noteworthy findings per application with a focus on comparing the orchestrators and the best performing scaling technique.

Social Network: Our overview of the breaking points shows that the Nomad orchestrator outperforms both Kubernetes and Docker Swarm. The testbed results are unexpected due to the drastic performance differences, further investigation showed that our configurations are correct so we cannot attribute it to a testbed bug. Ignoring Nomad, we can see that Kubernetes is able to perform better, with regards to horizontal scaling. Again, this is unexpected. However, the remaining results shows that horizontal and vertical scaling can improve the performance to up to 200%.

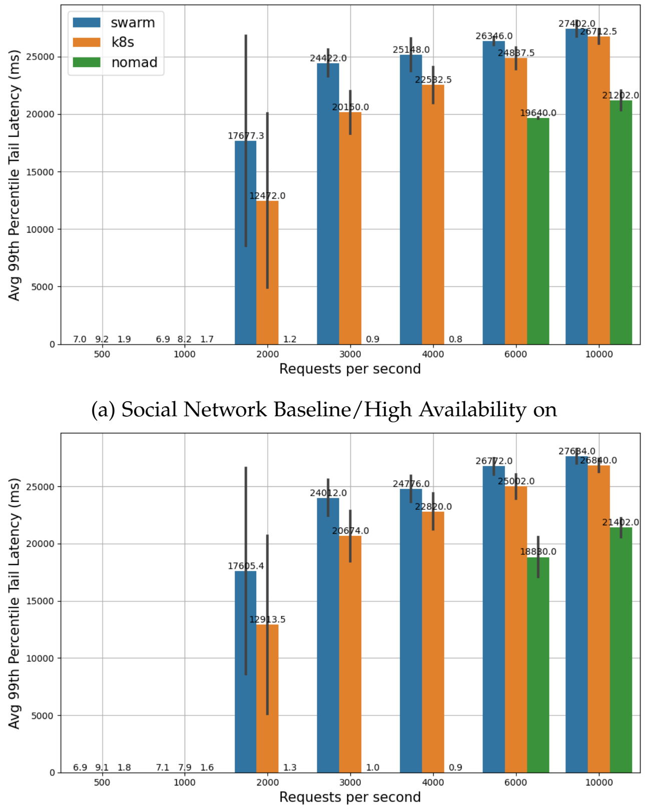
Media Microservice: The results shows that on all scaling scenarios, media microservices perform the same for each orchestrator. Furthermore, the vertical scaling indicates that increase of performance also translates to at maximum a 100% performance increase for both horizontal and vertical scaling. Except for Kubernetes, that has a decrease in performance when using horizontal scaling. This might indicate that some factor outside our testbed is affecting the results, such as a configuration bug in the orchestrator, which would need to be further investigated.

Hotel Reservations: Our final benchmark application, hotel reservation, is an interesting case. There are multiple conclusions that can be drawn from here. First, the horizontal scaling seems to either not work well for Kubernetes and Swarm, which could be due to the nature that application is setup or that these runs are an outlier. However, Nomad seems to contradict this, where it simply works, which could suggest that it could be due to an outlier run. The remaining table tells us that, compared to the baseline, vertical scaling is able to achieve a performance increase of $100\% - 200\%$ at best.

Conclusion D: In this section, we compared the multiple orchestrators and latency breaking limits. Our approach shows that there could be some factors that influence performance both at the orchestrator level and at the microservice architecture level. In the case of Nomad, which is the only orchestrator where horizontal scaling works for the Hotel Reservation application. Furthermore, comparing baseline performance with horizontal and vertical scaling often leads to a performance increase of $100\% - 200\%$, which is shown in the Social Network and Media Microservice. However, we cannot confidently say that some other factors are in play due to unexpected performance results, which could explain the performance of Nomad in the case of Social Network performance having an outlier performance compared to Swarm and Kubernetes on all levels.

6.7 *Experiment E.* Validating Tail Latency Comparison with Social Network and Multiple Runs for each Orchestrator

We feel that we require results with more repetitions to have more confidence in our previous conclusions, we will go in depth on the previous results of breaking points shown in Table 6.2 on the left side for social network application. Therefore, in this final experiment, we compare the social network benchmark where each orchestrator is compared in the four scaling scenarios (baseline/high availability on, high availability off, vertical scaling and horizontal scaling) but with more repetitions. The containers have limited resources, and the latencies are recorded for an increasing load starting at 500 req/s up to 10.000 req/s for a duration of 30 seconds per run. Each run has 5 iterations. In total, 105 runs have been performed.



(a) Social Network Baseline/High Availability on

(b) Social Network High-Availability Off

Figure 6.10: Social Network tail latency performance P99 based on 5 iterations. Comparing the orchestrators baseline and high availability off. Workloads are in requests per second run for a duration of 30 seconds.

Docker Swarm: Figure 6.10 and Figure 6.11 shows the average tail latency P99 for the baseline for Docker Swarm in blue or the left bar. Figure 6.10a shows the baseline, with Figure 6.10b showing high availability turned off (HA-off), vertical scaling (VS) is shown in Figure 6.11a and horizontal scaling (HS) is shown in Figure 6.11b for Docker Swarm. The baseline shows that Docker Swarm can handle up to 2.000 req/s before breaking. We can see that that error bar shows a significant variance increase when application reaches breaking point. As tail latency can shoot up to 25.000 ms, meaning the requests most likely did not finish or taking signficantly longer. HA-off is similar to baseline, which is expected based on our previous experiments. The error bar shows a similar large variance around the 2.000 req/s. VS actually increases the performance, where it can handle up to 3.000 req/s, compared to the baseline a 50% increase to performance. HS also breaks down at 3.000 req/s, which also means a 50% performance increase.

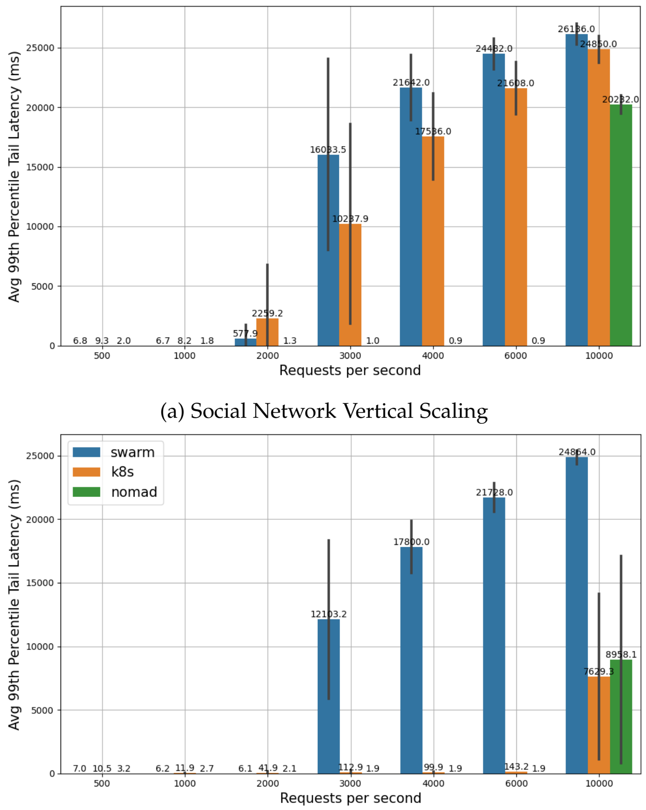
Our previous Table 6.2, representing our previous experiment results, indicate some similarities and differences for the results. The baseline still holds a similar breaking point up to 2.000 req/s, the new breaking point of HA-off is 2.000 req/s while our previous results said between 1.500 and 2.000 req/s. For VS, our current experiment shows that 2.000 req/s is where we see a noticeable performance decrease instead of the 3.000 req/s. However, 3.000 req/s is where the error bar is the largest, indicating that the breaking point is somewhere between 2.000 and 3.000 req/s, close to our previous results. For HS, we can see that 3.000 req/s is the clear breaking point for our current experiment, similar to our previous results shown. In short, we see that the results in the case of Docker Swarm are consistent and reproducible.

Kubernetes: Represented by the colour orange or the middle bar in Figure 6.10 and Figure 6.11. The baseline breaks down at 2.000 req/s, similar to Docker Swarm. HA-off shows no performance changes as it also breaks at 2.000 req/s. VS breaks around 2.000 req/s as the tail latency increases to an average of 2.300 ms, so no performance gains. HS in contrast is able to handle up to 10.000 req/s before breaking. Indicating a performance increase of 5 times the baseline, 400%. This seems to be supported by the results as shown by our previous experiment B (Kubernetes Performance). So the performance increase is more than one would expect from setting the replicas 2x.

Our previous Table 6.2, shows that the results for Kubernetes are similar. The baseline and HA-off breaks around 2.000 req/s, both in our previous and current experiment. For VS, our previously estimated breaking load was 3.000 req/s, and our new results show that 2.000 req/s has a noticeable performance decrease. Similarly to Docker Swarm for VS, we can see that the error bar is the largest around 3.000 req/s, close to our previous estimated breaking load. For HS, our previous results indicated an outlier of 12.000 req/s before breaking, however, this time we can see that 10.000 req/s is our new breaking limit, which is a large error bar. In short, we can see that the results of Kubernetes are similar enough to our previous results, at least with regard to the trend shown for performance, and that our previous runs had some outliers.

Nomad: As can be seen in green or the right bar in Figure 6.10 and Figure 6.11. The baseline can handle up to 6.000 req/s, the highest of all orchestrators. The HA-off has similar performance and shows no performance changes. VS breaks at 10.000 req/s, indicating a 67% performance increase. HS Actually also breaks at 10.000 req/s, which means a similar performance increase of 67%.

In the case of Table 6.2, which indicated that Nomad was the best performing orchestrator, we can see that the breaking limits are different. The baseline and HA-off in our current experiment shows a breaking limit around 6.000 req/s, in contrast to our previous 10.000 req/s breaking limit. For VS, our new breaking limit is 10.000 req/s, in comparison to 20.000 req/s previously. HS gives us the breaking limit of 10.000 req/s, with a large error bar. Previously, the breaking limit was around 6.000 req/s. In the case of Nomad, our results seem to indicate completely different breaking point, however, the overall trend of best performing orchestrator based on our results remains Nomad.



(a) Social Network Vertical Scaling

(b) Social Network Horizontal Scaling

Figure 6.11: Social Network tail latency performance P99 based on 5 iterations. Comparing the orchestrators vertical scaling and horizontal scaling. Workloads are in requests per second run for a duration of 30 seconds.

Conclusion E: In this section, we compared the multiple orchestrators and latency breaking limits for the social network repeating each increment of load 5 times per orchestrator. Reflecting on our previous experiments, we expect similar results. In this experiment, a clear indication of breaking is the variance shown by the error bar. Compared to the baseline, turning off high availability does not significantly change the performance of tail latency. VS can increase performance from 50% to 67%. HS interestingly enough also increases performance 50% to 67% but in the case of Kubernetes it increases up to 400%. In comparison to our previous results in Table 6.2 on breaking points, we can see that Docker Swarm and Kubernetes showed similar performance, while Nomad has the most significant performance differences. Overall, the trend is that performance increases in the case of VS and HS, with HS seeming to be preferable in most cases as higher performance gains can be made. The baseline and HA-off did not show significant differences. Our orchestrators results follow the same trend as our previous experiments, even in the case of Nomad which showed more extreme performance changes. In conclusion, based on our testbed results, Nomad is still outperforming our other orchestrators, in line with our previous experiments. Horizontal scaling has the highest performance gain (potential). Kubernetes would outperform the Docker Swarm.

Chapter 7

Discussion

In this chapter, the results of the experiments presented in Chapter 5 and Chapter 6 are discussed. First, using these results, the research questions are addressed. Secondly, we will look at our research implications in our field of study. Finally, we will end with a brief discussion of the limitations of our research and testbed.

7.1 Answering the research questions

RQ T.1: How to design and implement the DeathStarBench into a testbed?

Our approach to building a testbed includes building DeathStarBench as part of the whole testbed. Our testbed supports researchers to perform rigorous testing. The design of our testbed, described in Section 4.1, is divided into five layers: Infrastructure, Orchestration, Monitoring, Microservice Application/DeathStarBench, and Experimentation. Our minimal viable open-source testbed is extendable by design with other microservice benchmarks, for which we started by building DeathStarBench on top of our testbed.

RQ T.2: How do we implement consistency/reproducibility of our testbed? What is required to set up our testbed/experiments?

To work toward a reproducible testbed, as described in Section 4.2, we include scripts to deploy and set up a consistent test environment with multiple container orchestrator engines. On the infrastructure level, we use CloudLab which provides the platform for reusable experimentation with transparent resources available for research, such as profiles to create identical node clusters. The testbed is open source, and this thesis includes an explanation on recreating our testbed.

RQ T.3: Using our testbed, how can we ensure consistency in our experiments? What pre-tuning is required to conduct experiments, and what performance characteristics can we find?

To explore consistency and reproducibility, we compared benchmark applications in multiple preliminary experiments using only Docker Swarm, which we discussed in Section 5.1. We manage parameters such as the number of connections, threads, and duration of all experiments using the load generator. Additionally, our testbed includes parameters like high availability, vertical and horizontal scaling, limited or unlimited resources, and orchestrator settings. We controlled our test environment for software versions and ran identical hardware. However, our preliminary results, with the metric tail latency and throughput, cannot guarantee a consistent and reproducible result for the benchmarks if the experiments have not been repeated enough. Experiments such as time between workloads and re-deploying the application change the performance based on our results. For that reason, we conducted additional experiments to explore more in-depth performance characteristics of the microservice with multiple runs, sometimes three or up to five runs per load. In some experiments, our measured latency would start high and decrease with increasing load, indicating a cold start phenomenon. In our later experiments, in Chapter 6, we again seem to have some cold start issues w.r.t. hotel reservation. In summary, running the same experiments with the same parameters would reveal different results but show similar trends in performance.

RQ T.4: Using our testbed, are there performance differences in the orchestration tools in various scaling scenarios?

One of the main contributions of our testbed is the comparison of the benchmark applications deployed on Kubernetes, Docker Swarm, and Nomad, as described in Chapter 6. Given the sheer amount of results, it is hard to make some generalisations which depend on both the orchestrator and/or scaling scenario. We found that there are some performance differences between orchestrators and parameters: high availability does not implicate a performance change compared to the baseline. Increasing resources, by horizontal or vertical scaling, usually doubles the performance, but could also increase the performance by more than 100%. However, the type of application and orchestrator affects performance. We would have to research this increase, but that would be a future work. In short, Kubernetes and Docker Swarm seem to perform similarly for applications. Nomad showed better performance overall, but despite repeated results, we find it generalise this result and conclude the Nomad orchestration software has some innate ability to improve the performance of the microservice architecture. For now, more research would be needed to understand why.

7.2 Implications of the research

Here we address some of the open questions that we think are relevant to this thesis and our field of study.

What is the relevancy and importance of the experiments? As there is a lack of reproducible and open-source experiments with realistic benchmarks in the research field, we are happy to contribute with our empirical data. Before the start of this study, the field of microservice architecture was understudied concerning empirical evidence to show the limitations of the microservice architecture, which is relevant for future study. As microservice architecture is still relevant both in business and academia, the importance of empirical studies cannot be understated and should remain verifiable and transparant.

What will these results bring new to the field/community? The contribution is primarily twofold. First, we know that the baseline we created in this research can be used for future comparison, which is in the spirit of our goal of reproducibility, especially if we focus on exploring the social network benchmark. The results allow future work to be done both in depth and in breadth for empirical performance evidence. Secondly, with our testbed, we have worked towards a tool for the research community to reproduce our results. Our scripts can be used to run similar research with CloudLab or other clouds. As the several scripts are written in Bash, they can also be run on your own hardware, albeit with a bit of a rewrite. Our research will remain available on GitHub, reproducible in the case of future research, and can be run on different public or private (cloud) hardware for comparative research.

Could we have generated the results without our testbed? We believe that this research could be produced without our testbed. We must say that the cognitive load to produce similar research might prevent an early researcher from avoiding this topic, e.g. learning curve of multiple orchestration engines and microservice architecture design and debugging, etc. Furthermore, the Docker Swarm and Nomad deployment files had not yet been created in the open-source edition for all instances of the three applications of the DeathStarBench. Kubernetes/Openshift was only available for the Social Network and Media microservice application in the original research, but the Hotel reservation application had not been fully implemented for the orchestrators. If one would scope their research to the Social Network the results of studying the performance characteristics for the microservice architecture would be the most achievable when starting fresh. The other applications would require debugging and rewriting extensively depending on your hardware/cloud platform. The experimental setup would also have to be created, which you have to re-tune according to your setup. In other cases where you would not run some simple tests and want to compare using different orchestration engines, our testbed has at least the advantage of reducing your research time. Our tools have been experimented with on each orchestration engine to find the limits within each orchestration engine, and also include these as explicit resource definitions in the deployment scripts and files. This work is not trivial to your research time, saving your time which you can use to focus only on changing the testbed, benchmarks or experiments you require for your specific research goal.

More takeaways. The microservice architecture offers a high degree of flexibility but can also introduce an element of unpredictability in terms of system performance. To address this challenge, the following recommendations are proposed. (1) Use a diverse set of metrics to measure the performance of the system, as relying on a single or limited number of metrics may not accurately reflect the bottlenecks in the system. (2) Identify the weakest components in the architecture, as a few poorly configured services can significantly impact the overall performance of the system. (3) Consider scaling resources in both horizontal and vertical directions, as different applications and architectures may benefit from different approaches to scaling. (4) Keep in mind that while containers may have access to increased resources, certain services may need to be specifically configured to take advantage of these resources.

Limitations Experiments. The experiments, especially in the second part about orchestration, focused on a broader scope of the experiments instead of repetition. This broader scope was a trade-off as we put extensive effort in debugging and tuning the orchestrators with limited resource and automating this process, which was already quite challenging due to a combination of the microservice applications being experimental and require configuration changes. Furthermore, the orchestrators also posed some challenges, especially Nomad, as we had to manually verify the limits imposed of the orchestrators and at the same time ensure that this was due a limitation of bug in our testbed. Nomad seemed to require more extensive debugging due to a lack of documentation. If not done correctly, this could have not restricted container resources and invalidating some of our research, and that step took a while to verify. Based on the major performance differences to the other orchestrators, Nomad performance seems deployed as if it were run with no resource limits, which had us go back and forth a few times to check if the resources had been limited. At a certain point, we had to assume the orchestrator was correctly tuned as our checks continued to verify the limits. In future work, we would rerun the experiments to increase the repetitions and put major guard rails in place to ensure that the resources are limited for the applications to increase the confidence in the results of Nomad, such as limiting the resources on node level.

7.3 Limitations and Improvements for the Testbed

Before we developed this testbed we had some initial goals and ambitions. Many of these goals are simply not feasible due to time constraints or are not within the scope of our research goals. As such, this section will discuss the relevant (potential) limitations of the testbed and the improvements that could be made.

Our testbed currently supports running experiments on the DeathStarBench suite using a cluster of nodes as we have not tried deploying it to a larger cluster of nodes. At most nine nodes were used to operate the control plane, servers for the applications, and a test client to conduct the experiments. It has been tested only on the CloudLab environment, with Ubuntu 20.04. The tools are only in an early version, as support for multiple cloud/infrastructure platforms has not yet been incorporated and requires manual intervention. Further testing is required to verify our results, as we did not run our experiments in multiple runs or a sufficient amount of runs, which we would like to see in tenfolds. This implies that our results need to be made more statistically significant.

What the Testbed Does Not Do:

1. Does Not Simulate Real-World Scale Fully
2. Does Not Include Diverse Infrastructure
3. Does Not automate everything, several scripts need to be adjusted beforehand

7.3.1 Strengths and trade-off

1. **Strength** Provides comparative performance analysis of different orchestration engines, under similar controlled conditions, which can provide insights into their relative performance.

   Trade-off The controlled environment we have created, does not fully replicate the variability of real-world cloud deployments. It can create a baseline to build future research on in a controlled environment.

2. **Strength** Focus on detailed metric collection for 99th percentile tail latency and throughput.

   Trade-off There are more metrics in providing a holistic evaluation of the performance. Resource utilization can be an example.

3. **Strength** Enables repeatable experiments. The scripts all have been written to enable a wide range of experimentation. Either in parameter tuning or through repeatable experimentation. These experiments aid in a reliable comparison and benchmarking.

   Trade-off The repeatability and control of the environment have yet to be verified, but we made a start with the Social Network for each orchestrator. This future work is needed to ensure that we can be extrapolated to other environments. Our experiments should be run on different hardware to ensure the repeatability and control of our empirical results.

7.3.2 Caveats and Weak Points (Limitations) with Solutions

1. **Workload variability**: Currently, our experiments are run with a static workload. This may not be representative of real workloads, which can vary greatly.

   Solution: We can think of two solutions, either extending wrk2 to support dynamic workloads through existing extensions found online or to find a suitable replacement workload generator tool that can perform dynamic workload and re-implement the workload of each service per application.

2. **Orchestration/Monitoring Overhead**: We currently do not consider the orchestrator or monitoring overhead, which is running next to the applications. Ensuring that those are not excluded in some way when the applications are benchmarked ensures the results are not affected by monitoring tools, if even possible.

   Solution: Our proposed solution would be to extend our current scripts to ensure the monitoring services have been disabled before the experiments are run to ensure the state of the applications.

7.3.3 Improvements

1. **Reproducibility:** Orchestrator performance difference with our testbed has been shown in multiple experiments however we limited runs. The current experiment(s) showed that reproducibility for the Social Network applications seems possible. However, to ensure our conclusions are generalisable more future work has to be performed that shows the significance for both the benchmarks and for our other experiments, either on similar or different hardware.

2. **User-friendliness:** While our prototype has some instructions. We think that documentation and more examples of how to run our tools would help the community adapt to using this tool.

3. **Expand Benchmarks:** We would like to extend the testbed with a wider variety of existing open-source benchmarks that have already stood the test of time, such as TeaStore.

4. **Higher level descriptors:** Currently, there are many open source options to replace specific parts in our testbed implementation code. Tools such as Terraform/OpenTofu might help with creating a hardware/infrastructure agnostic testbed tool. Ansible/Chef can replace the configuration step of the nodes.

5. **Support major cloud providers, deployment files**. Existing results on cloud providers such as Azure, AWS and even Google Cloud are scarcely and mostly unverifiable available through commercial parties on blogs. If

the applications could be extended to be cloud-native, it could be a (relatively) small step. Then we think the academic value of our testbed for comparative studies will increase significantly. However, it would require implementation choices such as storage, network, and computing resources to choose from.

Chapter 8

Conclusion and Future Work

This chapter presents the conclusion of the thesis, which includes a summary of the research findings regarding the main research questions and the implications of these findings. Furthermore, the conclusion suggests potential avenues for future work based on the results of the testbed as a tool.

8.1 Conclusion

In conclusion, the use of microservice architecture in complex systems has been shown to have several potential benefits, including increased scalability, flexibility, and maintainability. However, it is important to carefully consider the trade-offs and potential challenges associated with this approach, such as the need for more sophisticated orchestration and the risk of increased communication overhead. Our approach introduces an open-source testbed that research can build on, including a baseline dataset for comparative research. Furthermore, we used our testbed to perform rigorous experimentation to compare multiple orchestrators. Based on our results, we can take some key lessons from this. Overall, the performance of microservice architecture in complex systems appears to be highly dependent on the specific applications, orchestrator, and scaling scenario, and further research is needed to better understand how to implement and optimise this approach effectively in practice. Compared, the difference in the performance of the same application between scaling scenarios could be 50% or more, where horizontal scaling is preferred, as it shows the most overall potential performance gains. Depending on the different orchestrators, performance differences have been noted, and purely on performance Nomad would be the best performing orchestrator, then comes Kubernetes and Docker Swarm ranks last.

8.2 Future Work

There are several promising directions for future research on the performance of microservice architecture in complex systems using our testbed. The first and most important steps are to further ensure a generic, robust, and reproducible testbed. Toward this goal, there are multiple approaches possible. One approach would require a more in-depth exploration, building on top of our existing research to ensure verifiability. This could be done using similar hardware to rerun the experiments in CloudLab. Additionally, we can improve the reproducibility of our research by extending the testbed to include a statistics module that reports the statistics around an experiment or extending the existing Jupyter Notebook scripts to do this. Furthermore, we can envision an extension of our current testbed setup with a more streamlined command-line interface to help better adapt the tools to the community. Finally, it is also possible to continue research on performance differences by extending the range of (open-source) container orchestration engines or to compare public cloud providers, either through a singular focus on Kubernetes or by deploying the testbed on a virtualized environment using multiple virtual machines. Overall, there is still much to learn about the performance of microservice architecture in complex systems and their relation to container orchestrators, and continued research in this area will be critical to improving the design and deployment of large-scale distributed systems and enabling researchers and organisations to make informed decisions about tool selection and deployment.

Bibliography

[1] *Acme Air Sample and Benchmark*. Acme Air, Jan. 28, 2021. URL: https://github.com/acmeair/acmeair (visited on 02/02/2021).

[2] Carlos M. Aderaldo et al. "Benchmark Requirements for Microservices Architecture Research". In: *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). Buenos Aires, Argentina: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), May 2017, pp. 8–13. ISBN: 978-1-5386-0417-5. DOI: 10.1109/ECASE.2017.4.

[3] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture". In: (May 2016), p. 13.

[4] David Balla, Csaba Simon, and Markosz Maliosz. "Adaptive Scaling of Kubernetes Pods". In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium. Apr. 2020, pp. 1–5. DOI: 10.1109/NOMS47738.2020.9110428.

[5] Luciano Baresi et al. "KOSMOS: Vertical and Horizontal Resource Autoscaling for Kubernetes". In: *Service-Oriented Computing*. Ed. by Hakim Hacid et al. Cham: Springer International Publishing, 2021, pp. 821–829. ISBN: 978-3-030-91431-8. DOI: 10.1007/978-3-030-91431-8_59.

[6] Kenny Bastani. *Spring Cloud Example Project*. Jan. 30, 2021. URL: https://github.com/kbastani/spring-cloud-microservice-example (visited on 02/02/2021).

[7] Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10 (2022), pp. 20357–20374. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152803.

[8] Vincent Bushong et al. "On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study". In: *Applied Sciences* 11.17 (17 Jan. 2021), p. 7856. ISSN: 2076-3417. DOI: 10.3390/app11177856.

[9] Ivan Čilić et al. "Performance Evaluation of Container Orchestration Tools in Edge Computing Environments". In: *Sensors (Basel, Switzerland)* 23.8 (Apr. 15, 2023), p. 4008. ISSN: 1424-8220. DOI: 10.3390/s23084008. pmid: 37112349.

[10] *CloudLab*. Mar. 11, 2022. URL: https://cloudlab.us/ (visited on 03/11/2022).

[11] *CloudLab - Show Profile*. Mar. 18, 2022. URL: https://www.cloudlab.us/show-profile.php?project=PortalProfiles&profile=small-lan (visited on 03/18/2022).

[12] *Consul by HashiCorp*. Consul by HashiCorp. Mar. 11, 2022. URL: https://www.consul.io/ (visited on 03/11/2022).

[13] *DeathStarBench Open-Source Benchmark Suite for Cloud Microservices*. delimitrou, Feb. 1, 2021. URL: https://github.com/delimitrou/DeathStarBench (visited on 02/02/2021).

[14] Andrea Detti, Ludovico Funari, and Luca Petrucci. “muBench: An Open-Source Factory of Benchmark Microservice Applications”. In: *IEEE Transactions on Parallel and Distributed Systems* 34.3 (Mar. 1, 2023), pp. 968–980. ISSN: 1045-9219, 1558-2183, 2161-9883. DOI: 10.1109/TPDS.2023.3236447.

[15] *Docker: Accelerated Container Application Development*. URL: https://www.docker.com/#build (visited on 04/29/2024).

[16] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. Apr. 20, 2017. arXiv: 1606.04036 [cs].

[17] Dmitry Duplyakin et al. “The Design and Operation of CloudLab”. In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[18] *Experience Implementing Service Mesh on Nomad and Consul*. Prog.World. May 7, 2020. URL: https://prog.world/experience-implementing-service-mesh-on-nomad-and-consul/ (visited on 03/11/2022).

[19] Michael Ferdman et al. “Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware”. In: *Association for Computing Machinery* 40 (Mar. 3, 2012), p. 11. DOI: 10.1145/2150976.2150982.

[20] *Flannel*. Mar. 18, 2022. URL: https://github.com/flannel-io/flannel (visited on 03/18/2022).

[21] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. “Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption”. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017 IEEE International Conference on Software Architecture (ICSA). Gothenburg, Sweden: IEEE, Apr. 2017, pp. 21–30. ISBN: 978-1-5090-5729-0. DOI: 10.1109/ICSA.2017.24.

[22] Yu Gan. *An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems*. Yu Gan at Cornell ECE. Apr. 15, 2019. URL: https://gy1005.github.io/publication/2019.asplos.deathstarbench/ (visited on 01/28/2021).

[23] Johann Hauswald et al. “Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers”. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (2015), p. 223–238.

[24] Robert Heinrich et al. “Performance Engineering for Microservices: Research Challenges and Directions”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ICPE ’17: ACM/SPEC International Conference on Performance Engineering. L’Aquila Italy: ACM, Apr. 18, 2017, pp. 223–226. ISBN: 978-1-4503-4899-7. DOI: 10.1145/3053600.3053653.

[25] *How Nodes Work*. Docker Documentation. Mar. 10, 2022. URL: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/ (visited on 03/11/2022).

[26] P. Jamshidi et al. “Microservices: The Journey So Far and Challenges Ahead”. In: *IEEE Software* 35.3 (May 2018), pp. 24–35. ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039.

[27] Isam Mashhour Al Jawarneh et al. "Container Orchestration Engines: A Thorough Functional and Performance Comparison". In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. ICC 2019 - 2019 IEEE International Conference on Communications (ICC). May 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[28] Harshad Kasture and Daniel Sanchez. "Tailbench: A Benchmark Suite and Evaluation Methodology for Latency-Critical Applications". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016 IEEE International Symposium on Workload Characterization (IISWC). Providence, RI, USA: IEEE, Sept. 2016, pp. 1–10. ISBN: 978-1-5090-3896-1. DOI: 10.1109/IISWC.2016.7581261.

[29] Staci D. Kramer. *The Biggest Thing Amazon Got Right: The Platform*. Oct. 12, 2011. URL: https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/ (visited on 01/25/2021).

[30] *Kubernetes*. URL: https://kubernetes.io/ (visited on 04/29/2024).

[31] James Lewis and Martin Fowler. *Microservices*. martinfowler.com. URL: https://martinfowler.com/articles/microservices.html (visited on 01/13/2021).

[32] Shutian Luo et al. "Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21: ACM Symposium on Cloud Computing. Seattle WA USA: ACM, Nov. 2021, pp. 412–426. ISBN: 978-1-4503-8638-8. DOI: 10.1145/3472883.3487003.

[33] Anshita Malviya and Rajendra Kumar Dwivedi. "A Comparative Analysis of Container Orchestration Tools in Cloud Computing". In: *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*. 2022 9th International Conference on Computing for Sustainable Global Development (INDIACom). Mar. 2022, pp. 698–703. DOI: 10.23919/INDIACom54597.2022.9763171.

[34] Mohammad Sadegh Hamzehloui, Shamsul Sahibuddin, Ardavan Ashabi are with University Technology Malaysia, Malaysia et al. "A Study on the Most Prominent Areas of Research in Microservices". In: *International Journal of Machine Learning and Computing* 9.2 (Apr. 2019), pp. 242–247. ISSN: 20103700. DOI: 10.18178/ijmlc.2019.9.2.793.

[35] *MusicStore Sample Application*. GitHub. URL: https://github.com/SteeltoeOSS/Samples (visited on 02/02/2021).

[36] *Nomad by HashiCorp*. Nomad by HashiCorp. Mar. 11, 2022. URL: https://www.nomadproject.io/ (visited on 03/11/2022).

[37] *Options for Highly Available Topology*. Kubernetes. URL: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/ (visited on 03/11/2022).

[38] Christian Posta. *Netflix OSS, Spring Cloud, or Kubernetes? How About All of Them!* Software Blog. June 2, 2016. URL: https://blog.christianposta.com/microservices/netflix-oss-or-kubernetes-how-about-both/ (visited on 04/30/2024).

[39] Verified Market Research. *Cloud Microservices Market 2020 Trends, Market Share, Industry Size, Opportunities, Analysis and Forecast by 2026 – Market Reports*. URL: https://www.instanttechnews.com/technology-news/2020/02/16/cloud-microservices-market-2020-trends-market-share-industry-size-opportunities-analysis-and-forecast-by-2026/ (visited on 01/25/2021).

[40] Ryan A. Rossi and Nesreen K. Ahmed. "The Network Data Repository with Interactive Graph Analytics and Visualization". In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*. Ed. by Blai Bonet and Sven Koenig. AAAI Press, 2015, pp. 4292–4293.

[41] Johan Siebens. *Hashi-Up*. Mar. 16, 2022. URL: https://github.com/jsiebens/hashi-up (visited on 03/18/2022).

[42] *Sockshop Microservice Demo Application*. Microservices Demo, Feb. 2, 2021. URL: https://github.com/microservices-demo/microservices-demo (visited on 02/02/2021).

[43] *Spring Cloud*. Jan. 25, 2021. URL: https://spring.io/projects/spring-cloud (visited on 01/25/2021).

[44] Akshitha Sriraman and Thomas F. Wenisch. "muSuite: A Benchmark Suite for Microservices". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. 2018 IEEE International Symposium on Workload Characterization (IISWC). Raleigh, NC: IEEE, Sept. 2018, pp. 1–12. ISBN: 978-1-5386-6780-4. DOI: 10.1109/IISWC.2018.8573515.

[45] Gil Tene. *Wrk2*. Dec. 6, 2021. URL: https://github.com/giltene/wrk2 (visited on 12/06/2021).

[46] Alexandru Uta et al. "Is Big Data Performance Reproducible in Modern Cloud Networks?" In: *17th USENIX symposium on networked systems design and implementation (NSDI 20)* (2020).

[47] Erwin Van Eyk et al. "Serverless Is More: From PaaS to Present Cloud Computing". In: *IEEE Internet Computing* 22.5 (Sept. 2018), pp. 8–17. ISSN: 1089-7801, 1941-0131. DOI: 10.1109/MIC.2018.053681358.

[48] Joakim vonKistowski et al. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). Sept. 2018, pp. 223–236. DOI: 10.1109/MASCOTS.2018.00030.

[49] Harlow Ward. *Golang Microservices Example*. Dec. 4, 2021. URL: https://github.com/harlow/go-micro-services (visited on 12/06/2021).

[50] Naweiluo Zhou et al. "Container Orchestration on HPC Systems through Kubernetes". In: *J. Cloud Comput.* 10.1 (Feb. 22, 2021). ISSN: 2192-113X. DOI: 10.1186/s13677-021-00231-z.

[51] Xiang Zhou et al. "Benchmarking Microservice Systems for Software Engineering Research". In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ICSE '18: 40th International Conference on Software Engineering. Gothenburg Sweden: ACM, May 27, 2018, pp. 323–324. ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3194991.

Appendices

Appendix A

Preliminary experiments

In our experiments, we only highlighted a part of the performed experiments that we believe are most interesting for our reader. These experiments were also performed to evaluate the testbed and Docker Swarm orchestrator further and gives a place to some of the experiments that did take place but were not that interesting to discuss.

A.1 Experiment Parameters

Unless otherwise stated, experiments are benchmarked with wrk2 using the parameters 4 in the first part and 8 threads for the second part, 512 connections and ran 30 seconds with 500 requests with three master nodes and five worker nodes with one test client. The benchmark applications are orchestrated using Docker Swarm in the preliminary phase. The complete overview of the parameters used during preliminary experiments and orchestrator experiments (Experiment 12 and onwards) are shown in Table A.1, furthermore Table A.2 shows the hardware specification used for experimentation in this thesis.

Table A.1: Overview of the relevant parameters used during experimentation

Parameters	Set of values explored	Description
orchestrator	[’swarm’, ’k8s’, ’nomad’]	The orchestrator container engine
benchmark	[’hr’, ’mm’, ’sn’]	The benchmark application
n_client	[1 3]	The number of clients used for testing
requests	[200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000, 6000, 7000, 8000, 10000, 12000, 14000, 6000, 18000, 20000, 22000, 24000, 26000.]	The number of requests generated by wrk2
connections	[8 16 1024 128 2048 512]	The amount of open connections by wrk2
threads	[4 8 16]	The amount of threads used by wrk2
duration	[30 60 150]	The duration of the load by wrk2
infinite	[0 1]	Whether the deployment files have unlimited resources. Default is 1, limited.
baseline	[0 1]	Whether no additional modifications to the cluster is made.
availability	[0 1]	Whether the cluster has high-availability enabled or not.
vertical	[0 1]	Whether the containers are vertically scaled
horizontal	[0 1]	Whether the containers are horizontally scaled
runs	[N]	Amount of repeated runs per application

Table A.2: Overview of the resources of node type c6525-25g

c6525-25g	AMD EPYC Rome, 16 core, 2 disk, 25Gb Ethernet
CPU	16-core AMD 7302P at 3.00GHz
RAM	128GB ECC Memory (8x 16 GB 3200MT/s RDIMMs)
Disk	Two 480 GB 6G SATA SSD
NIC	Two dual-port Mellanox ConnectX-5 25Gb GB NIC (PCIe v4.0)

A.2 Example of generated data

All experiments are run with the wrk2 command build from the DeathStarBench suite, with some modifications by Shuang [13] the researcher of the original paper. The changes make wrk2 an open-loop (in contract to a closed-loop) load generator, which means the http requests are sent no matter if the previous requests have been completed or not. The example output below is generated with the following command ‘wrk -D exp -t 4 -c 512 -d 30s -L -s ./scripts/social-network/mixed-workload.lua http://localhost:8080/ -R 200‘. The output reports the input parameters, stats of each load generating thread, (detailed) latency distribution with stats about the max, mean, standard deviation, and total amount of requests received.

```
Using argument nginx
using  args: threads=4 and connections=512 and duration=30 and requests=500
```

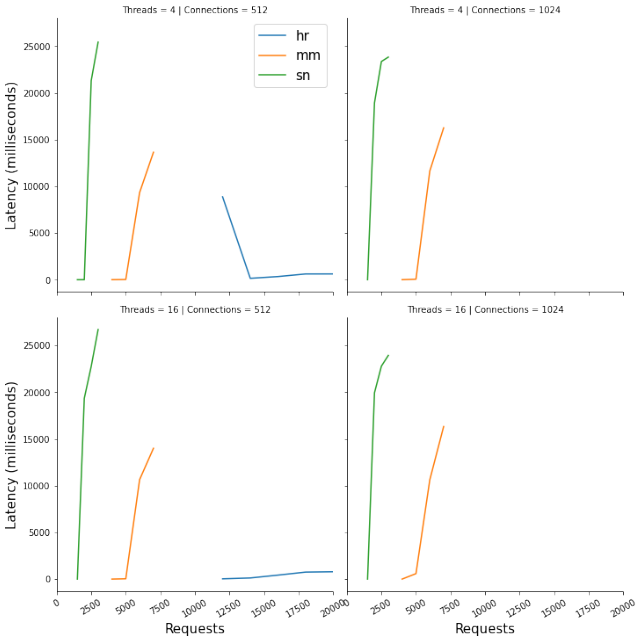


Figure A.1: The top figure is the same as the bottom image of Figure 5.6, the middle image shows the effect of time and the bottom image shows the effect of redeployment on latency.

```
Running 30s test @ http://10.10.1.7:5000
  4 threads and 512 connections
  Thread calibration: mean lat.: 2.796ms, rate sampling interval: 10ms
  Thread calibration: mean lat.: 2.597ms, rate sampling interval: 10ms
  Thread calibration: mean lat.: 2.554ms, rate sampling interval: 10ms
  Thread calibration: mean lat.: 2.744ms, rate sampling interval: 10ms
  Thread Stats   Avg      Stdev     99%   +/- Stdev
    Latency     2.57ms    1.64ms   7.75ms   80.18%
    Req/Sec   131.06    116.75   444.00     58.44%
  Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%    1.93ms
 75.000%    3.51ms
 90.000%    5.02ms
 99.000%    7.75ms
 99.900%    9.78ms
 99.990%   11.24ms
 99.999%   11.30ms
100.000%   11.30ms

  Detailed Percentile spectrum:
       Value   Percentile   TotalCount 1/(1-Percentile)

       0.370     0.000000            1         1.00
```

```
1.061   0.100000    991     1.11
1.277   0.200000   1986     1.25
1.575   0.300000   2971     1.43
1.758   0.400000   3968     1.67
1.930   0.500000   4956     2.00
2.038   0.550000   5446     2.22
2.171   0.600000   5941     2.50
2.367   0.650000   6435     2.86
2.975   0.700000   6932     3.33
3.507   0.750000   7425     4.00
3.727   0.775000   7675     4.44
3.929   0.800000   7921     5.00
4.119   0.825000   8168     5.71
4.363   0.850000   8418     6.67
4.687   0.875000   8664     8.00
4.859   0.887500   8787     8.89
5.023   0.900000   8914    10.00
5.215   0.912500   9034    11.43
5.423   0.925000   9158    13.33
5.671   0.937500   9282    16.00
5.831   0.943750   9345    17.78
5.983   0.950000   9405    20.00
6.103   0.956250   9468    22.86
6.267   0.962500   9530    26.67
6.563   0.968750   9591    32.00
6.707   0.971875   9623    35.56
6.823   0.975000   9653    40.00
6.947   0.978125   9684    45.71
7.095   0.981250   9715    53.33
7.339   0.984375   9746    64.00
7.423   0.985938   9761    71.11
7.535   0.987500   9778    80.00
7.695   0.989062   9792    91.43
7.831   0.990625   9808   106.67
8.007   0.992188   9823   128.00
8.059   0.992969   9831   142.22
8.207   0.993750   9839   160.00
8.327   0.994531   9846   182.86
8.559   0.995313   9854   213.33
```

```
       8.687     0.996094         9862       256.00
       8.775     0.996484         9866       284.44
       8.959     0.996875         9870       320.00
       9.151     0.997266         9873       365.71
       9.327     0.997656         9877       426.67
       9.519     0.998047         9881       512.00
       9.559     0.998242         9883       568.89
       9.631     0.998437         9885       640.00
       9.735     0.998633         9887       731.43
       9.775     0.998828         9889       853.33
       9.895     0.999023         9891      1024.00
       9.975     0.999121         9892      1137.78
      10.191     0.999219         9893      1280.00
      10.255     0.999316         9894      1462.86
      10.455     0.999414         9895      1706.67
      10.479     0.999512         9896      2048.00
      10.479     0.999561         9896      2275.56
      10.543     0.999609         9897      2560.00
      10.543     0.999658         9897      2925.71
      11.023     0.999707         9898      3413.33
      11.023     0.999756         9898      4096.00
      11.023     0.999780         9898      4551.11
      11.239     0.999805         9899      5120.00
      11.239     0.999829         9899      5851.43
      11.239     0.999854         9899      6826.67
      11.239     0.999878         9899      8192.00
      11.239     0.999890         9899      9102.22
      11.303     0.999902         9900     10240.00
      11.303     1.000000         9900          inf
#[Mean    =        2.571, StdDeviation   =        1.642]
#[Max     =       11.296, Total count    =         9900]
#[Buckets =           27, SubBuckets     =         2048]
----------------------------------------------------------
  14815 requests in 30.03s, 6.10MB read
  Socket errors: connect 0, read 0, write 0, timeout 1021
Requests/sec:    493.30
Transfer/sec:    208.09KB
```

***E.* Evaluating/Tuning the Influence of Configurations on the Performance of Nginx as a Web Server**

Our last experiment E is the tweaking of the front-end performance. The Social Network and media microservice both share Nginx (a reverse proxy) as their web server front-end, as this particular service was also tuned by the original paper. We will change the 'worker_connections' and 'worker_threads' from auto to the current threads and connections. Figure A.1 shows the effect of changing the parameters on the benchmarks. We can see that the load mm can handle is up to 5.000 Req/s.

Conclusion E: The workload on Social Network, breaks between 1.500 and 2.000 Req/s, except when the threads are set to 4 and the connections to 512.

A.2.1 Experiment 0 Exploring first run for workloads

With the default parameters, mentioned in Table 6.1, we run the experiment using 4 threads, 8 connections, and 30 seconds with 200 Req/s for each benchmark with finite resource off. Our results show the throughput of each benchmark: sn, mm and hr, throughput is rounded to 200 Req/s. Latency is 12.7ms for Hotel Reservation, 9.29ms for Media Microservices and 7.97ms for the Social Network. The results are shown in Figure A.2. Nothing noticeable so far.

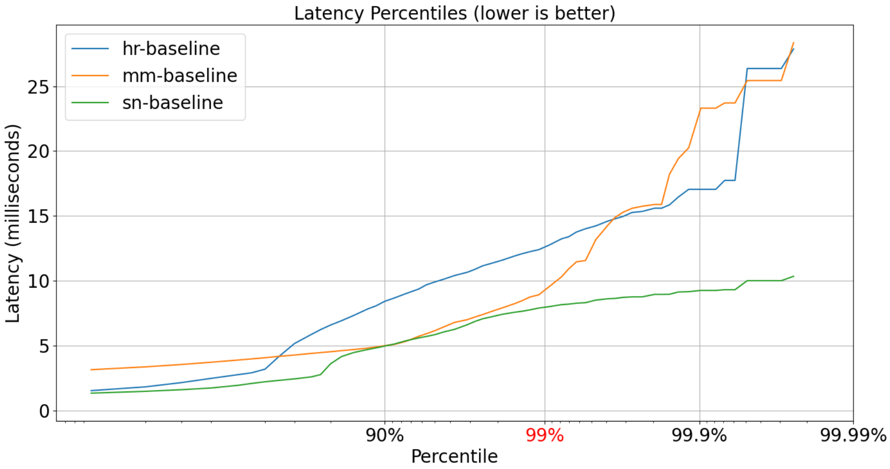
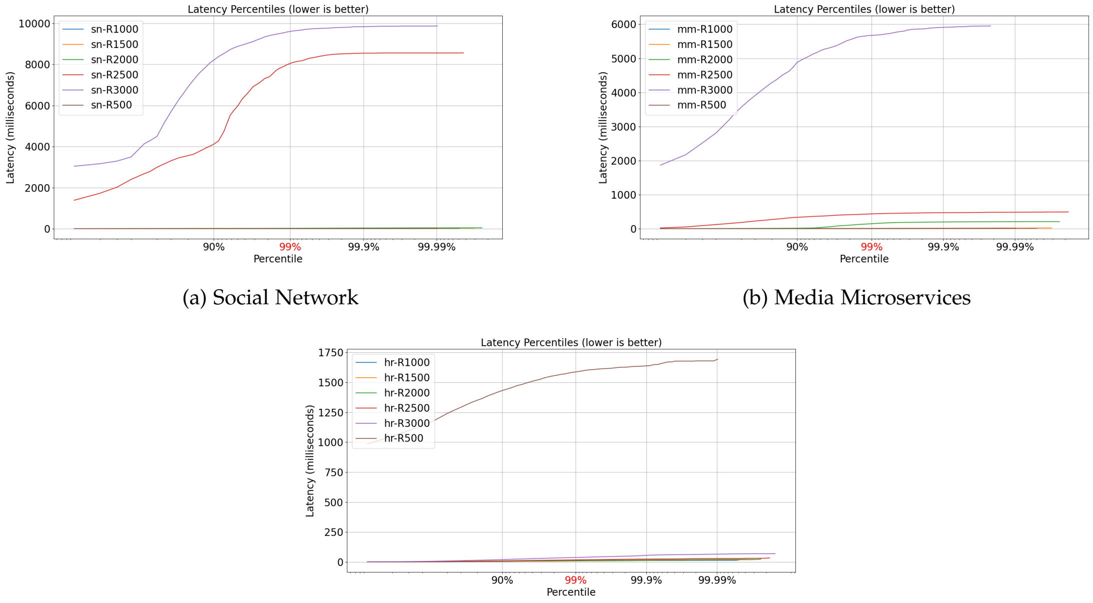


Figure A.2: Three clients measured (tail) latency for the benchmarks. The 99th percentile is our focus.

A.2.2 Experiment 1 Exploring the Parameter Space and Requests on Tail Latency

Our initial experiment to show the effect of changing each workload parameter on latency in each application. Each experiment is run with all parameters fixed and only varying one parameter. Req/s are 500, 1000, 1500, 2000 and 2500, 3000. Connections range 128, 512, 1024, 2048. Threads 4, 8, 16, for which we had to change the minimal amount of connection to 16 because the amount of connections needs to be higher than the number of threads. No resource limits have been applied.

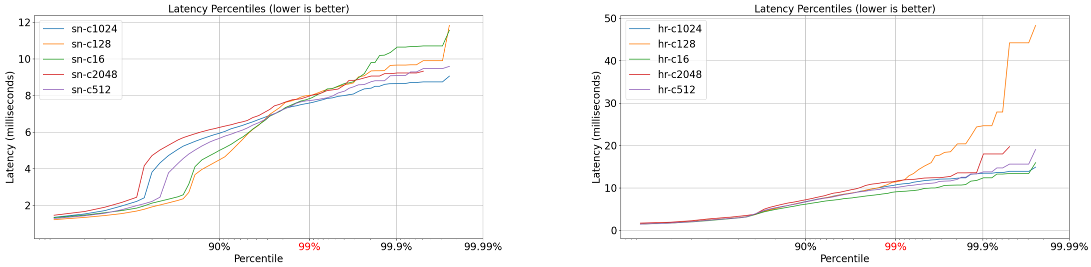


(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

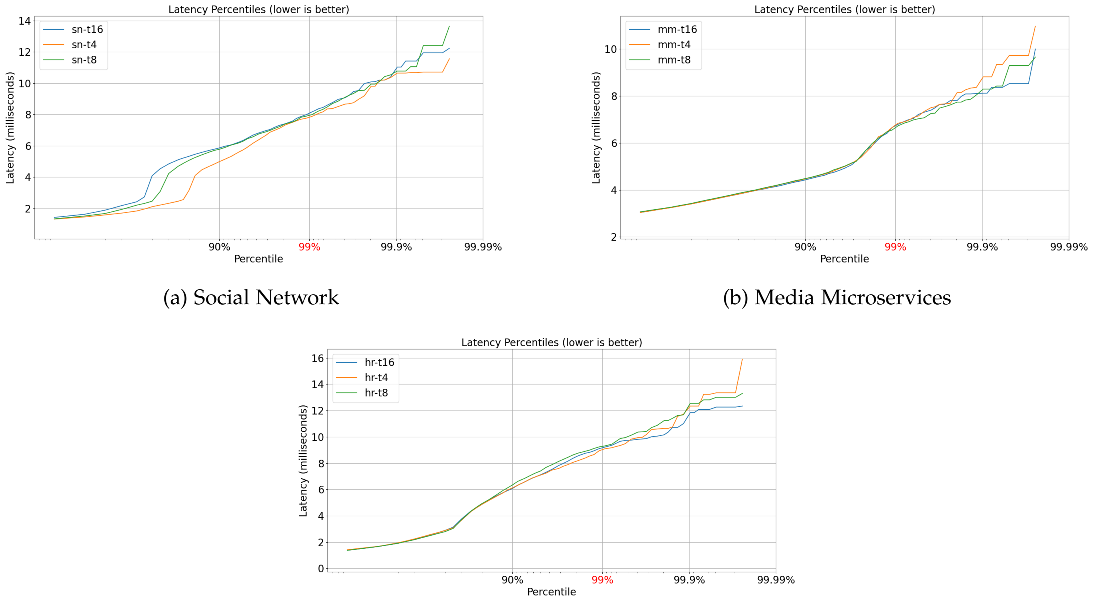
Figure A.3: The tail latency compared when all parameters are the same except the requests.



(a) Social Network

(b) Hotel Reservation

Figure A.4: The tail latency compared when all parameters are the same except the connections.



(a) Social Network

(b) Media Microservices

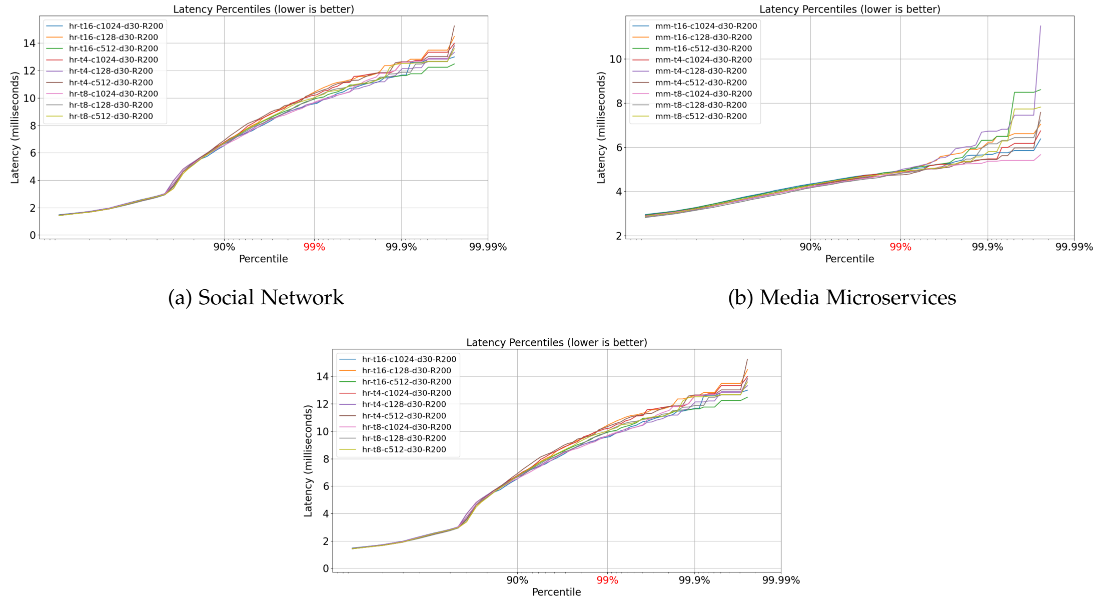
(c) Hotel Reservation

Figure A.5: The tail latency compared when all parameters are the same except the threads.

From the results shown in Figure A.4, Figure A.3, Figure A.5 only the increase of the amount of requests influences the latency of the applications in our initial experiment. SN shows latency larger than 1000ms from 2500 requests, MM shows increased latency when having around 3000 requests, HR seems to show increased latency when running the initial 500 requests but not thereafter.

A.2.3 Experiment 2 Rerun of Exploring Parameter Space of Workloads with fixed Requests

Our previous experiment 1 shows the effect of changing the requests can increase latency. To ensure the performance is not influenced by the threads and connections, we run experiments again, but keep the requests stable. We assume that enough connections need to be open with enough threads to properly stress the system. Each experiment is run, with only one parameter being varied. Requests from 200, 500. Connections from 128, 512, 1024. Threads from 4, 8, 16. Results are shown in Figure A.6 and Figure A.7.

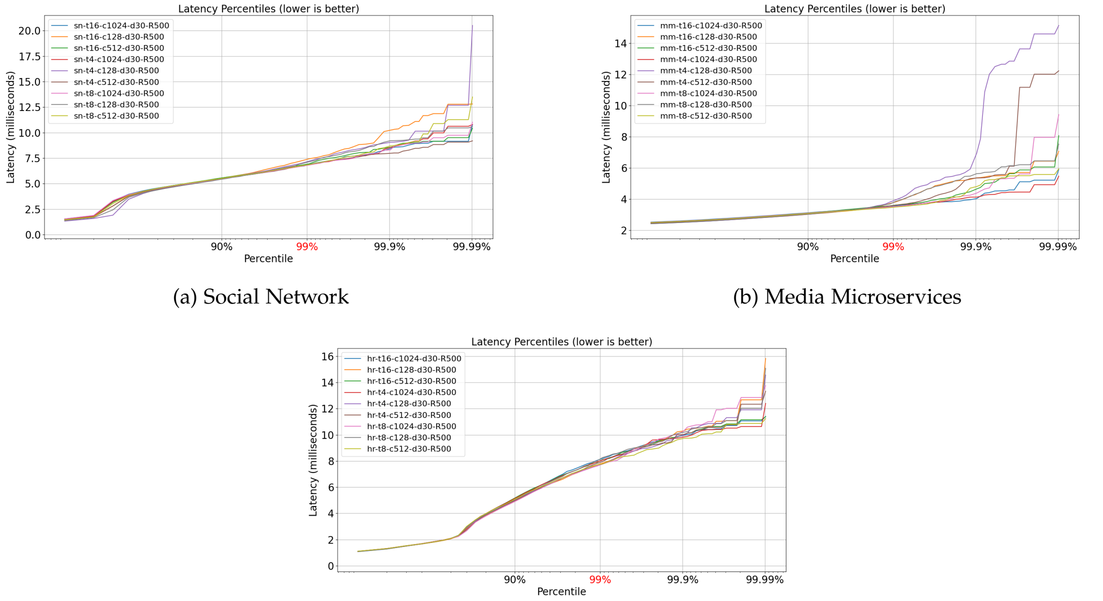


(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

Figure A.6: The tail latency compared to when workload has 200 requests with varying connections and threads.



(a) Social Network

(b) Media Microservices

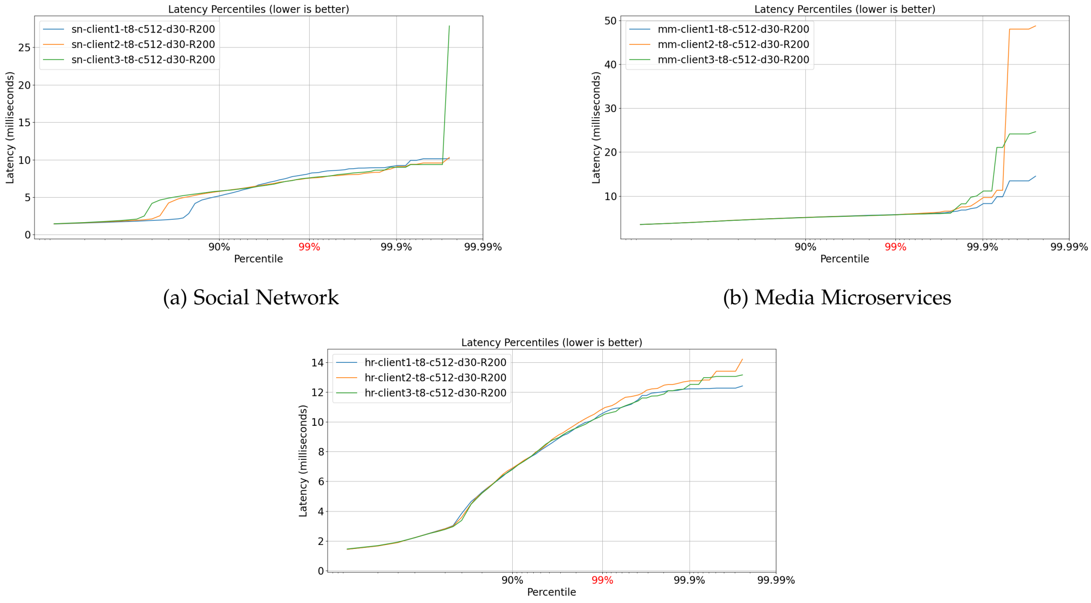
(c) Hotel Reservation

Figure A.7: The tail latency compared to when workload 500 requests with varying connections and threads.

Based on the results, we see no deviations to indicate that connections or threads require tuning for smaller amounts of requests.

A.2.4 Experiment 3 Exploring usage of multiple Test Clients

In this experiment, we use multiple clients to properly ensure that one or multiple clients does not affect the measured latency. The experiments are run with 8 threads, 512 connections with 500 requests for 30 seconds.



(a) Social Network

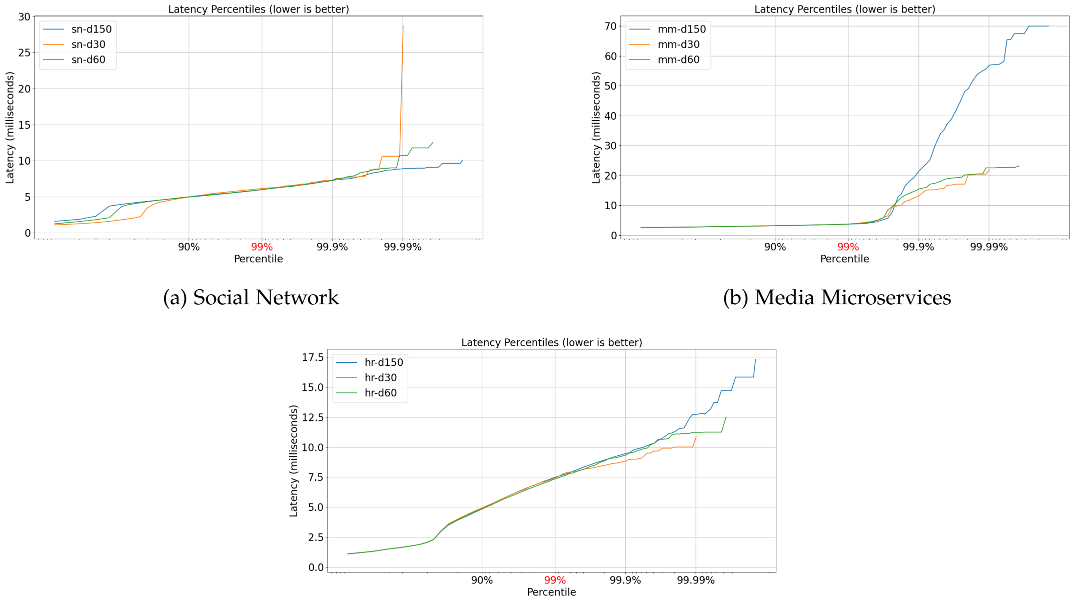
(b) Media Microservices

(c) Hotel Reservation

Figure A.8: The tail latency with multiple clients.

A.2.5 Experiment 4 Exploring Duration on the Latency

In experiment 4 we use the default parameters of 4 threads, 512 connections, and 500 requests to explore the effect of duration on the applications. The latency is measured with a duration of 30s, 60s, and 150s. All results are shown in Figure A.9.



(a) Social Network

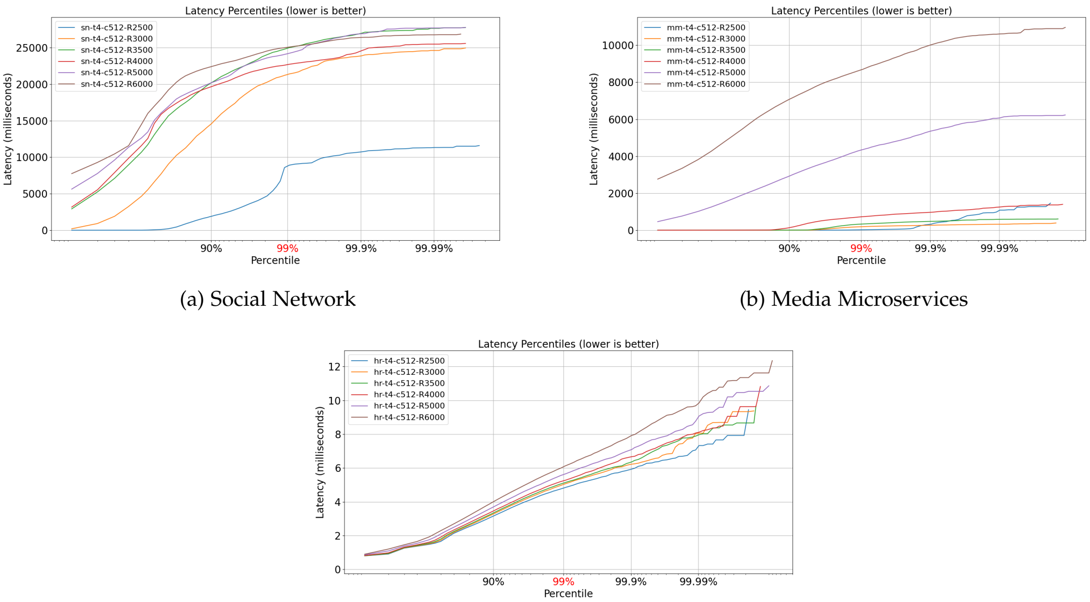
(b) Media Microservices

(c) Hotel Reservation

Figure A.9: The tail latency using the default parameters but with a duration of 30, 60 and 150 seconds.

A.2.6 Experiment 5 Breaking Points Run 1

In experiment 5 we stress the applications again. From experiment 1 we derived the breaking point of SN and MM to be around 2500 and 3000 requests. HR could handle at least 3000 requests. The latency will be measured with the requests rates 2500, 3000, 3500, 4000, 5000 and 6000. The results are shown in Figure A.10.



(a) Social Network

(b) Media Microservices

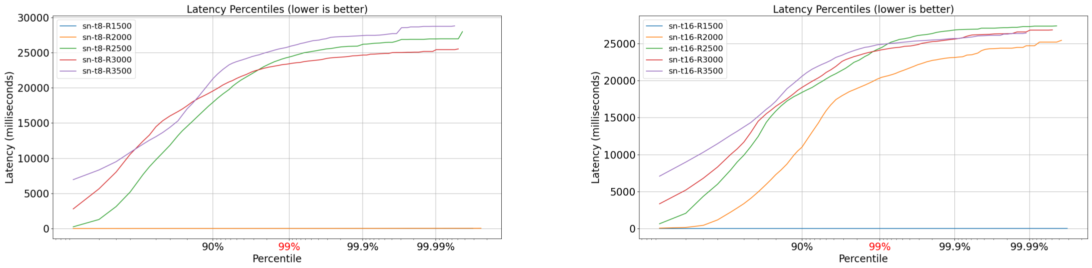
(c) Hotel Reservation

Figure A.10: The tail latency measured with varying requests.

The results show that SN is already fully stressed with a latency of 8.79s at the 2500 request rate. MM latency can handle up to 4000 requests before exceeding the 1s latency threshold with 5000 and onwards. HR shows increased latency, but with a latency of around 5ms to 6ms we need to further investigate the required amount of requests for it to have a noticeable latency increase.

A.2.7 Experiment 6 Rerun Breaking point with different Parameters

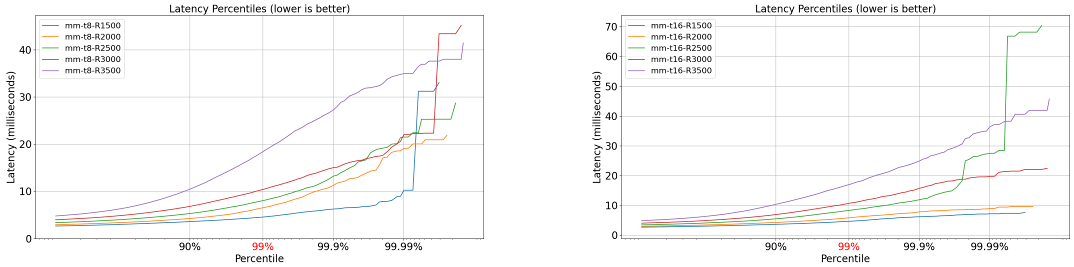
Now we rerun the experiment of 5 again and increase the threads and connections to see if the applications perform similarly. We would assume SN to break again around 2500, MM to break around 5000 and HR to be able to handle the workloads. Furthermore, the original paper of DSB states limitations regarding the Nginx service for the maximum number of connections it is able to handle. The results are shown in Figure A.11, Figure A.12 and Figure A.13.



(a) Social Network 8 threads

(b) Social Network 16 threads

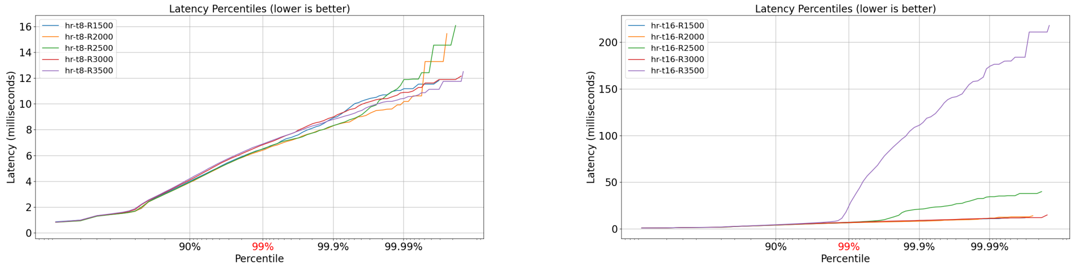
Figure A.11: The tail latency measured with 8 and 16 threads and 512 or 1024 connections for the Social Network.



(a) Media Microservices 8 threads

(b) Media Microservices 16 threads

Figure A.12: The tail latency measured with 8 and 16 threads and 512 or 1024 connections for the Media Microservices.



(a) Hotel Reservation 8 threads

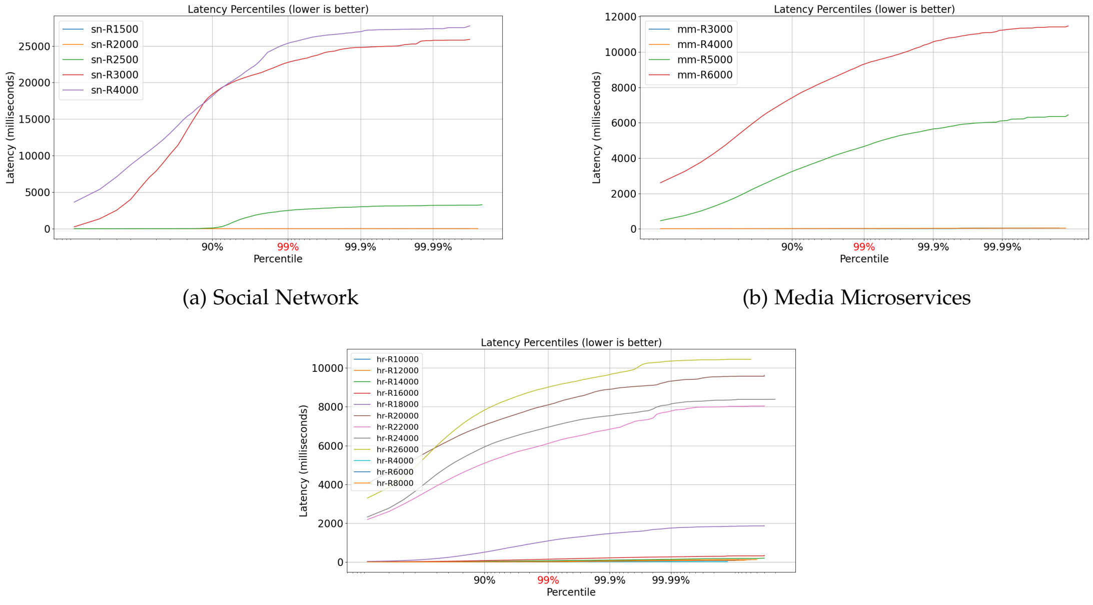
(b) Hotel Reservation 16 threads

Figure A.13: The tail latency measured with 8 and 16 threads and 512 or 1024 connections for the Hotel Reservation.

The results do have some noticeable influence on the latency. When changing the amount of threads, we see that SN breaks with 8 threads around 3000 requests and with 16 threads earlier around 2000 requests. MM shows similar results, but more requests should have been used. HR shows similar latency around 8 threads independent of the current input latency, but in the case of 16 threads with 3500 requests we see an increase of the latency although.

A.2.8 Experiment 7 Rerun Stress Applications with new Parameters

Experiment 7 we further explore the required amount of requests to stress the applications with the current default parameters, Table 5.1. Social Network is are run the requests 1500, 2000, 2500, 3000 and 4000, Media Microservices requests are 3000, 4000, 5000, 6000 and Hotel Reservation is run from 12000, 14000, 16000, 18000, 20000, 22000, 24000, 26000. Figure A.14.



(a) Social Network

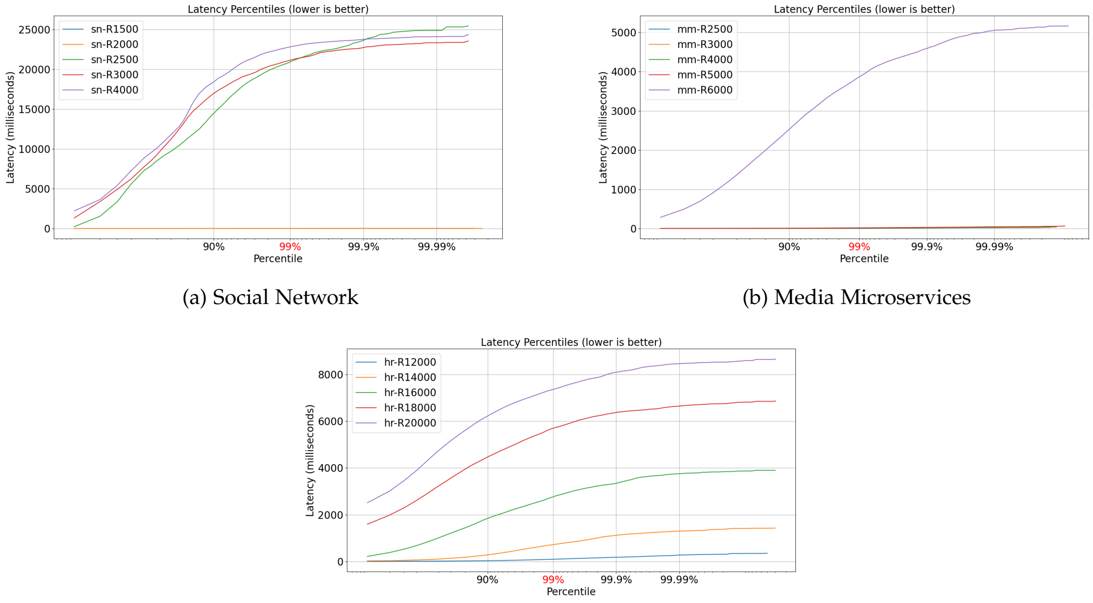
(b) Media Microservices

(c) Hotel Reservation

Figure A.14: The tail latency with multiple clients.

A.2.9 Experiment 8 Exploring time between experiments as factor on performance

In this experiment we simply look if the state of the experiments change if we leave any time in between experiments, some temporal aspect or cooling-down or warm up effect, the experiment is run without limiting the resources. We run the Social Network with 1500 2000 2500 3000 4000 Req/s, media microservice with 2500 3000 4000 5000 6000 Req/s and Hotel Reservation with 12000 14000 16000 18000 20000 Req/s. Between every run, we sleep 60 seconds. The results are shown in Figure A.15.



(a) Social Network

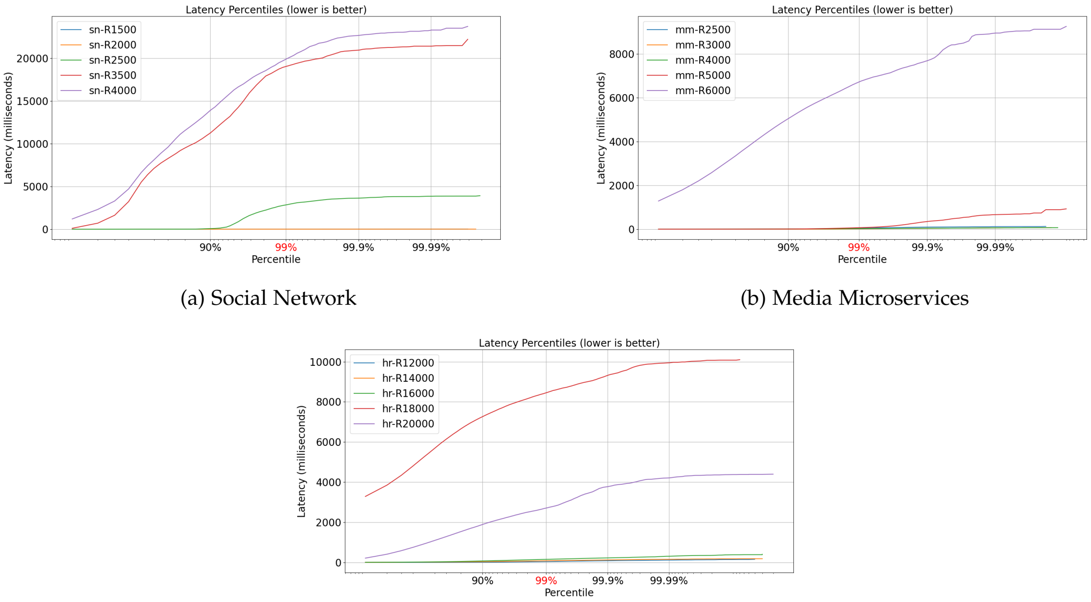
(b) Media Microservices

(c) Hotel Reservation

Figure A.15: The tail latency measured with 60 seconds between each experiment workload.

A.2.10 Experiment 9 Full redeploy of applications after each run

We look at the results given that we redeploy each application after each experiment run, only varying the amount of requests. We run Social Network with 1500, 2000, 2500, 3500, 4000, requests, Media Microservices with 2500, 3000, 4000, 5000, 6000 and Hotel Reservation with 12000, 14000, 16000, 18000, 20000 Req/s. The results are shown in Figure A.16.



(a) Social Network

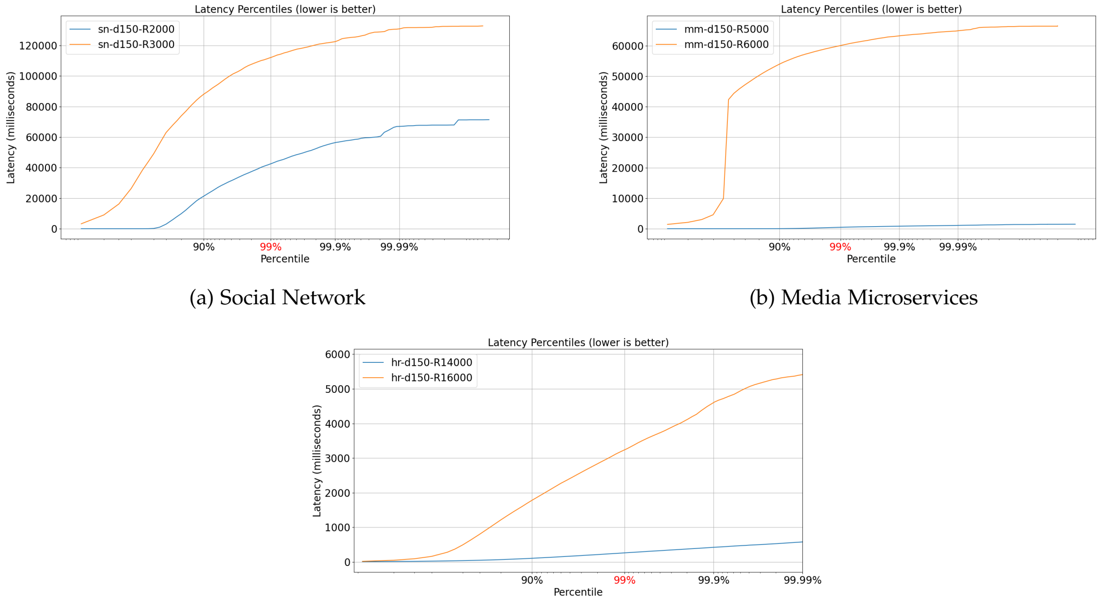
(b) Media Microservices

(c) Hotel Reservation

Figure A.16: The tail latency measured between application, but where we redeploy the application.

A.2.11 Experiment 10 Rerun to confirm breaking points

In experiment 10 we run the workloads again for with the current belief of requests to break the system, except that we increase the duration of each experiment. In the case of SN we believe between 2000 and 3000, MM we believe between 5000 and 6000, HR is expected to be 14000 and 16000 requests. The results confirm that the breaking points are similar Figure A.17.



(a) Social Network

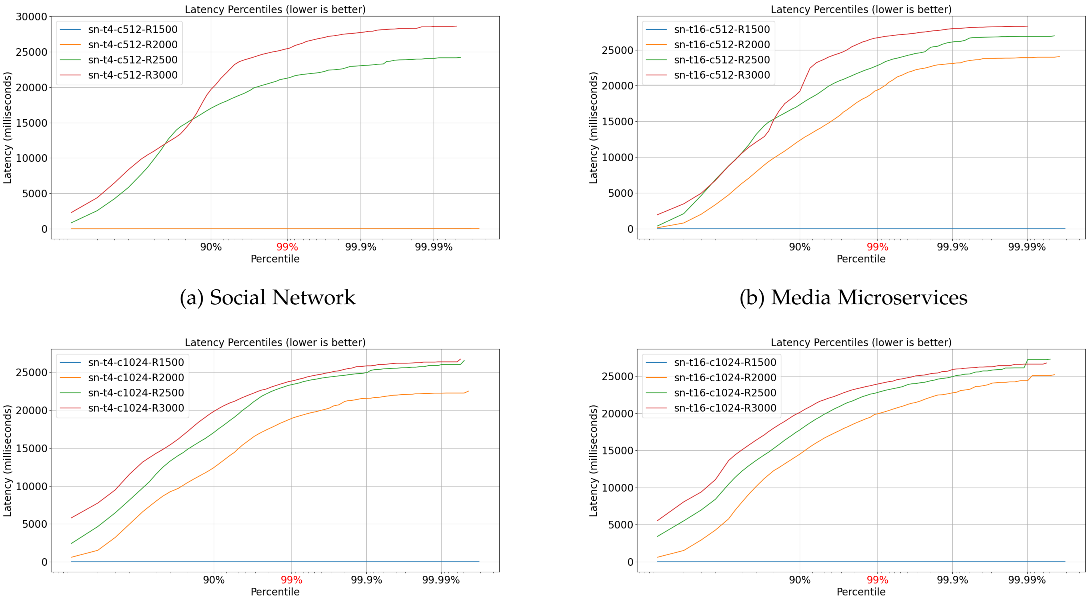
(b) Media Microservices

(c) Hotel Reservation

Figure A.17: The tail latency measured with an increased duration.

A.2.12 Experiment 11 Nginx configurations experiments on performance

In experiment 11 we configure Nginx to set the amount of connections and threads to 'auto' which we assume not to change the current performance and rerun all setups to ensure consistency. The results are shown in Figure A.18. This time we run with 200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000, 6000, 7000 Req/s and 512 connections and 4 threads. The Hotel Reservation was run with 200, 500, 1000, 1500, 2500, 3500, 4000, 5000, 6000, 7000 Req/s and 512 connections and 8 threads and Hotel Reservation with 8000, 9000, 10000, 11000, 12000, 13000, 14000, 15000, 16000 Req/s. The results are shown in Figure A.19, Figure A.20.



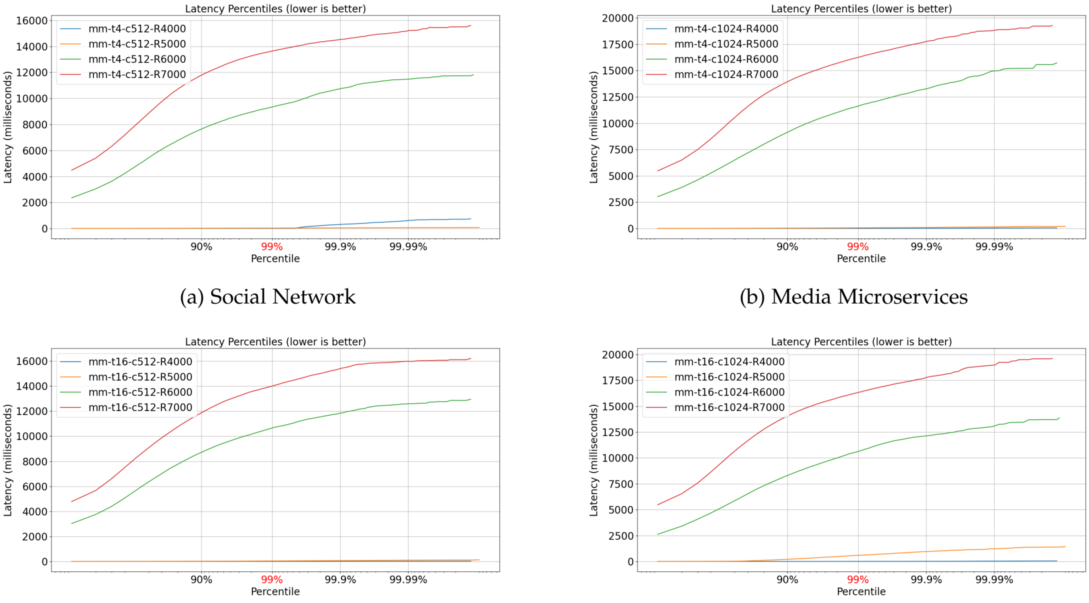
(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

(d) Hotel Reservation

Figure A.18: The tail latency measured with Social Network



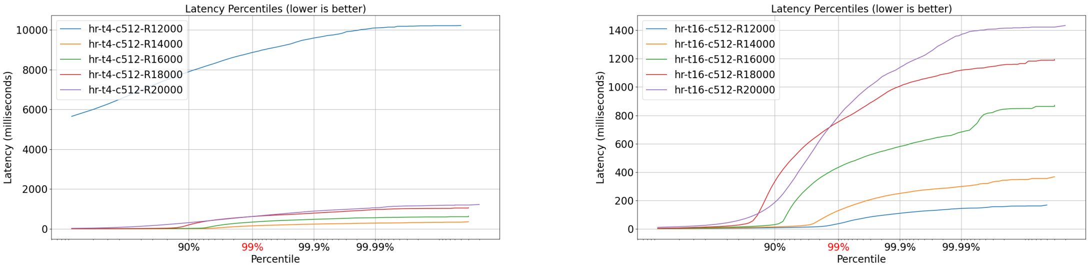
(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

(d) Hotel Reservation

Figure A.19: The tail latency measured with Media Microservices



(a) Social Network

(b) Media Microservices

Figure A.20: The tail latency measured with Hotel Reservation

Appendix B

Orchestrator Experiments

This chapter consists of an overview of all experiments concerning the orchestrators and can provide more insights. The applications are benchmarked with wrk2 using the parameters 4 threads, 512 connections, and ran for 30 seconds with 500 requests with 3 master nodes and 5 worker nodes with 1 test client. The benchmark applications are orchestrated using Docker Swarm, Kubernetes and Nomad. The complete overview of the parameters used during preliminary experiments and orchestrator experiments (Experiment 12 and onwards combined in these sections) are shown in Table B.1.

Table B.1: Overview of the relevant parameters used during experimentation

Parameters	Set of values explored	Description
orchestrator	['swarm', 'k8s', 'nomad']	The orchestrator container engine
benchmark	['hr', 'mm', 'sn']	The benchmark application
n_client	[1]	The number of clients used for testing
requests	[200, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 5000, 6000, 7000, 8000, 10000, 12000, 14000, 6000, 18000, 20000, 22000, 24000, 26000.]	The number of requests generated by wrk2
connections	[512]	The amount of open connections by wrk2, similar to users
threads	[4]	The amount of threads used by wrk2
duration	[30]	The duration of the load by wrk2
infinite	[1]	Whether the deployment files have unlimited resources. Default is 1, limited.
baseline	[0 1]	Whether no additional modifications to the cluster is made.
availability	[0 1]	Whether the cluster has high-availability enabled or not.
vertical	[0 1]	Whether the containers are vertically scaled
horizontal	[0 1]	Whether the containers are horizontally scaled

B.0.1 Overview Orchestrators

An overview of the latency and throughput measured for the orchestrator with a baseline compared to vertical, horizontal scaling and turning high-availability off. Figure B.1

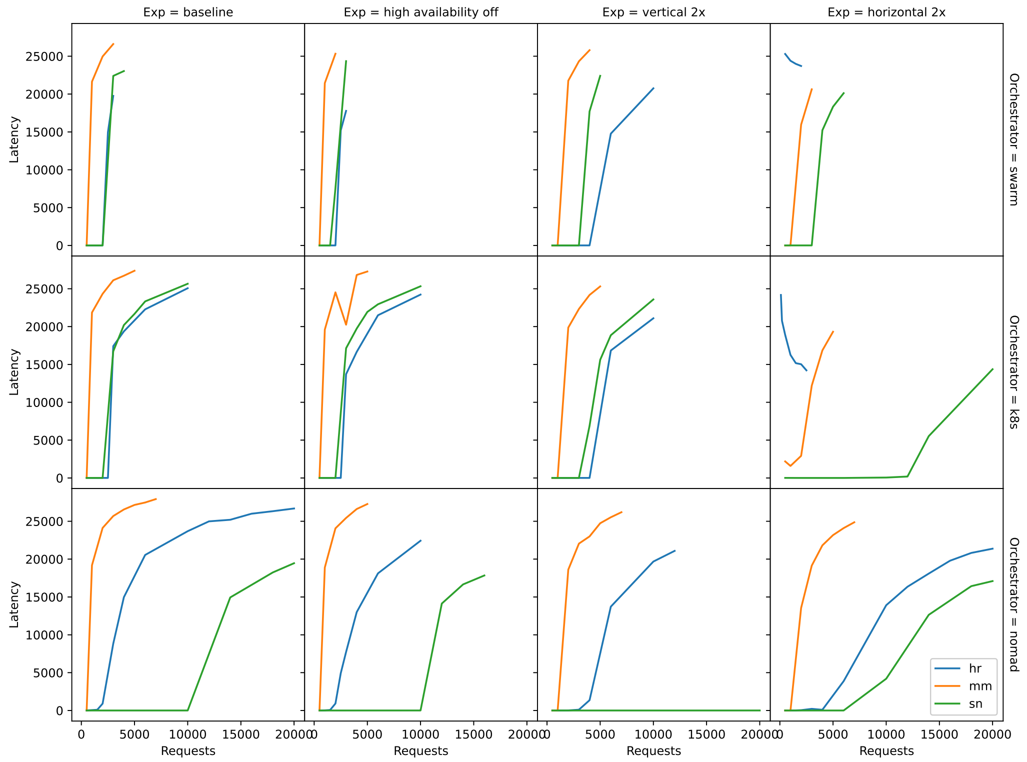


Figure B.1: Latency of the different experiments and orchestrators with increasing load.

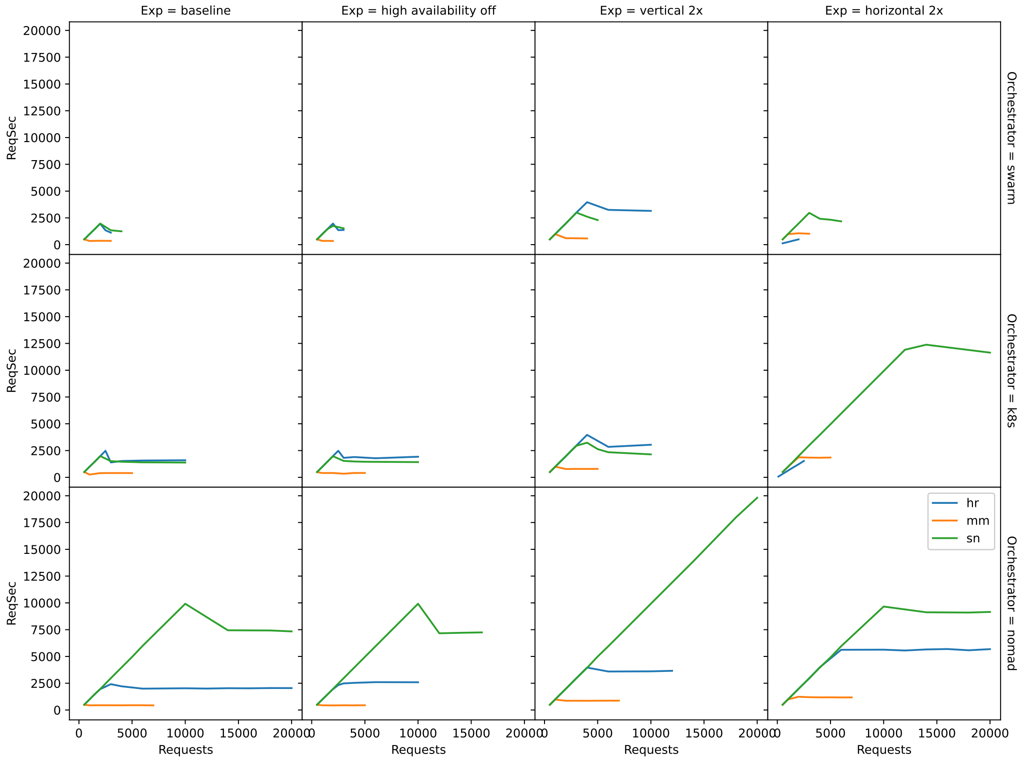
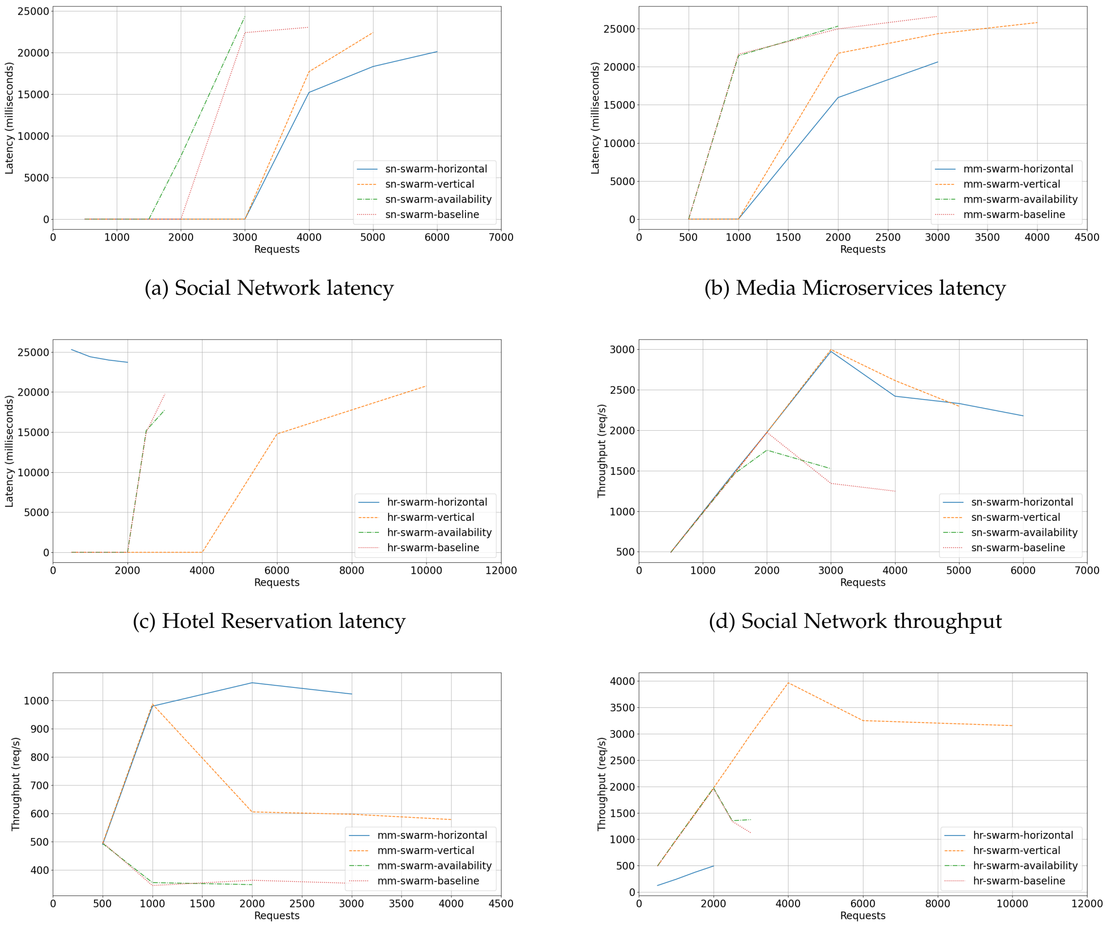


Figure B.2: Throughput of the different experiments and orchestrators with increasing input requests.

B.1 Docker Swarm Experiment All Figures



(a) Social Network latency

(b) Media Microservices latency

(c) Hotel Reservation latency

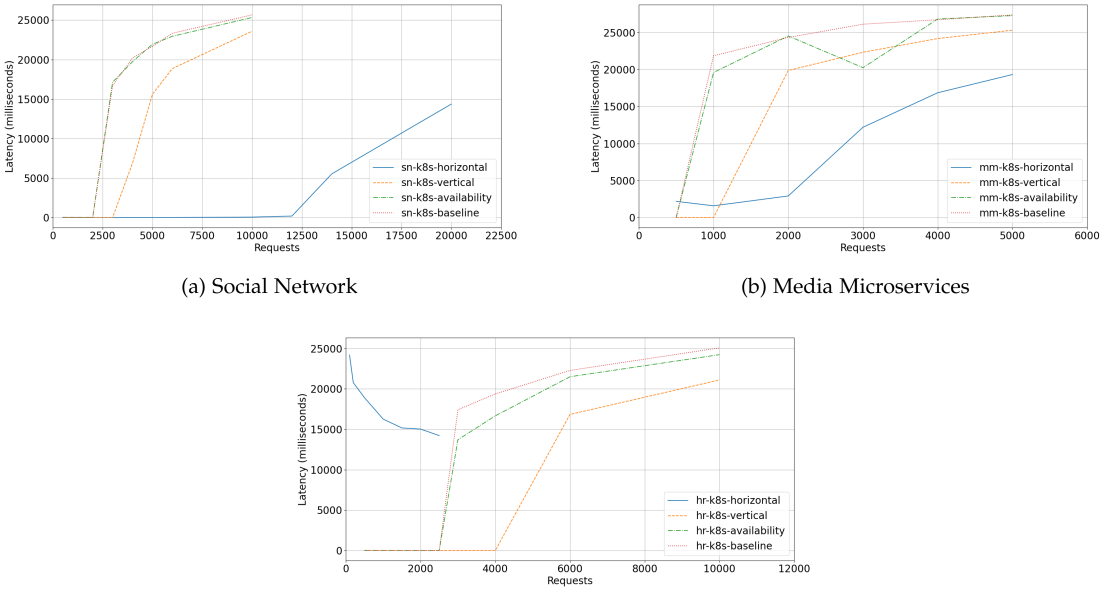
(d) Social Network throughput

(e) Media Microservices throughput

(f) Hotel Reservation throughput

Figure B.3: Social Network tail latency and throughput compared in the scaling scenarios.

B.2 Kubernetes Experiment All Figures

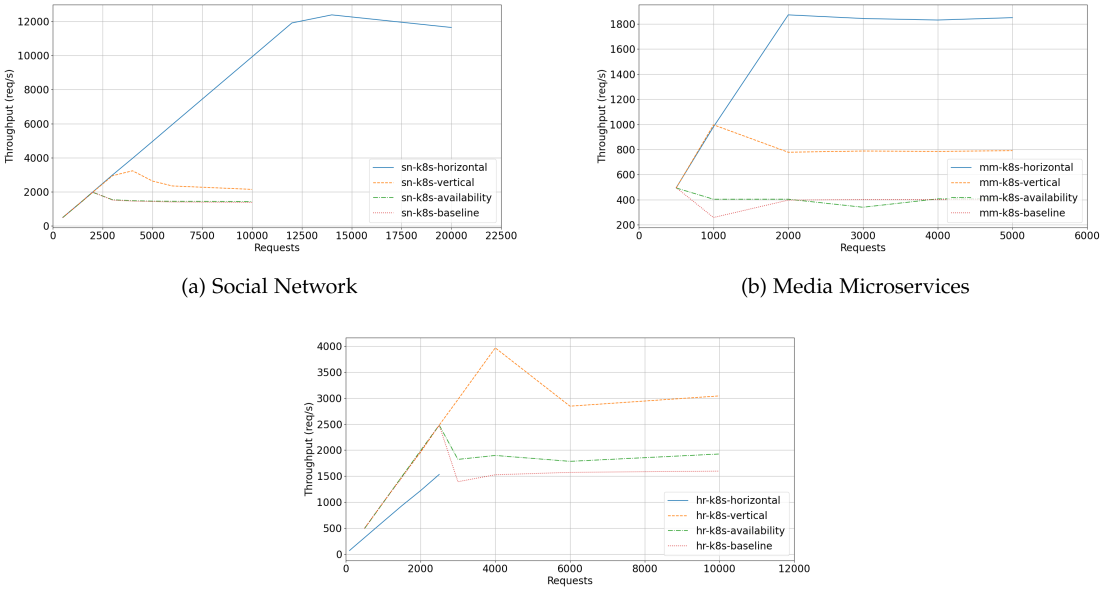


(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

Figure B.4: The effect of the parameters on the tail latency for the benchmarks



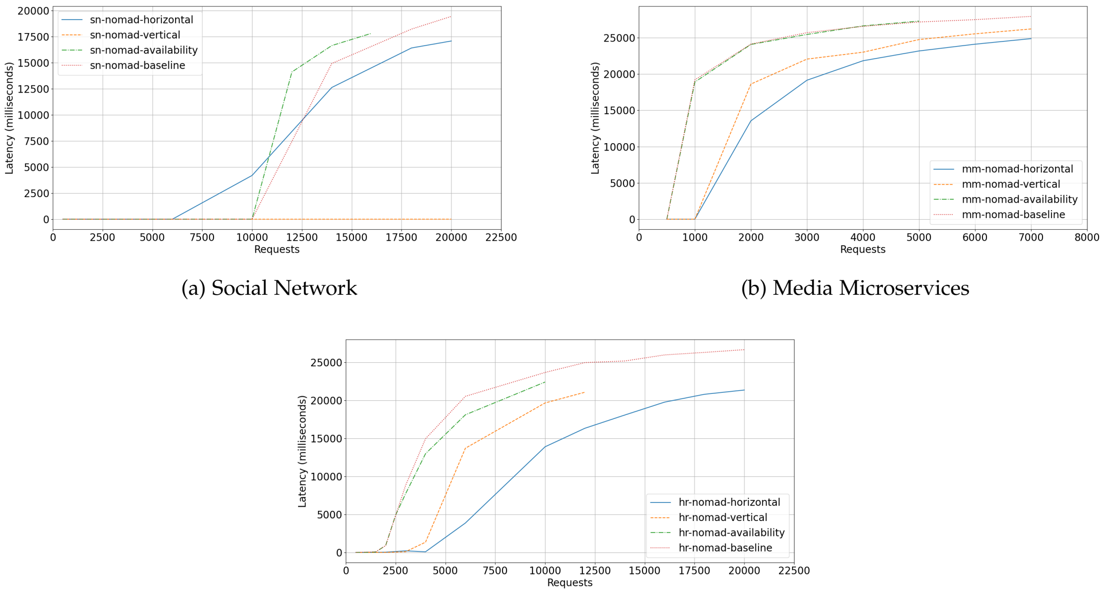
(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

Figure B.5: The effect of the parameters on the tail latency for the benchmarks

B.3 Nomad Experiment All Figures

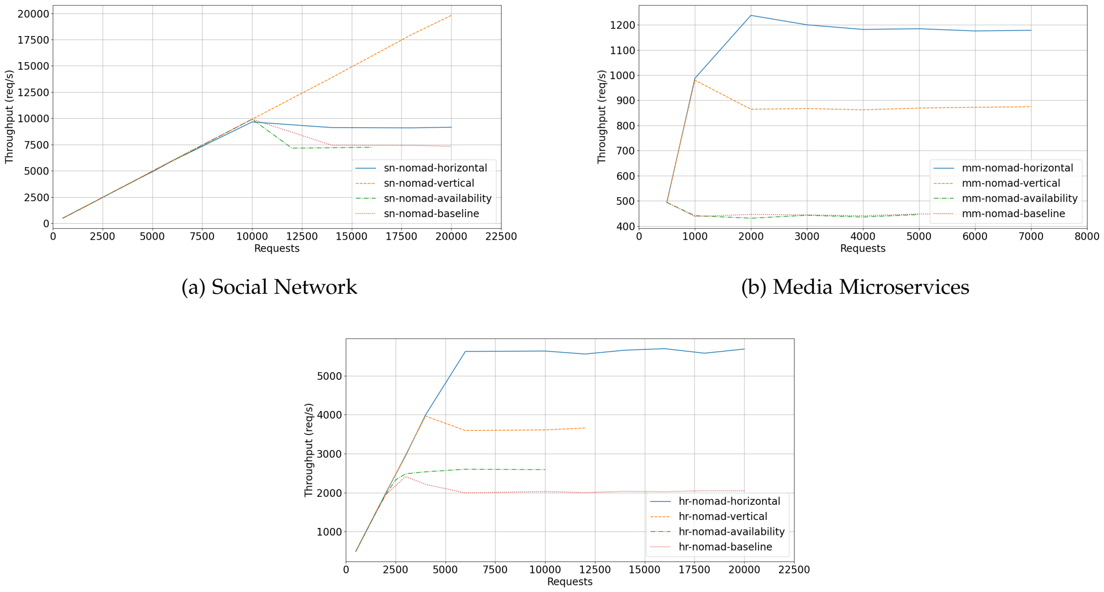


(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

Figure B.6: The effect of the parameters on the tail latency for the benchmarks



(a) Social Network

(b) Media Microservices

(c) Hotel Reservation

Figure B.7: The effect of the parameters on the throughput for the benchmarks

Appendix C

Workload

The workload scripts, created for each benchmark and written in Lua. No modification have been made from the original workload in the DeathStarBench suite for our experiments.

C.1 Mixed workload Social-network

```lua
require "socket"
local time = socket.gettime()*1000
math.randomseed(time)
math.random(); math.random(); math.random()

local charset = {'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's',
  'd', 'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm', 'Q',
  'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', 'A', 'S', 'D', 'F', 'G', 'H',
  'J', 'K', 'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '1', '2', '3', '4', '5',
  '6', '7', '8', '9', '0'}

local decset = {'1', '2', '3', '4', '5', '6', '7', '8', '9', '0'}

local function stringRandom(length)
  if length > 0 then
    return stringRandom(length - 1) .. charset[math.random(1, #charset)]
  else
    return ""
  end
end

```

```
local function decRandom(length)
  if length > 0 then
    return decRandom(length - 1) .. decset[math.random(1, #decset)]
  else
    return ""
  end
end

local function compose_post()
  local user_index = math.random(1, 962)
  local username = "username_" .. tostring(user_index)
  local user_id = tostring(user_index)
  local text = stringRandom(256)
  local num_user_mentions = math.random(0, 5)
  local num_urls = math.random(0, 5)
  local num_media = math.random(0, 4)
  local media_ids = '['
  local media_types = '['

  for i = 0, num_user_mentions, 1 do
    local user_mention_id
    while (true) do
      user_mention_id = math.random(1, 962)
      if user_index ~= user_mention_id then
        break
      end
    end
    text = text .. " @username_" .. tostring(user_mention_id)
  end

  for i = 0, num_urls, 1 do
    text = text .. " http://" .. stringRandom(64)
  end

  for i = 0, num_media, 1 do
    local media_id = decRandom(18)
    media_ids = media_ids .. "\"" .. media_id .. "\","
    media_types = media_types .. "\"png\","
  end

  media_ids = media_ids:sub(1, #media_ids - 1) .. "]"
  media_types = media_types:sub(1, #media_types - 1) .. "]"

  local method = "POST"
  local path = "http://localhost:8080/wrk2-api/post/compose"
  local headers = {}
  local body
  headers["Content-Type"] = "application/x-www-form-urlencoded"
  if num_media then
    body   = "username=" .. username .. "&user_id=" .. user_id ..
        "&text=" .. text .. "&media_ids=" .. media_ids ..
        "&media_types=" .. media_types .. "&post_type=0"
  else
    body   = "username=" .. username .. "&user_id=" .. user_id ..
        "&text=" .. text .. "&media_ids=" .. "&post_type=0"
  end

  return wrk.format(method, path, headers, body)
end

local function read_user_timeline()
  local user_id = tostring(math.random(1, 962))
  local start = tostring(math.random(0, 100))
  local stop = tostring(start + 10)

  local args = "user_id=" .. user_id .. "&start=" .. start .. "&stop=" .. stop
  local method = "GET"
  local headers = {}
  headers["Content-Type"] = "application/x-www-form-urlencoded"
  local path = "http://localhost:8080/wrk2-api/user-timeline/read?" .. args
  return wrk.format(method, path, headers, nil)
end

local function read_home_timeline()
    local user_id = tostring(math.random(1, 962))
    local start = tostring(math.random(0, 100))
    local stop = tostring(start + 10)

    local args = "user_id=" .. user_id .. "&start=" .. start .. "&stop=" .. stop
    local method = "GET"
    local headers = {}
    headers["Content-Type"] = "application/x-www-form-urlencoded"
    local path = "http://localhost:8080/wrk2-api/home-timeline/read?" .. args
    return wrk.format(method, path, headers, nil)
  end

request = function()
    cur_time = math.floor(socket.gettime())
    local read_home_timeline_ratio = 0.60
    local read_user_timeline_ratio = 0.30
    local compose_post_ratio       = 0.10

    local coin = math.random()
    if coin < read_home_timeline_ratio then
      return read_home_timeline()
    elseif coin < read_home_timeline_ratio + read_user_timeline_ratio then
      return read_user_timeline()
    else
      return compose_post()
    end
  end
```

C.2 Compose workload Media-microservices

Note, the original file contains all movies titles

```
require "socket"
time = socket.gettime()*1000
math.randomseed(time)
math.random(); math.random(); math.random()

local charset = {'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's',
  'd', 'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm', 'Q',
  'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', 'A', 'S', 'D', 'F', 'G', 'H',
  'J', 'K', 'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '1', '2', '3', '4', '5',
  '6', '7', '8', '9', '0'}

local movie_titles = {
  "Avengers: Endgame",
  "Kamen Rider Heisei Generations FOREVER",
  "Captain Marvel",
  "Pokémon Detective Pikachu"
}

function string.random(length)
  if length > 0 then
    return string.random(length - 1) .. charset[math.random(1, #charset)]
  else
    return ""
  end
end

request = function()
  local movie_index = math.random(1000)
  local user_index = math.random(1000)
  local username = "username_" .. tostring(user_index)
  local password = "password_" .. tostring(user_index)
  local title = urlEncode(movie_titles[movie_index])
  local rating = math.random(0, 10)
  local text = string.random(256)

  local path = "http://127.0.0.1:8080/wrk2-api/review/compose"
  local method = "POST"
  local headers = {}
  local body = "username=" .. username .. "&password=" .. password .. "&title=" ..
                  title .. "&rating=" .. rating .. "&text=" .. text
  headers["Content-Type"] = "application/x-www-form-urlencoded"

  return wrk.format(method, path, headers, body)
end

function urlEncode(s)
     s = string.gsub(s, "([^%w%.%- ])", function(c) return string.format("%%%02X", string.byte(c)) end)
    return string.gsub(s, " ", "+")
end
```

C.3 Mixed workload Hotel-reservation

```
require "socket"
math.randomseed(socket.gettime()*1000)
math.random(); math.random(); math.random()

local function get_user()
  local id = math.random(0, 500)
  local user_name = "Cornell_" .. tostring(id)
  local pass_word = ""
  for i = 0, 9, 1 do
    pass_word = pass_word .. tostring(id)
  end
  return user_name, pass_word
end

local function search_hotel()
  local in_date = math.random(9, 23)
  local out_date = math.random(in_date + 1, 24)

  local in_date_str = tostring(in_date)
  if in_date <= 9 then
    in_date_str = "2015-04-0" .. in_date_str
  else
    in_date_str = "2015-04-" .. in_date_str
  end

  local out_date_str = tostring(out_date)
  if out_date <= 9 then
    out_date_str = "2015-04-0" .. out_date_str
  else
    out_date_str = "2015-04-" .. out_date_str
  end

  local lat = 38.0235 + (math.random(0, 481) - 240.5)/1000.0
  local lon = -122.095 + (math.random(0, 325) - 157.0)/1000.0

  local method = "GET"
  local path = "http://localhost:5000/hotels?inDate=" .. in_date_str ..
```

```
    "&outDate=" .. out_date_str .. "&lat=" .. tostring(lat) .. "&lon=" .. tostring(lon)

  local headers = {}
  -- headers["Content-Type"] = "application/x-www-form-urlencoded"
  return wrk.format(method, path, headers, nil)
end

local function recommend()
  local coin = math.random()
  local req_param = ""
  if coin < 0.33 then
    req_param = "dis"
  elseif coin < 0.66 then
    req_param = "rate"
  else
    req_param = "price"
  end

  local lat = 38.0235 + (math.random(0, 481) - 240.5)/1000.0
  local lon = -122.095 + (math.random(0, 325) - 157.0)/1000.0

  local method = "GET"
  local path = "http://localhost:5000/recommendations?require=" .. req_param ..
    "&lat=" .. tostring(lat) .. "&lon=" .. tostring(lon)
  local headers = {}
  -- headers["Content-Type"] = "application/x-www-form-urlencoded"
  return wrk.format(method, path, headers, nil)
end

local function reserve()
  local in_date = math.random(9, 23)
  local out_date = in_date + math.random(1, 5)

  local in_date_str = tostring(in_date)
  if in_date <= 9 then
    in_date_str = "2015-04-0" .. in_date_str
  else
    in_date_str = "2015-04-" .. in_date_str
  end

  local out_date_str = tostring(out_date)
  if out_date <= 9 then
    out_date_str = "2015-04-0" .. out_date_str
  else
    out_date_str = "2015-04-" .. out_date_str
  end

  local hotel_id = tostring(math.random(1, 80))
  local user_id, password = get_user()
  local cust_name = user_id

  local num_room = "1"

  local method = "POST"
  local path = "http://localhost:5000/reservation?inDate=" .. in_date_str ..
    "&outDate=" .. out_date_str .. "&lat=" .. tostring(lat) .. "&lon=" .. tostring(lon) ..
    "&hotelId=" .. hotel_id .. "&customerName=" .. cust_name .. "&username=" .. user_id ..
    "&password=" .. password .. "&number=" .. num_room
  local headers = {}
  -- headers["Content-Type"] = "application/x-www-form-urlencoded"
  return wrk.format(method, path, headers, nil)
end

local function user_login()
  local user_name, password = get_user()
  local method = "GET"
  local path = "http://localhost:5000/user?username=" .. user_name .. "&password=" .. password
  local headers = {}
  -- headers["Content-Type"] = "application/x-www-form-urlencoded"
  return wrk.format(method, path, headers, nil)
end

request = function()
  cur_time = math.floor(socket.gettime())
  local search_ratio      = 0.6
  local recommend_ratio   = 0.39
  local user_ratio        = 0.005
  local reserve_ratio     = 0.005

  local coin = math.random()
  if coin < search_ratio then
    return search_hotel()
  elseif coin < search_ratio + recommend_ratio then
    return recommend()
  elseif coin < search_ratio + recommend_ratio + user_ratio then
    return user_login()
  else
    return reserve()
  end
end
```

Appendix D

Overview benchmark resources

This chapter includes the explicit resources used for the microservices in our testbed. The container images are available on dockerhub[1] and have multiple tags based on the orchestrator. The flags correspond where the :latest tag is specific to swarm, :kubernetes to kubernetes, and :nomad to nomad, e.g. stvdputten/openrest-thrift:nomad. The open-source images include redis:alpine1.13, dns-proxy-server:2.19.0, mongo:4.4.6, memcached:1.6.9, jaegertracing/all-in-one:1.23.0 and consul:1.9.6.

D.1 Social Network resources

Table D.1: Resource overview of the Media Microservices benchmark. (B) is the benchmark limits set for experiment 12, (V) has vertical scaling with 2x the resources for experiment 13, (H) has horizontal scaling with 2x the containers for experiment 14, (G) is the setup based on the open-source Github where NA means the deployment files does not exist at time or writing, (O) is the observed resources used. Memory is set using Gibibyte. Inf means the explicit resource limit is not set.

Microservices (social-network)	(B) Cores	(B) Memory	(V) Cores	(V) Memory	(H) Cores	(H) Memory	(G) Cores	(G) Memory	(O) Cores	(O) Memory
jaeger	NA	NA	NA	NA	NA	NA	inf	inf	1	0.1GiB
compose-post-service	1	1	2	2	2	2	1	1	1	0.1GiB
home-timeline-redis	1	1	2	2	2	2	inf	inf	1	0.2GiB
home-timeline-service	1	1	2	2	2	2	1	1	1	0.1GiB
media-frontend	1	1	2	2	2	2	1	1	1	0.1GiB
media-memcached	1	1	2	2	2	2	inf	inf	1	0.1GiB
media-mongodb	1	1	2	2	2	2	inf	inf	1	0.1GiB
media-service	1	1	2	2	2	2	1	1	1	0.1GiB
nginx-thrift	4	4	8	8	8	8	4	4	10	0.3GiB
post-storage-memcached	1	1	2	2	2	2	inf	inf	1	0.1GiB
post-storage-mongodb	1	1	2	2	2	2	inf	inf	1	0.2GiB
post-storage-service	1	1	2	2	2	2	1	1	16	0.1GiB
social-graph-mongodb	1	1	2	2	2	2	inf	inf	1	0.3GiB
social-graph-redis	1	1	2	2	2	2	inf	inf	1	0.1GiB
social-graph-service	1	1	2	2	2	2	1	1	1	0.1GiB
text-service	1	1	2	2	2	2	1	1	1	0.1GiB
unique-id-service	1	1	2	2	2	2	1	1	1	0.1GiB
url-shorten-memcached	1	1	2	2	2	2	inf	inf	1	0.1GiB
url-shorten-mongodb	1	1	2	2	2	2	inf	inf	1	0.2GiB
url-shorten-service	1	1	2	2	2	2	1	1	1	0.1GiB
user-memcached	1	1	2	2	2	2	inf	inf	1	0.1GiB
user-mention-service	1	1	2	2	2	2	1	1	1	0.1GiB
user-mongodb	1	1	2	2	2	2	inf	inf	1	0.2GiB
user-service	1	1	2	2	2	2	1	1	1	0.1GiB
user-timeline-mongodb	1	1	2	2	2	2	inf	inf	1	0.2GiB
user-timeline-redis	1	1	2	2	2	2	inf	inf	1	0.1GiB
user-timeline-service	1	1	2	2	2	2	1	1	1	0.1GiB

[1] https://hub.docker.com/repository/docker/stvdputten/

D.2 Media Microservices resources

Table D.2: Resource overview of the Media Microservices benchmark. (B) is the benchmark limits set for experiment 12, (V) has vertical scaling with 2x the resources for experiment 13, (H) has horizontal scaling with 2x the containers for experiment 14, (G) is the setup based on the open-source Github where NA means the deployment files does not exist at time or writing, (O) is the observed resources used. Memory is set using Gibibyte. Inf means the explicit resource limit is not set.

Microservices (media-microsvc)	(B) Cores	(B) Memory	(V) Cores	(V) Memory	(H) Cores	(H) Memory	(G) Cores	(G) Memory	(O) Cores	(O) Memory
jaeger	NA	NA	NA	NA	NA	NA	NA	NA	11	34GiB
cast-info-memcached	1	1	2	2	2	2	NA	NA	1	0.1GiB
cast-info-mongodb	1	1	2	2	2	2	NA	NA	1	0.2GiB
cast-info-service	1	1	2	2	2	2	NA	NA	1	0.1GiB
compose-review-memcached	1	1	2	2	2	2	NA	NA	4	0.1GiB
compose-review-service	1	1	2	2	2	2	NA	NA	15	0.1GiB
dns-media	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-id-memcached	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-id-mongodb	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-id-service	1	1	2	2	2	2	NA	NA	6	0.1GiB
movie-info-memcached	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-info-mongodb	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-info-service	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-review-mongodb	1	1	2	2	2	2	NA	NA	7	1GiB
movie-review-redis	1	1	2	2	2	2	NA	NA	1	0.1GiB
movie-review-service	1	1	2	2	2	2	NA	NA	4	0.1GiB
nginx-web-server	4	4	8	8	8	8	NA	NA	5	0.2GiB
plot-memcached	1	1	2	2	2	2	NA	NA	1	0.1GiB
plot-mongodb	1	1	2	2	2	2	NA	NA	1	0.1GiB
plot-service	1	1	2	2	2	2	NA	NA	1	0.1GiB
rating-redis	1	1	2	2	2	2	NA	NA	1	0.1GiB
rating-service	1	1	2	2	2	2	NA	NA	4	0.1GiB
review-storage-memcached	1	1	2	2	2	2	NA	NA	1	0.1GiB
review-storage-mongodb	1	1	2	2	2	2	NA	NA	2	1GiB
review-storage-service	1	1	2	2	2	2	NA	NA	2	0.1GiB
text-service	1	1	2	2	2	2	NA	NA	1	0.1GiB
unique-id-service	1	1	2	2	2	2	NA	NA	1	0.1GiB
user-mongodb	1	1	2	2	2	2	NA	NA	1	0.1GiB
user-review-mongodb	1	1	2	2	2	2	NA	NA	8	1GiB
user-review-redis	1	1	2	2	2	2	NA	NA	1	0.1GiB
user-review-service	1	1	2	2	2	2	NA	NA	5	0.1GiB
user-service	1	1	2	2	2	2	NA	NA	2	0.1GiB

D.3 Hotel Reservation resources

Table D.3: Resource overview of the Media Microservices benchmark. (B) is the benchmark limits set for experiment 12, (V) has vertical scaling with 2x the resources for experiment 13, (H) has horizontal scaling with 2x the containers for experiment 14, (G) is the setup based on the open-source Github where NA means the deployment files does not exist at time or writing, (O) is the observed resources used. Memory is set using Gibibyte. Inf means the explicit resource limit is not set.

Microservices (hotel-reservation)	(B) Cores	(B) Memory	(V) Cores	(V) Memory	(H) Cores	(H) Memory	(G) Cores	(G) Memory	(O) Cores	(O) Memory
jaeger	NA	NA	NA	NA	NA	NA	NA	NA	9	19GiB
consul	1	1	2	2	2	2	NA	NA	1	0.1GiB
frontend	4	4	8	8	8	8	NA	NA	11	0.1GiB
geo	1	1	2	2	2	2	NA	NA	3	0.1GiB
memcached-profile	1	1	2	2	2	2	NA	NA	1	0.1GiB
memcached-rate	1	1	2	2	2	2	NA	NA	1	0.1GiB
memcached-reserve	1	1	2	2	2	2	NA	NA	2	0.1GiB
mongodb-geo	1	1	2	2	2	2	NA	NA	1	0.1GiB
mongodb-profile	1	1	2	2	2	2	NA	NA	1	0.1GiB
mongodb-rate	1	1	2	2	2	2	NA	NA	1	0.1GiB
mongodb-recommendation	1	1	2	2	2	2	NA	NA	1	0.1GiB
mongodb-reservation	1	1	2	2	2	2	NA	NA	1	0.1GiB
mongodb-user	1	1	2	2	2	2	NA	NA	1	0.1GiB
profile	1	1	2	2	2	2	NA	NA	7	0.1GiB
rate	1	1	2	2	2	2	NA	NA	5	0.1GiB
recommendation	1	1	2	2	2	2	NA	NA	2	0.1GiB
reservation	1	1	2	2	2	2	NA	NA	8	0.1GiB
search	1	1	2	2	2	2	NA	NA	11	0.1GiB
user	1	1	2	2	2	2	NA	NA	1	0.1GiB

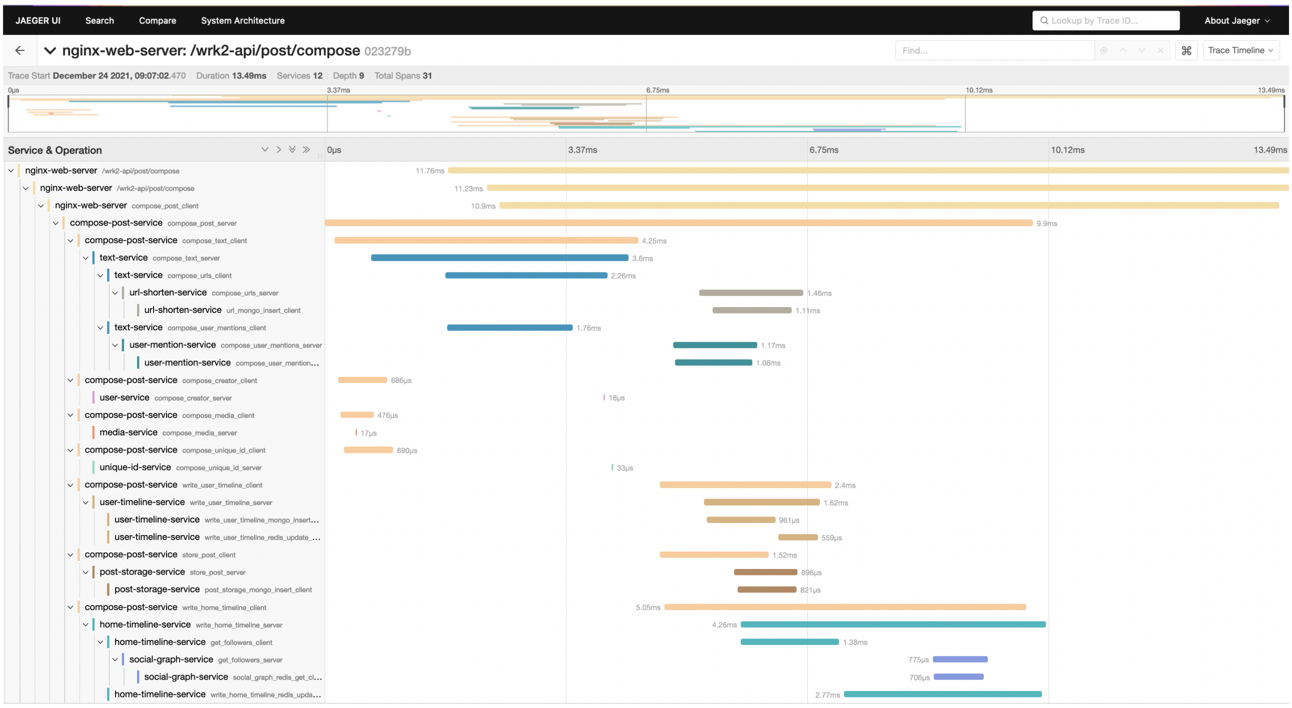
Appendix E

Jaeger

The following chapter includes figures of the Jaeger as observed during runs where we specifically observed breaking points of our applications during benchmarking. It includes the trace of the microservice services and can clearly show which of the service is the cause of the breaking points observed in Jaeger. This can help the reader understand how we used certain tools in the monitoring layer to help us experiment.

E.1 Tracing in Jaeger

Figure E.1, Figure E.2 and Figure E.3 illustrate the traces and the duration taken by each microservice. During the experiments, the load generator sends the HTTP request to API of the application. The URI call can be seen in the top left of the examples sn and mm ‘/compose‘ and hr ‘/hotels‘.



nginx-web-server: /wrk2-api/post/compose 023279b

Trace Start **December 24 2021, 09:07:02**.470 Duration **13.49ms** Services **12** Depth **9** Total Spans **31**

Service & Operation		
nginx-web-server	/wrk2-api/post/compose	11.76ms
nginx-web-server	/wrk2-api/post/compose	11.23ms
nginx-web-server	compose_post_client	10.9ms
compose-post-service	compose_post_server	9.9ms
compose-post-service	compose_text_client	4.25ms
text-service	compose_text_server	3.6ms
text-service	compose_urls_client	2.26ms
url-shorten-service	compose_urls_server	1.46ms
url-shorten-service	url_mongo_insert_client	1.11ms
text-service	compose_user_mentions_client	1.76ms
user-mention-service	compose_user_mentions_server	1.17ms
user-mention-service	compose_user_mention...	1.08ms
compose-post-service	compose_creator_client	686µs
user-service	compose_creator_server	18µs
compose-post-service	compose_media_client	476µs
media-service	compose_media_server	17µs
compose-post-service	compose_unique_id_client	690µs
unique-id-service	compose_unique_id_server	33µs
compose-post-service	write_user_timeline_client	2.4ms
user-timeline-service	write_user_timeline_server	1.62ms
user-timeline-service	write_user_timeline_mongo_insert...	961µs
user-timeline-service	write_user_timeline_redis_update_...	559µs
compose-post-service	store_post_client	1.52ms
post-storage-service	store_post_server	896µs
post-storage-service	post_storage_mongo_insert_client	821µs
compose-post-service	write_home_timeline_client	5.05ms
home-timeline-service	write_home_timeline_server	4.26ms
home-timeline-service	get_followers_client	1.38ms
social-graph-service	get_followers_server	775µs
social-graph-service	social_graph_redis_get_cl...	706µs
home-timeline-service	write_home_timeline_redis_upda...	2.77ms

Figure E.1: Latency overview of the Social Network application.

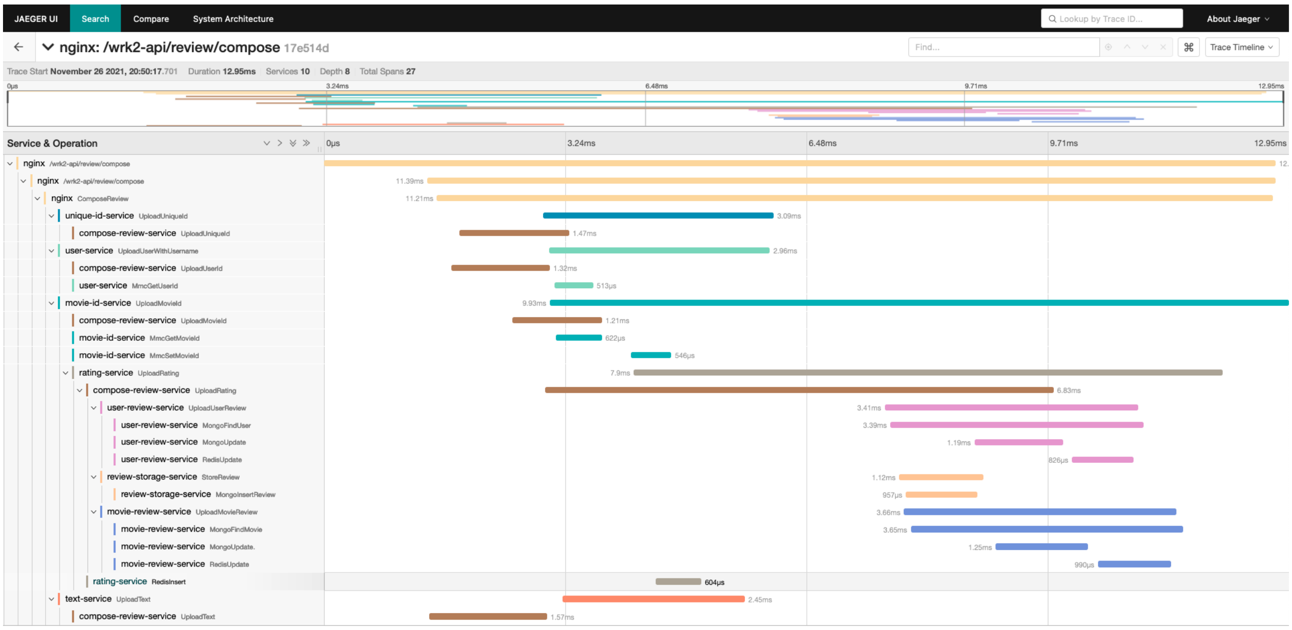

JAEGER UI
Search
Compare
System Architecture
Lookup by Trace ID...
About Jaeger
nginx: /wrk2-api/review/compose 17e514d
Find...
Trace Timeline
Trace Start November 26 2021, 20:50:17.701 | Duration 12.95ms | Services 10 | Depth 8 | Total Spans 27
0µs
3.24ms
6.48ms
9.71ms
12.95ms
Service & Operation
nginx /wrk2-api/review/compose 12.
nginx /wrk2-api/review/compose 11.39ms
nginx ComposeReview 11.21ms
unique-id-service UploadUniqueId 3.09ms
compose-review-service UploadUniqueId 1.47ms
user-service UploadUserWithUsername 2.96ms
compose-review-service UploadUserId 1.32ms
user-service MmcGetUserId 513µs
movie-id-service UploadMovieId 9.93ms
compose-review-service UploadMovieId 1.21ms
movie-id-service MmcGetMovieId 622µs
movie-id-service MmcSetMovieId 546µs
rating-service UploadRating 7.9ms
compose-review-service UploadRating 6.83ms
user-review-service UploadUserReview 3.41ms
user-review-service MongoFindUser 3.39ms
user-review-service MongoUpdate 1.19ms
user-review-service RedisUpdate 826µs
review-storage-service StoreReview 1.12ms
review-storage-service MongoInsertReview 957µs
movie-review-service UploadMovieReview 3.66ms
movie-review-service MongoFindMovie 3.65ms
movie-review-service MongoUpdate. 1.25ms
movie-review-service RedisUpdate 990µs
rating-service RedisInsert 604µs
text-service UploadText 2.45ms
compose-review-service UploadText 1.57ms

Figure E.2: Latency overview of the Media Microservices app.

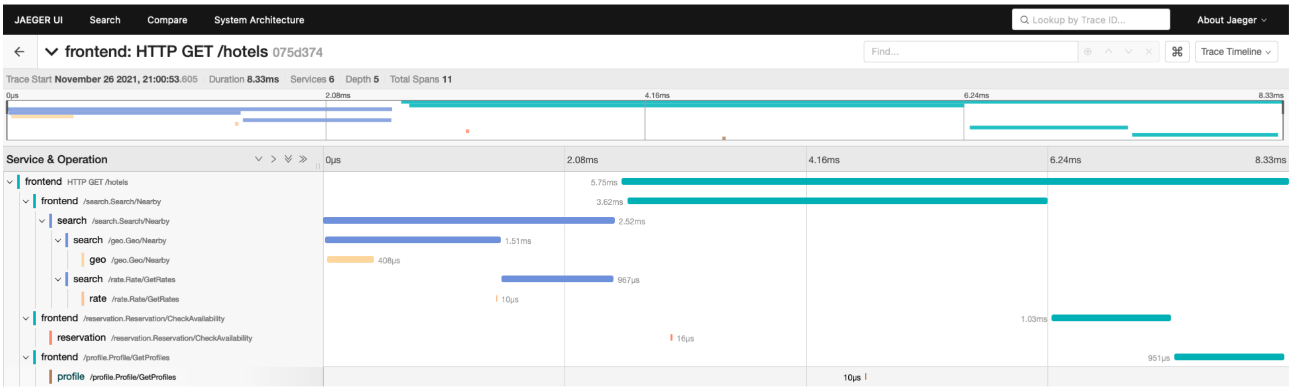
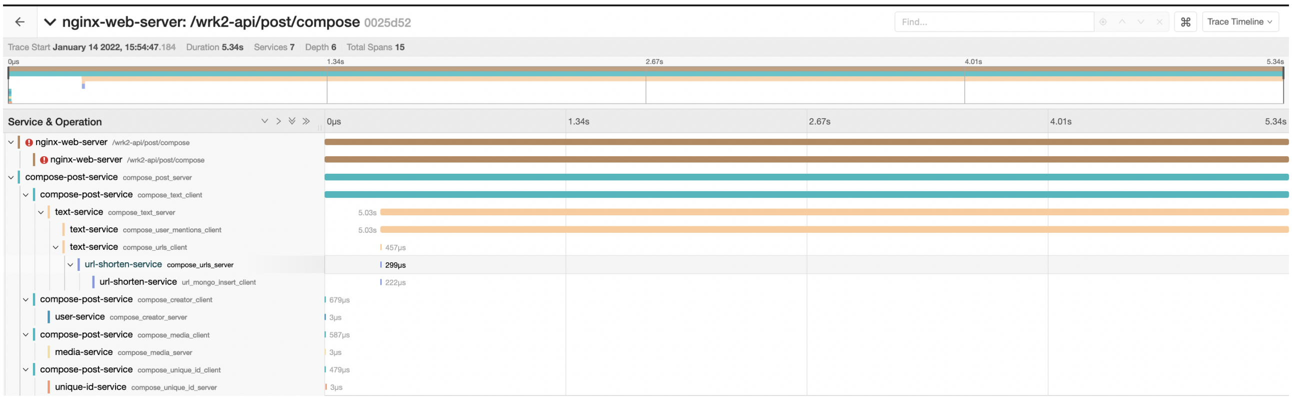

JAEGER UI
Search
Compare
System Architecture
Lookup by Trace ID...
About Jaeger
frontend: HTTP GET /hotels 075d374
Find...
Trace Timeline
Trace Start November 26 2021, 21:00:53.605 | Duration 8.33ms | Services 6 | Depth 5 | Total Spans 11
0µs
2.08ms
4.16ms
6.24ms
8.33ms
Service & Operation
frontend HTTP GET /hotels 5.75ms
frontend /search.Search/Nearby 3.62ms
search /search.Search/Nearby 2.52ms
search /geo.Geo/Nearby 1.51ms
geo /geo.Geo/Nearby 408µs
search /rate.Rate/GetRates 967µs
rate /rate.Rate/GetRates 10µs
frontend /reservation.Reservation/CheckAvailability 1.03ms
reservation /reservation.Reservation/CheckAvailability 16µs
frontend /profile.Profile/GetProfiles 951µs
profile /profile.Profile/GetProfiles 10µs

Figure E.3: Biggest span and latency overview of the Hotel Reservation app.

E.2 Breaking point in Jaeger

The breaking point of the benchmarked applications can be observed through the latency of the load generator or confirmed by monitoring the Jaeger traces. Figure E.4, Figure E.5 and Figure E.6 show examples of microservice failure. Specifically, the time spent at a microservice and the red exclamation mark which is shown when an error event has occurred.

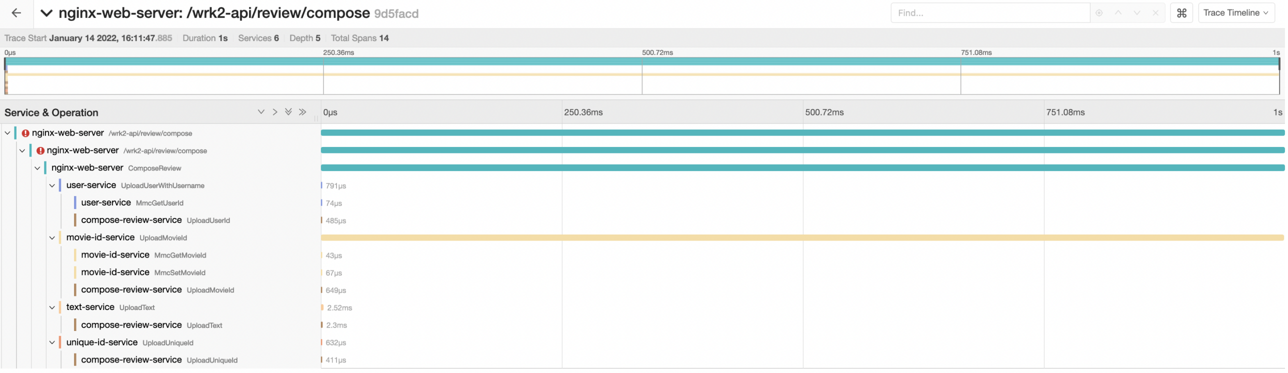


nginx-web-server: /wrk2-api/post/compose 0025d52

Trace Start **January 14 2022, 15:54:47**.184 | Duration **5.34s** | Services **7** | Depth **6** | Total Spans **15**

Service & Operation	
nginx-web-server /wrk2-api/post/compose	
nginx-web-server /wrk2-api/post/compose	
compose-post-service compose_post_server	
compose-post-service compose_text_client	
text-service compose_text_server	5.03s
text-service compose_user_mentions_client	5.03s
text-service compose_urls_client	457µs
url-shorten-service compose_urls_server	299µs
url-shorten-service url_mongo_insert_client	222µs
compose-post-service compose_creator_client	679µs
user-service compose_creator_server	3µs
compose-post-service compose_media_client	587µs
media-service compose_media_server	3µs
compose-post-service compose_unique_id_client	479µs
unique-id-service compose_unique_id_server	3µs

Figure E.4: The Nginx service returns an error. Tracing the latency and breaking point to the text-service for the Social Network benchmark.

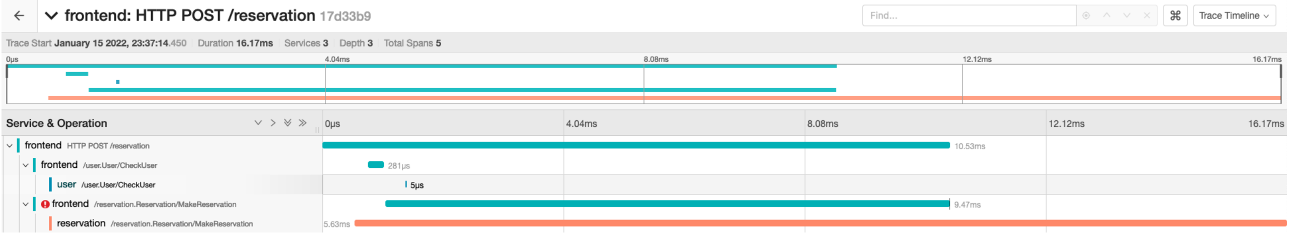


nginx-web-server: /wrk2-api/review/compose 9d5facd

Trace Start **January 14 2022, 16:11:47**.885 | Duration **1s** | Services **6** | Depth **5** | Total Spans **14**

Service & Operation	
nginx-web-server /wrk2-api/review/compose	
nginx-web-server /wrk2-api/review/compose	
nginx-web-server ComposeReview	
user-service UploadUserWithUsername	791µs
user-service MmcGetUserId	74µs
compose-review-service UploadUserId	485µs
movie-id-service UploadMovieId	
movie-id-service MmcGetMovieId	43µs
movie-id-service MmcSetMovieId	67µs
compose-review-service UploadMovieId	649µs
text-service UploadText	2.52ms
compose-review-service UploadText	2.3ms
unique-id-service UploadUniqueId	632µs
compose-review-service UploadUniqueId	411µs

Figure E.5: The Nginx service returns an error. Tracing the latency and breaking point to the movie-id-service for the Media Microservices benchmark.



frontend: HTTP POST /reservation 17d33b9

Trace Start **January 15 2022, 23:37:14**.450 | Duration **16.17ms** | Services **3** | Depth **3** | Total Spans **5**

Service & Operation	
frontend HTTP POST /reservation	10.53ms
frontend /user.User/CheckUser	281µs
user /user.User/CheckUser	5µs
frontend /reservation.Reservation/MakeReservation	9.47ms
reservation /reservation.Reservation/MakeReservation	5.63ms

Figure E.6: The front end service returns an error. Tracing the latency and breaking point to the reservation-service for the Hotel Reservation benchmark.

Appendix F

Experiment and benchmark setup files

Includes examples lines from the scripts used to run the experiments in the experiment layer. Environment variables are used to determine the parameters for the experiments. Furthermore, deployments files are shown for Kubernetes, Docker Swarm and Nomad as examples of our files.

F.1 Experiment bash script setup

```
# experiment params
export availability=0
export unlimited=1
export horizontal=1
export vertical=1

for benchmark in socialNetwork mediaMicroservices hotelReservation; do
  echo "Running the baseline tests stress $experiment for $benchmark"
  export benchmark=$benchmark
  for requests in 500 1500 2000 3000 4000 5000 10000 15000 20000; do
    for connections in 512; do
      for threads in 4; do
        ./setup-experiments.sh -t $threads -c $connections -d 30 -R  $requests
      done
    done
  done
done
```

F.2 Docker Swarm Social Network

```
  nginx-thrift:
    deploy:
      resources:
          limits:
            cpus: '4.0'
            memory: 4GiB
      replicas: 1
      placement:
        constraints:
          - node.role==worker
      restart_policy:
        condition: any
    image: stvdputten/openresty-thrift:latest
    hostname: nginx-thrift
    ports:
      - 8080:8080
    depends_on:
            - jaeger
    volumes:
      - ./nginx-web-server/lua-scripts:/usr/local/openresty/nginx/lua-scripts
      - ./nginx-web-server/pages:/usr/local/openresty/nginx/pages
      - ./nginx-web-server/conf/nginx.conf:/usr/local/openresty/nginx/conf/nginx.conf
      - ./nginx-web-server/jaeger-config.json:/usr/local/openresty/nginx/jaeger-config.json
      - ./gen-lua:/gen-lua
      - ./docker/openresty-thrift/lua-thrift:/usr/local/openresty/lualib/thrift
      - ./keys:/keys
```

F.3 Kubernetes Hotel Reservation

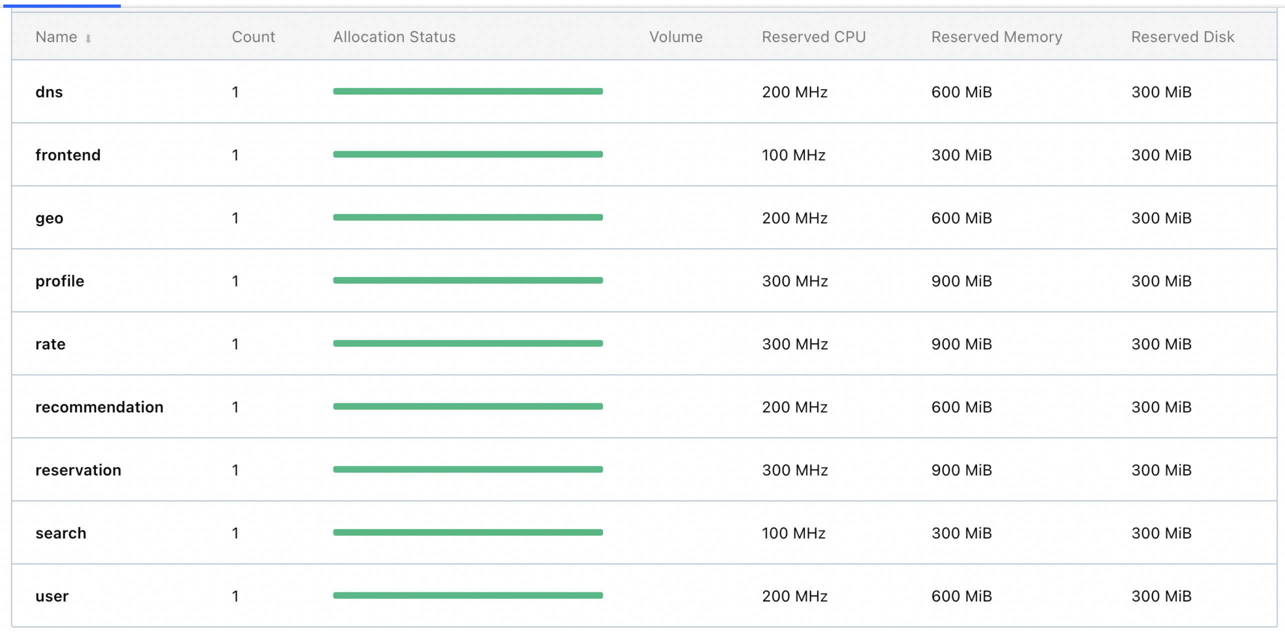
```
    spec:
      containers:
        - command:
            - frontend
          image: stvdputten/hotel_reserv_frontend_single_node:kubernetes
          name: frontend
          ports:
            - containerPort: 5000
          resources:
```

```
          limits:
            cpu: "4.0"
            memory: "4Gi"
        volumeMounts:
        - mountPath: /go/src/github.com/harlow/go-micro-services/config.json
          subPath: config.json
          name: config-json
    restartPolicy: Always
    volumes :
    - name: config-json
      configMap:
        name: configmap-config-json
        items:
        - key: config.json
          path: config.json
```

F.4 Nomad Media Microservices

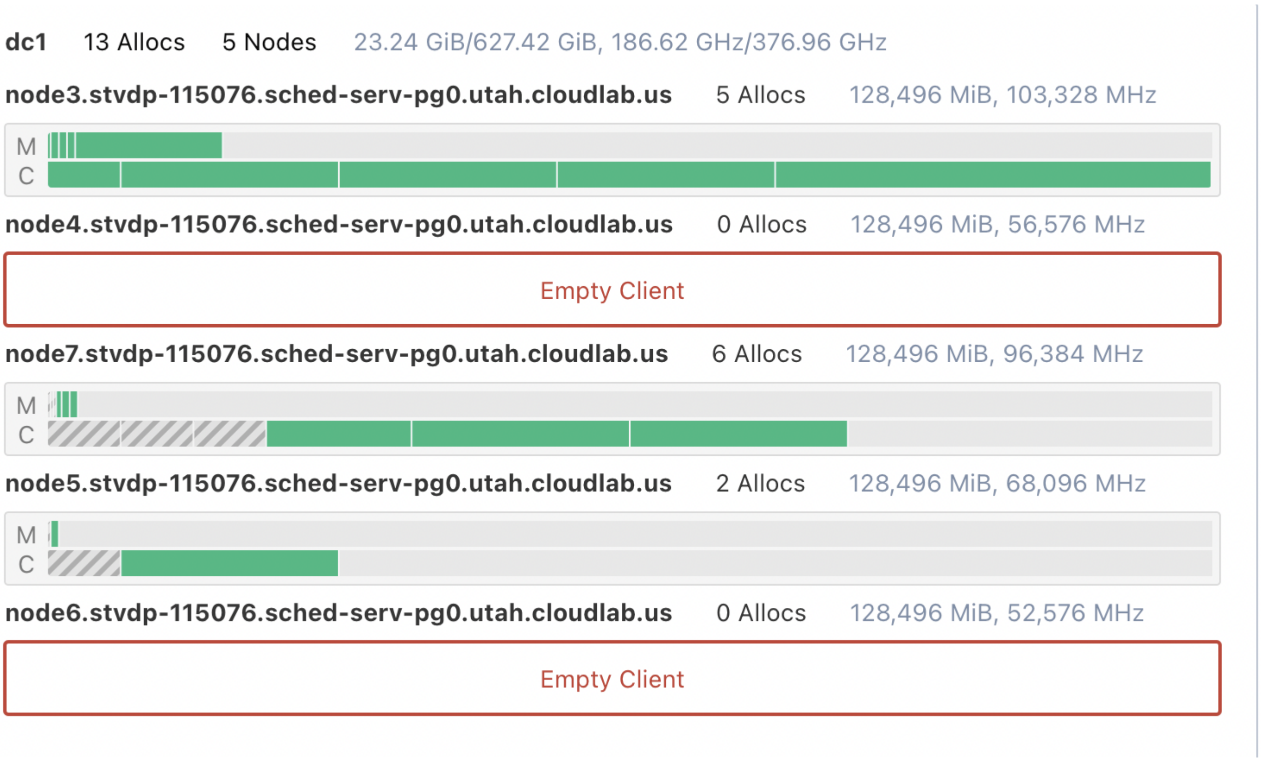
```
group "nginx-web-server" {
  count = 1
  constraint {
    attribute = "${attr.unique.hostname}"
    value     = "${var.hostname}"
  }
  network {
    mode = "bridge"
    port "nginx" {
      static = 8080
    }
    port "jaeger-ui" {
      static = 16686
    }
    port "jaeger" {
      static = 6831
    }
    dns {
      servers  = ["${var.dns}", "8.8.8.8"]
      searches = ["service.consul"]
    }
    }

    task "nginx-web-server" {
      driver = "docker"

      resources {
        cpu = 4000
        memory_max = 4000
      }

      config {
        memory_hard_limit = 4000
        cpu_hard_limit = true

        image   = "yg397/openresty-thrift:xenial"
        ports   = ["nginx"]
        command = "sh"
        args    = ["-c", "echo '127.0.0.1 jaeger.service.consul' >> /etc/hosts && echo '127.0.0.1 jaege
        mount {
          type   = "bind"
          target = "/usr/local/openresty/nginx/lua-scripts"
          source = "/users/stvdp/DeathStarBench/mediaMicroservices/nomad/lua-scripts"
        }
        mount {
          type   = "bind"
          target = "/usr/local/openresty/nginx/conf/nginx.conf"
          source = "/users/stvdp/DeathStarBench/mediaMicroservices/nomad/nginx.conf"
        }
        mount {
          type   = "bind"
          target = "/usr/local/openresty/nginx/jaeger-config.json"
          source = "/users/stvdp/DeathStarBench/mediaMicroservices/nomad/jaeger-config.json"
        }
        mount {
          type   = "bind"
          target = "/gen-lua"
          source = "/users/stvdp/DeathStarBench/mediaMicroservices/nomad/gen-lua"
```

F.5 Nomad Examples GUI



Name	Count	Allocation Status	Volume	Reserved CPU	Reserved Memory	Reserved Disk
dns	1			200 MHz	600 MiB	300 MiB
frontend	1			100 MHz	300 MiB	300 MiB
geo	1			200 MHz	600 MiB	300 MiB
profile	1			300 MHz	900 MiB	300 MiB
rate	1			300 MHz	900 MiB	300 MiB
recommendation	1			200 MHz	600 MiB	300 MiB
reservation	1			300 MHz	900 MiB	300 MiB
search	1			100 MHz	300 MiB	300 MiB
user	1			200 MHz	600 MiB	300 MiB

Figure F.1: Overviw of the Hotel Reservation benchmark on Nomad



dc1 13 Allocs 5 Nodes 23.24 GiB/627.42 GiB, 186.62 GHz/376.96 GHz

node3.stvdp-115076.sched-serv-pg0.utah.cloudlab.us 5 Allocs 128,496 MiB, 103,328 MHz

node4.stvdp-115076.sched-serv-pg0.utah.cloudlab.us 0 Allocs 128,496 MiB, 56,576 MHz

Empty Client

node7.stvdp-115076.sched-serv-pg0.utah.cloudlab.us 6 Allocs 128,496 MiB, 96,384 MHz

node5.stvdp-115076.sched-serv-pg0.utah.cloudlab.us 2 Allocs 128,496 MiB, 68,096 MHz

node6.stvdp-115076.sched-serv-pg0.utah.cloudlab.us 0 Allocs 128,496 MiB, 52,576 MHz

Empty Client

Figure F.2: Topology of the containers from the Nomad GUI