Analysing the Effects of Scalability in Microservices to User-Perceived Performance and Cloud Costs

University of Turku
Department of Computing
Master of Science (Tech) Thesis
Software Engineering
December 2025
Kaarle Järvinen

The originality of this thesis has been checked in accordance with the University of Turku quality assurance system using the Turnitin OriginalityCheck service.

UNIVERSITY OF TURKU
Department of Computing

Kaarle Järvinen: Analysing the Effects of Scalability in Microservices to User-Perceived Performance and Cloud Costs

Master of Science (Tech) Thesis, 102 p., 2 app. p.
Software Engineering
December 2025

Microservices are widely adopted in cloud computing systems, where scalability is essential for maintaining performance and controlling operational costs under rapidly changing workloads. While cloud computing platforms provide elastic infrastructure, the ability of microservices to benefit from elasticity depends on their scalability. This thesis analyses how various levels of scalability in microservices affect user-perceived performance and cloud costs, and identifies operational characteristics that drive scalability.

This thesis adopts an empirical and quantitative methodology combining a conceptual analysis of microservice implementation technologies, a structured literature review and controlled empirical experiments. Microservice benchmark applications with various levels of scalability are compared under simulated workloads to compare latency and resource usage. Java-based microservice frameworks are used to compare the implications of scalability in a common runtime ecosystem, while the analysis remains framework and language agnostic.

The results show that scalability in microservices is driven by application startup time, container image size, resource efficiency and request throughput. Poor scalability manifests as increased tail-latency, latency spikes during scale-out and unpredictable response times, thus degrading overall user experience. In contrast, improved scalability enables high resource-efficiency, reduces the need for resource over-provisioning and thus leads to lower cloud computing costs.

Keywords: Microservices, Scalability, Cloud Computing, User-Perceived Performance, Tail Latency, containerisation, Cloud Costs, Elasticity, Kubernetes

AI assistance declaration

Artificial intelligence (AI) tools were used in this thesis in a limited capacity. AI was used as a programming assistance tool during the development of the benchmark applications used in the empirical experiments. See Appendix A for more. The research design, methodology, analysis, interpretation of results and all written content of the thesis were all produced by the author.

Contents

1 Introduction **1**

1.1 Research Questions 2

1.2 Methodology 3

1.3 Research Structure 4

2 Background **6**

2.1 Cloud Computing 6

2.2 Microservices Architecture 13

2.3 Container Technologies 20

2.4 Scalability 24

3 Java-based Microservice Frameworks **30**

3.1 Java's Role for Scalable Microservices 30

3.2 Spring Boot 32

3.3 Quarkus 37

3.4 Micronaut 39

3.5 Differences in Operational Characteristics 41

4 Literature review **43**

4.1 Methodology 43

4.2 Literature Review Results 52

4.3 Discussion of Results 60

5 Scalability Experiments 65

5.1 Methodology 65

5.2 Experiment E1 74

5.3 Experiment E2 79

5.4 Experiment E3 86

6 Discussion 94

6.1 Operational Characteristics Driving Scalability (RQ1) 94

6.2 Impact of Scalability on User-Perceived Performance (RQ2) 96

6.3 Impact of Scalability on Cloud Costs and SLA Adherence (RQ3) . . . 97

6.4 Limitations 98

6.5 Future Work 99

7 Conclusion 101

References 103

Appendices

A The Use of AI A-1

B Empirical Experiments Source Code B-1

1 Introduction

Software systems have been traditionally implemented with the monolithic architecture, where all application logic and responsibilities are bundled into a single deployable unit with a single codebase. Monoliths are simple and easy to start with, but as the system grows they become increasingly complex and rigid. A minor change in the application logic requires rebuilding and redeploying of the whole application. [1] Teams developing separate features might be blocking each other and scaling the application horizontally means cloning the whole system instead of only just the components under heavy use, leading to resource waste. Upgrading technologies means upgrading the whole application and a single fault in one module can cascade to the whole system.

To address these limitations, modern software systems are increasingly implemented with or migrated to a microservices architecture. Systems are decomposed into microservices that run in independent, lightweight containers and are orchestrated by platforms such as Kubernetes. [2] [3] However orchestrating a microservices deployment with several services might be complex and the initial costs might surpass the costs of a monolithic deployment. A poorly configured microservices deployment with bad elasticity can be outperformed by a monolithic deployment in user-perceived performance and cloud costs. Modern autoscaling solutions address these limitations by scaling resources down when they are not needed and scaling them up to match demand when needed. [4] [5] [6] However the effectiveness of

autoscaling algorithms are affected by the scalability of individual services, thus optimising the scalability of individual services potentially has benefits to the elasticity of the whole microservices deployment [5] [6]. This thesis aims to research the implications of an individual service's scalability to a whole microservices deployment.

Java remains the most dominant programming language in the Java Virtual Machine (JVM) ecosystem and it is especially popular in enterprise systems where enterprises have invested heavily in Java-specific tooling. The Java ecosystem is mature and offers various development frameworks for microservices such as Spring Boot, Quarkus and Micronaut that offer fast release cycles, cloud-native compilation and elastic scaling [7]. These frameworks have differing design goals, offer various levels of scalability, and thus they act as good candidates for researching the implications of scalability.

1.1 Research Questions

This thesis aims to answer three research questions (RQs) centred around the scalability of microservices. The first question aims to research the drivers of scalability. The second question focuses on scalability from the end user's point of view. The third question focuses on scalability from the business perspective.

RQ1: What kind of operational characteristics drive scalability in containerised cloud deployments? This question maps the different operational characteristics of microservice applications e.g. image sizes, startup time, throughput, and how they affect scalability and elasticity.

RQ2: How different levels of scalability affect user-perceived performance in microservices under heavy load? This question focuses on the different observable user-centred metrics and how they are affected by systems under heavy

load. Performance metrics in this context mean latency percentiles of requests.

RQ3: How different levels of scalability affect the total cost of ownership of deployed microservices? This question connects system scaling metrics to business outcomes, such as total economic cost and Service Level Agreement (SLA) adherence.

1.2 Methodology

This thesis adopts an empirical and quantitative research approach to analyse the effects of scalability in microservices to user-perceived performance and cloud costs. The research is conducted within a positivist paradigm, which allows the investigation of scalability related concepts and trends under a controlled environment. The purpose of the thesis is evaluative, since it aims to evaluate how different scalability characteristics and optimisations influence user-perceived performance and cloud costs instead of proposing new frameworks or implementation techniques.

The research methodology consists of three components: a conceptual analysis, a structured literature review and controlled empirical experiments. The conceptual analysis consists of a non-empirical examination of microservices frameworks based on their documentation and design choices and is used to identify operational characteristics relevant to scalability. The literature review examines existing research on scalability concepts, performance metrics and cost implications in cloud-native systems. The conceptual analysis and literature review direct the design of the empirical experiments and provide a foundation for interpreting the experimental results.

The empirical research of this thesis is based on three controlled experiments conducted in a container-orchestrated environment. Microservices applications with various levels of scalability are deployed and evaluated under simulated workload.

User-perceived performance is compared through latency metrics and resource usage and absolute costs are compared in a cloud environment. For systematic comparison, JVM-based microservice frameworks are used within a common runtime ecosystem. However, the analysis and conclusions focus on general scalability characteristics and trends instead of focusing on technology specific behaviour.

The scope of this thesis is constrained to enable controlled and repeatable experimentation. While the use of a single runtime ecosystem limits the generalisability of the results, this approach allows for direct observation of scalability related effects and systematic comparison. Therefore the findings of this thesis are interpreted as indicative of general trends and implications of scalability, rather than as absolute guarantees for all microservices deployments.

1.3 Research Structure

This thesis is organised into several chapters that start from the conceptual background to empirical experimentation and finally to analysis and conclusions of the research.

Chapter 2 provides the conceptual background of the thesis. It introduces the microservices architecture, its historical context and the primary design principles that enable independent deployment and scalability. The chapter also discusses cloud computing, container technologies and defines scalability related concepts relevant to the thesis.

Chapter 3 includes the conceptual analysis of three widely used Java-based microservice frameworks: Spring Boot, Quarkus and Micronaut, which all have various levels of scalability. The section examines their runtime models, build processes and containerisation features in order to identify operational characteristics that the frameworks aim to optimise for improved scalability.

Chapter 4 examines existing literature related to scalability in cloud environ-

ments. The section aims to investigate the drivers of scalability and how scalability affects user-perceived performance and cloud costs.

Chapter 5 describes the methodology and design of the empirical experiments. It presents the benchmark applications, the deployment environments and measurement tools, and presents the results of the experiment.

Chapter 6 discusses and synthesises the results of the literature review and the empirical experiments. The section analyses the findings against the three research questions and examines the implications of scalability to user-perceived performance and cloud costs. Additionally, the section discusses the limitations of the thesis and proposes directions for future research.

Finally, Chapter 7 concludes the thesis by summarising the answers to the research questions and highlights the main contributions.

2 Background

This chapter provides the conceptual background for the thesis and the research problem. It introduces the microservices architecture, the cloud computing paradigm, the container technologies that enable it and concepts related to scalability and elasticity. The chapter also defines performance and cost terminology used across the thesis. The purpose of this chapter is to establish a conceptual foundation for the analysis of scalability and to clarify how scalability is understood in the context of this thesis.

2.1 Cloud Computing

2.1.1 Cloud Definition

Cloud computing has enabled a great shift in the Information Technology industry, specifically in how computing resources are delivered and used. Defined by the National Institute of Standards and Technology (NIST): "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [8]. Adopting cloud computing allows businesses to flexibly acquire scalable resources without extensive initial investments. This fundamentally changes the cost dynamics and accessibility of computing re-

sources.

Although the idea of cloud computing can be traced back to the concept of computer resource sharing first discussed in the 1960s, the birth of cloud computing as we see it today dates to the early 2000s, when the dot-com bubble burst. At that time businesses were looking for cost effective solutions to mitigate excess computing resources, which were generating unnecessary costs. The year 2002 marked a critical point, when Amazon Web Services (AWS) was founded. AWS demonstrated the practicality and commercial viability of cloud computing and effectively started the widespread trend of businesses migrating into using cloud computing, which we still see today. [9]

The primary motivation for adopting cloud computing is its economic efficiency through one of the fundamental aspects of cloud computing; cloud elasticity. It allows the effective utilisation of idle computing resources, which leads to significantly better resource utilisation rates and reduced unnecessary energy consumption. Additionally cloud computing inherently comes with centralisation and virtualisation of resources and services, which in turn lead to enhanced reliability and reduced complexity. Thus cloud computing services facilitate more effective and secure management of computing resources compared to traditional on-premise data centres. [8] [10]

Crucial to its widespread adoption cloud computing introduces several inherent attributes which make it beneficial to businesses. These are resource pooling, rapid elasticity, broad network access, measured service and on-demand self-service [8]. These characteristics allow for businesses to scale in response to varying demand, which leads to greater operational agility and advantage over on-premise services.

2.1.2 Cloud Service and Deployment Models

NIST defines three service models and four deployment models that model and set constraints on how cloud resources are delivered, distributed and managed. These models determine the amount of control over capacity, automation and performance and thus determine where scalability can be managed.[8]

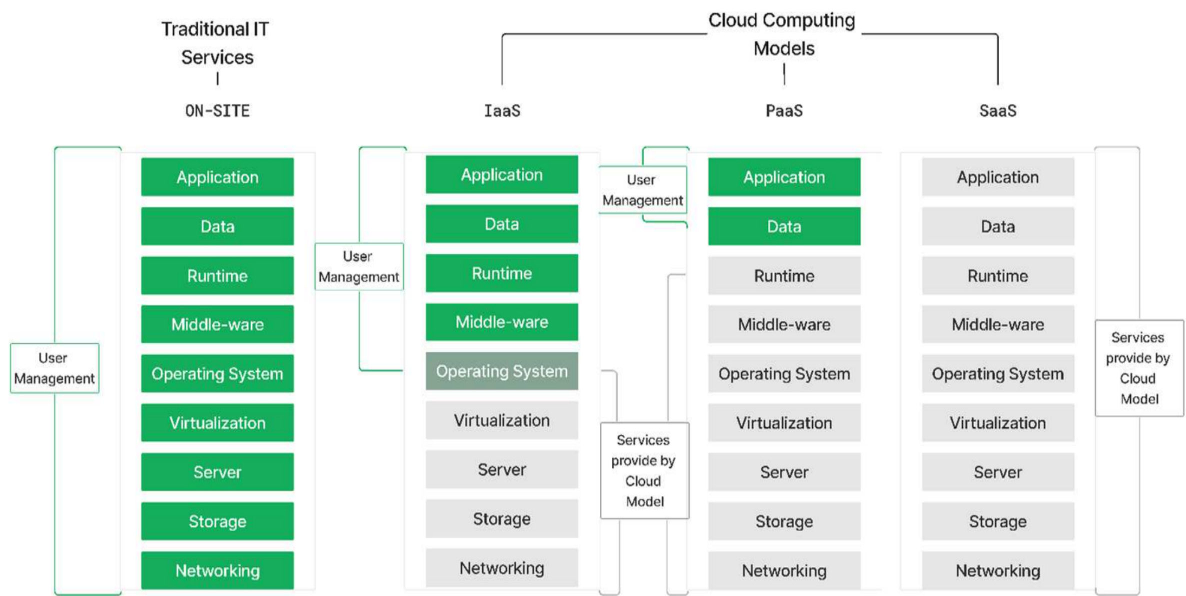


Figure 2.1: A figure of service models from Younis et al. [11].

The service models are abstraction layers that describe how much control the Cloud Service User (CSU) has over infrastructure and how much the platform controls as Figure 2.1 illustrates. Infrastructure as a Service (IaaS) gives the most control over operating systems, storage and networking. It is a perfect model for CSUs that require fine-tuning of autoscaling strategies, selecting optimized instance types or mixing containers and virtual machines in order to optimize latency and throughput. Platform as a Service (PaaS) abstracts infrastructure management and offers a managed runtime with build and deploy tooling and ready-made autoscaling. It makes delivery fast, but reduces the control over infrastructure. It is perfect when the runtime constraints of a cloud provider are enough. Software as a Service (SaaS)

places infrastructure and runtime control with the Cloud Service Provider (CSP). The CSU can then configure scaling between the boundaries set by the CSP. This makes SaaS suitable for CSUs that require focus on the application functionality and not in fine-tuned control over performance. [8] [12] [11] [13]

Table 2.1: A table of deployment models from Saraswat and Tripathi [13].

Attributes	**Public**	**Private**	**Community**	**Hybrid**
Scope of Service	Open for General Public and Large Industrial group: system and services are easily accessible to general public on demand	Open for licensed users, single organization: services & accessibility of cloud infrastructure is exclusively available within organization or owned persons	Open for community users that have shared concerns (mission, security, policy etc.)	Open for general public & licensed users
Owned by	Always Third Party i.e CSPs	Single organization/lessened users	Several organizations	Organization and Third Party(CSPs)
Size of Data Centre	~50,000 Server	~5,000 Server	15000 (Depends on number of orgns)	Less than public but more than private
Security	Low: because of it openness i.e E-mail	Very High: because of its private nature	High	Medium
Location	Off-Premise	off or on Premise	off or on Premise	Off and on Premise
Managed by	Only Third Party(CSPs)	Single Organization or CSPs	Several Organizations and CSPs	Both (organization and CSPs)
Cost of Implementation	No-Initial Cost	High-Initial Cost	(Depends on number of organizations)	Medium

Deployment models describe how the cloud infrastructure is provisioned and who manages the physical infrastructure. Public clouds offer elastic capacity and

measured service, which supports rapid scale out and scale in operations. This is usually the best option for horizontal elasticity, since public clouds leverage the computing potential of numerous data centres of cloud providers. Private clouds emphasize control, physical location of data and security policies. They can scale, but the limit is set by the physically owned computing capacity. Community clouds are like private clouds, but the infrastructure is shared among organisations with similar regulatory compliance requirements. Hybrid clouds combine the properties of both public and private cloud, by leveraging the elasticity and geographical reach of public cloud and keeping sensitive operations in private cloud. [8] [11] [13]

When designing a highly elastic system, choosing the right service model and deployment model is crucial. If custom autoscaling configurations or container orchestrations is needed, IaaS is the best option. If delivery speed and managed autoscaling are important, then PaaS is a good option. For deployment, public or hybrid models are best option if demand surges are expected. Private deployment is suitable, when regulatory compliance is needed, but the constraints on elasticity need to be accepted. [9]

2.1.3 Cloud Elasticity

As discussed in Section 2.1.1, elasticity is central in cloud computing. Elasticity is the ability to dynamically provision and release computing resources based on varying demand [8]. The goal of elasticity is to scale resources rapidly up or down to match demand as accurately as possible, thus maximizing resource utilisation and cost-efficiency [4] [5] [6]. Elasticity can be further divided into horizontal and vertical elasticity. Horizontal elasticity means adding or removing computing instances, such as virtual machines (VMs) or containers. Vertical elasticity means adjusting the resources inside computing instances, such as CPU, memory or storage capacity. [6]

Container based virtualization and containers have significantly improved cloud

elasticity, due to the lightweight nature of the technology. They scale quicker compared to traditional VMs and consume less resources. Containers, such as Docker containers, can also be managed by a container orchestrator such as Kubernetes. Kubernetes enables elasticity of container based cloud services through rapid deployments, minimal overhead and quick start up times. Central mechanisms to Kubernetes are the horizontal- and vertical pod autoscalers, that manage scaling decisions proactively and reactively based on metrics, such as CPU and memory usage. [14] [15]

While cloud elasticity promises powerful advantages, it comes with the complexity of configuration and potential issues in provisioning. Improper configurations can lead to over-provisioning resources, which hinders cost-efficiency or under-provisioning, which hinders the availability of services. In addition to configurations and autoscaling policies, the speed of provisioning is also affected by "cold starts". They occur when new instances are provisioned and initialized, and the longer they take a longer a system is over- or under-provisioned. [4] [5] [6]

2.1.4 Service Level Agreements

Service Level Agreements (SLAs) are formal agreements between cloud service providers (CSPs) and cloud service users (CSUs). They specify service level objectives (SLOs) for latency, throughput, availability, reliability and even cost and security, which are then tracked with appropriate service level indicators (SLIs). [16] SLAs are enforceable agreements with penalties or credits for violations. They should be designed around concrete SLIs that CSPs and CSUs both can observe. Typical SLIs include service uptime, response time, throughput and error rates with explicit thresholds and evaluation intervals. [17]

The SLA lifecycle is very important. A good process includes discovery and selection of a CSP, negotiation of an SLA, execution and monitoring. Negotiation

and monitoring are essential to modern SLA processes. Automated protocols and brokers align CSU requirements with CSP capabilities and can initiate new negotiations when conditions change. [16] [17] SLAs require continuous monitoring in order to be enforced. Without it, violations cannot be detected or remedied. [17]

For CSUs with high elasticity requirements, SLAs can acknowledge elasticity explicitly. CSUs care about the effect of scaling on objectives and not about the internal mechanisms. An elasticity aware SLO could define that a response must be under a second except for in situations where a surge in demand has been detected a response can take longer. After a grace period, the original objective takes place. This SLO pattern is aware of elasticity and acknowledges the realities of over- and under-provisioning. [18]

For an SLA to be relevant and effective, it must include a small set of SLIs relevant to the CSUs. How often they are tracked and evaluated. How violations are detected and what kind of penalties and remedies follow. What kind of changes initiate re-negotiations. And finally monitoring the relevant SLIs is important, since without it the SLA is useless. [16] [17]

2.1.5 Total Cost of Ownership

Total Cost of Ownership (TCO) is the full economical cost of provisioning, operating, developing and disposing of a cloud infrastructure throughout its lifecycle. TCO encompasses the direct costs of the cloud platform and the indirect costs related to operating infrastructure in the cloud, such as migrating to cloud, modernizing infrastructure, right sizing resources, observability, support and development required to maintain the deployed software. It is different from Return on Investment (ROI) as it does not account benefits. TCO can be used to compare the different cost implications of scalability choices and it supports design choices when designing a scalable and cost-effective cloud architecture. [19] [20]

$$TCO = Upfront\ Costs + Recurring\ Costs + Termination\ Costs \tag{2.1}$$

$$Tangible\ Benefits = Incremental\ Revenues + Lower\ Costs \tag{2.2}$$

$$ROI = \frac{(Tangible\ Benefits + Intangible\ Benefits) - TCO}{TCO} \tag{2.3}$$

Equations 2.1, 2.2 and 2.3 illustrate how cloud TCO and ROI are calculated and how recurring costs affect them[19]. In equation 2.1, the upfront costs and termination costs are fixed variables that depend on the chosen CSP. However the recurring costs depends on the cloud resources used and varying demand. In Equation 2.2 we see that lower costs lead to better tangible benefits and that in turn leads to better ROI in Equation 2.3. The goal of elasticity and scalability is to lower recurring costs as much as possible while keeping SLOs intact, thus minimizing TCO [20] and leading to better ROI.

2.2 Microservices Architecture

2.2.1 History and Core Principles

Microservices Architecture (MSA) is an architectural style where software systems are decomposed into small, cohesive and independently deployable services that communicate through lightweight mechanisms, typically HTTP APIs [21]. The architectural style is fundamentally different from the traditional monolithic architecture, where a software system is mostly composed of a single deployable system encompassing all the features of the software system. This fundamental difference makes microservices modular, independently scalable and technologically diverse compared to monoliths. [22]

Historically software architectures have been monolithic in their designs, but they have moved towards modular and service oriented designs. This shift was caused by the complexity of maintaining and scaling large monolithic software systems. MSA was formally introduced for the first time in a workshop of software engineers in 2011, and it started to gain popularity after Netflix, Amazon, Uber and many other famous technology companies shared their successful experiences with the architecture style. [22] [23] In 2014 Lewis and Fowler published a famous blog post, where they defined the architecture style's core principles, practical benefits and how it differed from the earlier Service-Oriented Architecture (SOA) styles, further increasing its popularity. [24] [23]

The core principles of microservices include decomposition into bounded contexts, independent deployment and decentralized governance [1] [22]. A microservice is explicitly defined as a cohesive and independent executable process that communicates by passing messages [22]. Explicitly restricting the responsibilities within clear boundaries makes microservices agile, easily maintainable, and allows teams to work independently of other teams without excessive communication [1] [21] [25].

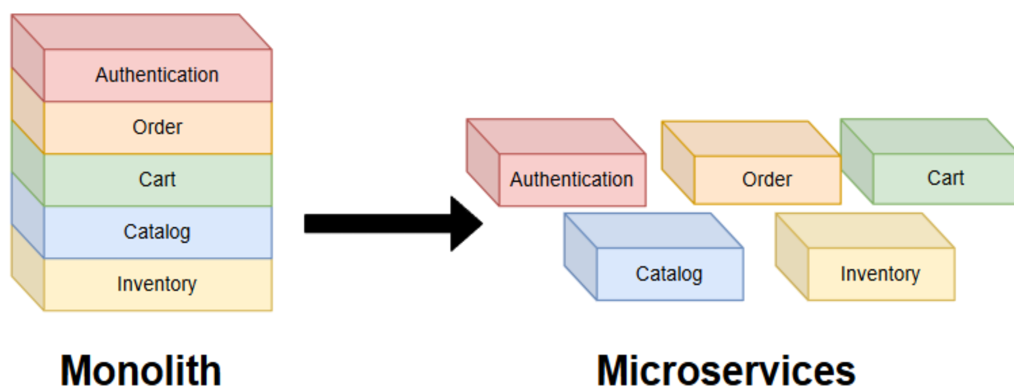


Figure 2.2: The difference of monolith and microservices architectures.

Decentralized data management is another fundamental principle of microservices. It means that each service maintains their own independent databases which

they control whereas monolithic architectures typically have a single centralized database. The decentralization allows each microservice to choose the database technology best suited to their requirements in order to get the best performance. This also allows each microservice and their database to evolve and scale independently. [23]

MSA also promotes freedom in technological choices, thus facilitating heterogeneity in technologies. Monolithic software systems have technology lock-ins that enforce a single technology stack, whereas MSA does not. Because of this organizations can choose the best technologies for each service. [25] The DevOps practices with MSA also are flexible as service updates are independent and isolated, which promotes continuous delivery since the whole system is not redeployed. This reduces the risk of faulty updates taking the whole system down and thus leading to downtime[1]

Resilience and fault tolerance is also a fundamental characteristic of MSA. The aim in MSA is to isolate service failures and minimize impacts of failures to the whole system. In monolithic systems, errors tend to easily propagate through the whole system, which affects the whole systems availability and reliability causing violations in SLA. Whereas in microservices, a failure in a single service does not automatically propagate throughout the system. [1] [23] [25]

2.2.2 Decomposition and Bounded Context

Badly defined service boundaries can negatively affect the scalability of microservices. In monoliths tightly coupled modules are typical, which complicates maintenance and scalability. In microservices, having overlapping boundaries creates unnecessary dependencies that leads to communication overhead resulting from increased latency and resource utilisation. [1] [22] Therefore decomposition and clear service boundaries are very important in order to optimize performance and scala-

bility.

Domain Driven Design (DDD) and its concept of bounded contexts help in designing services in a way to avoid overlapping boundaries. In a bounded context, a specific business domain is modelled into a single clear model and language. These single cohesive models prevent overlapping responsibilities between services and minimizes dependencies across services. [22] Using DDD principles in a microservice context directly supports MSA principles and the decomposition process by providing cohesion and modularization. [25]

Granularity of services brings advantages to scalability, however there is not a consensus on the optimal size of microservices. Services can range from a hundred to thousands of lines of code, which introduces a trade off between total resource usage and complexity. [21] Empirical evidence shows that migrating to microservices increases total resource consumption, specifically memory usage, but is balanced out by the improved horizontal scalability [23]. Thus, smaller microservices reduce the footprint of an individual service, leading to smaller startup latencies. However, it also increases the total number of microservices in the system, leading to increased resource usage and complexity of the system.

The two most critical failures in a decomposition process, as discussed in this chapter, are shared databases and excessively granular microservices. Shared databases between services break the isolation of services and increase coupling, preventing individual scalability of services [26]. Too small microservices increase the total resource usage and complexity of the system, hindering the scalability benefits of microservices. It is essential to be aware of these issues when designing systems with MSA.

2.2.3 Inter-service communication

Inter-service communication in microservices introduces latency and affects the whole performance of the system. Due to the distributed nature of services in MSA, interactions between services introduce overhead through network latency [27] [28]. Since inter-service communication affects the whole systems performance, it is crucial to choose the right communication mechanisms to ensure responsiveness and resilience in order to adhere to SLAs.

The two main communication mechanisms in microservices are synchronous and asynchronous communication. Synchronous communication mechanisms, such as REST and gRPC, are popular since they are relatively simple and offer high performance. But they may be inefficient under heavy loads. Out of the two synchronous mechanisms, REST is relatively simple to use and demonstrates low CPU utilisation. However, gRPC typically demonstrates lower latency than REST, as it is built on HTTP/2, which provides native multiplexing support that allows multiple concurrent request over a single connection. Asynchronous communication mechanisms, such as Kafka and RabbitMQ, work best in fault tolerant and high throughput scenarios, since they decouple services, thus increasing overall system resilience. [28]

API Gateways are a fundamental part of microservices. They route client requests to the appropriate backend services, acting as a single entry point to multiple services. In monoliths a single API Gateway typically routes requests to a single monolithic backend service. In microservices, an API Gateway must route requests to multiple services, which increases complexity and latency. There are two distinct API Gateway patterns that aim to mitigate the complexity and latency. The Gateway Aggregation pattern simplifies client interactions by combining multiple service responses into a single response to the client. The Federated Gateway pattern delegates request processing to multiple specialized gateways responsible of distinct services. The Aggregated Gateway pattern simplifies client interactions,

but it introduces latency due to processing required to combine service responses. The Federated Gateway pattern combats the latency by distributing the request processing, but it also increases the total resource usage of the system. [29]

In inter-service communication, circuit breakers, timeouts and retries are important patterns to improve resilience [30] [31]. Cascading failures and uncertainty of service availability under heavy loads are common problems in MSA. Circuit breakers mitigate these issues by isolating failing services for a certain timeout period, while retries reattempt failed requests mitigating transient failures. [31] Circuit breakers work by tracking a service instance's state and errors. When a certain failure threshold is exceeded, circuit breakers halt requests to that service and gradually allows new requests to the service. [30]

These different service integration patterns are very fundamental to service availability and thus are critical to SLAs. Choosing between different synchronous and asynchronous communication mechanisms is balancing between immediate performance and resilience under heavy loads. Different API Gateway patterns can improve performance and availability, however they can introduce complexity and overall resource usage. And the combinations of fault tolerance patterns are important to ensure availability, but improper configurations can do the opposite.

2.2.4 Operational Concerns in Microservices

Beyond the theoretical principles and design phases of building microservices, operational scalability, resilience and maintainability is what really matters to SLAs, SLOs and ultimately to CSPs and CSUs. Operations have to make sure that microservices can dynamically adapt to varying workloads, recover gracefully from failures and provide visibility into the system's state [22] [21].

In microservices, scalability and elasticity are usually managed by container orchestration platforms. Kubernetes is a popular tool for this purpose, which enables

horizontal and vertical scaling through Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA). The effectiveness of these scaling mechanisms depends considerably on the workloads. Notably in bursty workloads, autoscalers have difficulties. They might scale too late or fail to scale down after a burst in demand has passed. Moreover if horizontal scaling is used for bursty workloads, contrary to expectations it might reduce performance in some cases. These issues lead to inefficient resource usage and potentially SLA violations. It is important to configure autoscalers appropriately concerning the suitable mechanisms, CPU utilisation thresholds and latency triggers. [32]

As we saw in Section 2.2.3, resilience of microservices is improved by circuit breakers, timeouts and retries. Research shows that when these are configured properly, failure propagation can be significantly reduced. However, if they are configured improperly, they can increase system load and reduce system availability. Retries can cause excessive reattempts of requests under heavy loads causing "retry storms", while too long timeouts in circuit breakers can reduce the availability of otherwise healthy services. [30] [31] As with autoscalers, resiliency patterns also require configuration and tuning in order to ensure optimal performance and minimize service downtime.

Observability is difficult in the distributed environment of microservices, but is essential in managing the complexity of microservices. Observability is achieved primarily through metrics, logging and distributed tracing. Distributed tracing frameworks, such as OpenTelemetry, enables visibility into service interactions and dependencies. However distributed tracing frameworks can introduce performance overhead by increasing response latency and reducing throughput. Furthermore, the initialization and setup of tracing in service instances can severely increase cold start latency. It is important to consider the trade-off between tracing granularity, sampling strategies, export methodologies and performance in order to enable

observability without compromising the performance and scalability of the system. [33]

2.3 Container Technologies

2.3.1 Containerisation principles

Containerisation is a very common practice in cloud applications and microservices. In containers, applications are packaged with their dependencies into single lightweight units. Containers use the host operating system kernel, unlike virtual machines (VMs), which use one operating system per instance. This leads to containers having a lighter footprint as such that they can boot more quickly, scale well and use far less resources than VMs. [34] [35] These advantages make containers particularly suitable for the dynamic scaling requirements of microservices.

A container is instantiated from a container image, which is a packaged bundle of the built application and its dependencies, including binaries and libraries. The container lifecycle has three phases: building, distributing and running. In the build phase, a declarative specification such as a Dockerfile describes the required software and dependencies, and the commands used to build the application. in distribution, the result image is distributed to a container registry, where it can be pulled from in the running phase and executed. [36]

Linux kernel primitives, primarily namespaces and control groups, are central to containerisation. Namespaces enable the isolation of containers by providing isolated views of system resources, such as process IDs, network interfaces and files. Control groups support the isolation provided by namespaces with granular resource allocation and management. These two primitives together force containers to only have visibility inside their resource boundaries and avoid resource contention, which could degrade performance. [35] [37]

The adoption of containerisation was accelerated by Docker, the most popular container technology, after its introduction in 2013 [36] [37]. Since then many new container technologies has been introduced. The adoption and evolution of container technologies have been aided by standardization efforts, most notably the Open Container Initiative (OCI). OCI sets a standard for container images, runtime environments and lifecycles. It enables portability and consistency of containers between different platforms. [37]

The low overhead of containers, isolation and fast startup times directly support the scalability requirements of microservices. Updating services is fast, containers are highly available and utilise resources efficiently. All this makes containers highly suitable for microservices.

2.3.2 Orchestration and Autoscaling

Orchestration is essential in containerised services, since it enables autoscaling of containerised services and provides elasticity. Without orchestration, containers are just single instances of services incapable of adjusting to varying demand. Kubernetes is the most popular container orchestration tool and is thought to be the de facto orchestrator. It provides robust scheduling capabilities, advanced resource management and scalability features. [15] [38] [39]

Kubernetes orchestrates containers through a structured hierarchy of clusters, nodes, pods and a control plane, which makes decisions about a cluster and its nodes and pods [38]. A cluster consists of worker nodes, which are physical or virtual machines. Their responsibility is to host and run pods. A pod is the smallest deployable unit in Kubernetes. Pods encapsulate one or more containers with shared resources, storage and networking. The control plane enables high availability and elasticity of pods by replicating them across multiple nodes, while automatically managing their lifecycles and distribution. [40]

Scheduling is a fundamental aspect of Kubernetes and it affects resource allocation, utilisation efficiency and the overall performance of clusters. The default scheduler of Kubernetes mainly considers CPU and memory when allocating pods to nodes. [41] [42] The power of Kubernetes scheduling is that its modular. The scheduling component can be changed and swapped with other scheduling components to consider other metrics, such as network latency. [41] A latency-aware scheduler can consider inter-node network latency and allocate pods in a way to reduce the network overhead of a whole cluster [42]. In microservices environments with abundant inter-service communication, this is particularly beneficial.

In Kubernetes a pod cannot become ready for traffic until a node has pulled an image from a container registry and unpacked the image layers for each container [43] [44]. The image pulling can make up even 76% of a container's startup time [45]. Of course images can be layered and the layers can be cached. But if a node does not have cached copies of layers, then the image size is the most predictable metric for predicting startup time as it takes more time to transfer and uncompress the image layers. And since a pod is only ready when image layers have been built and the application is ready, reducing the image size reduces startup time and improves elasticity. [43] [44]

Kubernetes offers multiple ways to autoscale clusters. But for autoscaling specific microservices and pods, Kubernetes offers the Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) [39]. HPA dynamically adjusts the number of Pod replicas based on metrics such as CPU and memory usage, or custom-defined metrics [46]. The default Kubernetes autoscalers are reactive by nature and scale resources up only after demand changes, potentially affecting service availability when the demand changes rapidly. Although workloads can be scheduled to scale manually beforehand for known peak hours, it still does not help with unpredicted demand spikes. [39] Enhanced autoscaling mechanisms benefit from machine learn-

ing algorithms that predict workload fluctuations and proactively scale to improve availability and efficiency [15].

Vertical scaling of pods in Kubernetes works by manually updating the resource requirements of pods in place or through the VPA, which does not come with Kubernetes by default and is a separate project of its own [39]. The VPA works by tracking the usage patterns of pods and dynamically allocating more memory and CPU resources within set limits [47]. The benefit of this approach is that it allows the scaling of pods without the delays and cold starts of replicating new pods. Using both HPA and VPA at the same time requires careful considerations, since they can only be used together if the HPA tracks other metrics than CPU and memory. A hybrid approach works best by using vertical scaling to determine the optimal resource allocations for pods and then using horizontal scaling to replicate the pods [15].

2.3.3 Networking, Storage, Observability and Security

Networking, storage, observability and security are fundamental operational aspects of deployed cloud services, and thus microservices. In Kubernetes, Container Network Interface (CNI) plugins are used to enable inter-pod communication [48]. There are various CNI plugins that are optimized for different use cases and network layers. Performance analyses reveal considerable differences among the CNI plugins in latency and throughput performance. [49]

Storage in Kubernetes is handled by persistent volumes (PVs). They provide persistent stateful data management for pods. [50] Provisioning PVs and connections between PVs and pods introduces provisioning delays and possible performance overhead depending on the underlying technologies. Performance evaluations of different PV technologies by various cloud providers, reveal variability in input/output performance and latency. [51]

As we already discussed in Section 2.2.4 observability is essential for microservices, but distributed tracing frameworks can introduce performance overhead. In worst case scenarios throughput can be reduced by up to 80% and latency can be increased up to 175%, stemming from the tracing instrumentation and exporting trace data to storage systems. [33]

Securing Kubernetes environments involves many different practices. On top of the best practices of the used cloud platform, securing Kubernetes environments also include container image scanning, runtime policy enforcement and compliance assessments [52] [53]. As with the earlier mentioned operational practices, security practices also can introduce performance overhead. Security hardening studies show that some security hardening tools and practices can increase performance overhead similarly to distributed tracing frameworks [53].

All of the operational aspects discussed in this chapter are fundamental to operating services in the cloud, but they can introduce performance overhead. When designing microservices to be deployed with Kubernetes, different CNI, PV, distributed tracing and security practices and their granularity need to be carefully considered, since they can increase pod startup times, latency and decrease throughput, thus negatively impacting scalability.

2.4 Scalability

2.4.1 Scalability vs Elasticity

Scalability and elasticity are similar and related, and are often used interchangeably in literature, but they mean two different things. Scalability is a system's capacity to handle increased workload and ability to utilise resources effectively. It is central to RQ2 and RQ3, since it describes how a system can handle increasing workload and how efficiently it utilises resources. Elasticity describes how fast and accurately

resources are provisioned following demand and it aims to keep costs low while keeping SLOs intact. Scalability together with automation and optimisation creates elasticity. [5] [6]

Bondi decomposes scalability into four general types of scalability: structural-, space-, space-time- and load scalability. A system can be scalable in more than one ways and these types can overlap. Out of these scalability types structural regards growth without architectural limits, space regards growth without intolerable memory requirements, space-time regards growth without intolerable increases in operation time and load regards growth without delay and intolerable resource consumption. A system with all these mentioned scalability types can handle increased workload, while utilising resources effectively, without degrading performance, without an intolerable increase in memory and without the implementation setting any constraints on capacity. [54] A system with good scalability also is economically attractive, since when traffic grows, resources do not grow intolerably and are utilised efficiently, preventing costs from growing linearly [5] [6]. In this thesis we adopt a scalability definition from a microservice point of view: it is the ability of a service to handle increasing workload by utilising more resource while avoiding resource inefficiencies and keeping an acceptable performance [5] [6] [54].

In contrast to scalability, elasticity is a systems ability to dynamically provision and deprovision resources so that supply tracks demand as accurately and rapidly as possible as discussed in Section 2.1.3. Both scalability and elasticity are important, since good elasticity cannot compensate for poor scalability. Scalability is treated as time-independent and it tells the steady state capacity of a system for a certain resource level [6]. If the steady state capacity of a service is low, it would require more service instances to match demand than a high capacity service would. Also the time required to match demand is longer, since the number of required instances is higher for a service with bad scalability, and spinning up instances takes time due

to cold starts. Importantly a service with bad scalability uses resources inefficiently, requiring more resources to achieve SLOs and leading to increased costs. [5] [6]

2.4.2 From Scalability to Cost

Scalability of individual services define recurring costs in TCO, since in order to sustain a service's SLOs, scalability defines how much computing resources are needed. As discussed in Section 2.4.1, when a service's scalability is low, it uses resources inefficiently and with poor performance as demand grows. Both of these consequences raise operational costs of a service. Bondi characterizes poor scalability as leading to wasteful activity, inefficient scheduling or inability to exploit parallelism, each of these leading to higher resource usage and costs as traffic increases [54]. As we discussed in Section 2.1.5, recurring costs is the channel in which a service implementation primarily affects TCO and the scalability of individual microservices is what primarily defines recurring costs.

Let InfraCost/hr be the hourly total of node prices and optionally storage, networking and observability prices if used. Let C_{SLA} be the steady state capacity or sustained Requests Per Second (RPS) that meet SLOs. A comparable economic measure for microservices would be cost per 1000 requests.

$$\frac{InfraCost_{/\mathrm{hr}}}{C_{SLA}} \times 1000 \tag{2.4}$$

At a fixed SLO, a higher C_{SLA} or lower InfraCost/hr reduces cost per 1000 requests. Better scalability means delivering more work per resource unit, thus directly lowering recurring costs.

In containerised deployments, the recurring cost can be lowered by packing efficiency in to each pod. The per pod density can be determined from the CPU request/limit and memory footprint.

$$pods_{/node} = \min\left(\frac{nodeVCPU}{reqCPU}, \frac{nodeGiB}{reqMem}\right) \tag{2.5}$$

$$nodes = \frac{totalPods}{pods_{/node}} \tag{2.6}$$

If a service with a steady SLO has high RPS per vCPU and a small GiB per request, it requires less pods to accommodate demand. Node count is smaller, which leads to lower InfraCost/hr. In contrast a low C_{SLA} means more nodes or larger nodes are needed to accommodate demand, a symptom of poor scalability which leads to higher recurring costs [54].

Elasticity builds on top of scalability and it affects the cost during demand changes. Autoscaling adjusts the capacity over time in order to preserve SLOs and optimize cost during varying demand. As discussed in Section 2.1.3 over-provisioning protects SLOs but increases costs, while under-provisioning reduces costs but risks SLOs with latency spikes and errors. Elasticity is characterized along accuracy; the magnitude and duration of over- or under-provisioning, and timeliness; the reaction time to demand changes, and finally links these to cost and Quality of Service outcomes. [5] [6] Operators typically keep capacity headroom to absorb bursting demand and preserve SLOs. The effective steady state capacity takes this headroom h into account.

$$C_{SLA,eff} = (1 - h)C_{SLA} \tag{2.7}$$

Thus the additional headroom increases cost per 1000 requests by a factor of $\frac{1}{1-h}$. Since elasticity builds atop scalability and aims to minimize over- and under-provisioning, the most cost-efficient configuration with steady SLOs combines strong steady state capacity with accurate and rapid elastic controls [5] [6].

2.4.3 Metrics and SLIs

As discussed in Section 2.1.4, SLAs operationalize their service level quality through latency, throughput, availability, reliability and in some cases cost-related quality objectives. These qualities are described by SLOs that work as objectives for these quality attributes. SLIs reveal the current state of those objectives in the target system through metrics. They act as a contractual bridge between the system behaviour and delivered quality. [16]

Latency is primarily represented through latency percentiles (p50, p95 and p99) in SLIs for services deployed to the cloud. These percentiles summarize the distribution of response times. Tail-latency refers to the high end of this distribution, for example the slowest 1% or 5% of requests. In microservices a request to a service might often fan out to multiple subrequests to other services. If even one of the subrequests takes a long time to respond, it makes the response time for the main service slow. The probability of at least one subrequest being slow rises with fan out. The end to end latency is therefore mainly dictated by the slowest subrequest. Even though the chance of a slow request might be low, in a large service with millions of users 0.1% of requests being slow means thousands of users are experiencing slowness. For this reason latency percentiles are used as latency indicators rather than average latency. [55]

Throughput is another important SLI used to measure performance [16]. Throughput describes how many requests a service can handle simultaneously and its calculated by dividing the total amount of requests processed by a service with the total time in seconds used to process the requests [23]. Higher throughput essentially means that a service can take on more requests without requests piling up and thus avoiding tail-latency.

Availability as an SLI is very relevant in microservices as elasticity and the scalability of an individual service aim to improve availability. Availability means the

uptime of the provided cloud services and it is affected by service outages and downtime as Equation 2.8 shows. In the equation, A_v is defined as the proportion of time a service is available during a specific observation period, expressed as a percentage. Poor scalability and elasticity in microservices can lead to downtime as an under-provisioned service instance can lead to failed requests and thus decreased benefits or revenue [23]. In SLAs availability is usually expressed by 99% uptime or 99.999% uptime, which is considered high availability and allows for 5 minutes of downtime per year. [16] [56] A bad availability means that some users can not be served, which in turn decreases benefits or revenue.

$$A_v = \left(1 - \frac{downtime}{uptime + downtime}\right) \times 100 \tag{2.8}$$

3 Java-based Microservice Frameworks

This chapter includes a non-empirical, conceptual analysis of chosen microservices frameworks. The goal is to identify *operational characteristics* relevant to scalability. The analysis is based on an examination of documentation and design choices of the chosen frameworks, instead of empirical observation. The analysis is constrained to Java-based frameworks, as it provides a consistent space for design and analysis of the empirical experiments later in Chapter 5.

3.1 Java's Role for Scalable Microservices

3.1.1 Java Adoption and Cloud-Native Capabilities

Java remains a strong and relevant choice for scalable microservices shown by popularity studies. In the 2025 Stack Overflow Developer Survey, Java was one of the most popular programming languages, surpassed only by JavaScript, TypeScript and Python [57]. Some surveys even find Java as the most used language. Lu et al. analysed over 6 million open source projects and found that Java was the most widely used [58]. Java's popularity, wide ecosystem and enterprise adoption make it a very strong choice for scalable microservices [59] [60].

As discussed in Section 2.3 containers are a very common and beneficial practice

in cloud applications and microservices. Modern implementations of the Java platform align with cloud-native principles and respect Linux kernel primitives in order to guarantee reliability in containers [61].

Java applications also have options for cloud optimised compilation and deployment strategies. By default Java applications are run on the JVM, with Just-In-Time (JIT) compilation, which means that bytecode is generated during runtime when needed. It increases memory overhead and cold start duration due to the necessary initialisations being done at application startup. The other option is Ahead-Of-Time (AOT) compilation, where static analysis moves most of the initialisations and bytecode generation to build time, thus avoiding runtime bytecode generation. [62] [63]

The AOT compilation path also provides a further option of compiling applications to GraalVM native images. These GraalVM images are platform specific binary executables optimised for cloud deployments without the requirement of running on the JVM. Removing the JVM startup from container executions significantly reduces cold start duration, resource usage and container image sizes. [62] [63]

3.1.2 Java Frameworks at a Glance

Comparative studies present Spring Boot, Quarkus and Micronaut as the most suitable JVM frameworks for microservices and analyse their startup behaviour, resource usage and steady-state performance [59] [64]. Out of these three frameworks, Spring Boot is the oldest and most used [57]. It has a wide ecosystem and support community [64]. Quarkus and Micronaut are relatively new and do not enjoy the same level of maturity, but they offer similar features for development as Spring Boot and are more tailored to cloud-native environments and microservices [59] [64]. These three frameworks with different levels of scalability act as good candidates for researching the effects of scalability.

The comparative studies compare different performance metrics of the JVM frameworks, such as average response time, latency percentiles and throughput. Additionally the studies compared different operational metrics that are relevant for deployed microservices, such as application startup time, resource usage and container image size. These metrics act as *operational characteristics* that drive scalability and affect elasticity in deployed microservices, namely *startup time*, *resource usage*, *image size* and *throughput*. [59] [64]

The key differences in design philosophy and architectural traits of these frameworks focus on improving these *operational characteristics* (OC). RQ1 focuces on these OCs and aims to analyse their effect on scalability and elasticity. The key differences of the frameworks are their concurrency and I/O models, Dependency Injection (DI), build processes, Ahead-of-Time (AOT) compilation and GraalVM native image support, and are designed to emphasise one or more OCs [59] [64] [65] [66].

3.2 Spring Boot

3.2.1 Dependency Injection

Spring Boot builds on top of the Spring framework and essentially takes an opinionated view of the framework in order to provide a rapid and accessible "getting-started" experience. [67] The Inversion of Control (IOC) principle is a central aspect in modern Java frameworks and Spring implements it with Dependency Injection (DI). In DI, objects define their dependencies through constructor arguments or properties that are set on the object after its instantiation. Spring's IOC container then injects those dependencies when the object is instantiated. This is the fundamental inversion, since the object does not control the dependency instantiation and lifecycle, instead it is controlled from the outside. Objects that are managed by

the IOC container are called beans. [68]

In Spring applications, most beans are discovered implicitly. The most common way of declaring beans is with bean annotations. Classes are annotated with for example @Component, @Service or @RestController. @Component is a generic annotation for any bean, and the other bean annotations are stereotype annotations that provide additional functionality. The framework scans the application's classpath to detect classes with bean annotations and applies corresponding bean definitions. The IOC container then knows how to instantiate and manage these beans. [68] Listing **??** shows an example in Spring Boot, where a class is annotated with @Controller. The framework knows that this is a RestController bean and instantiates the bean and creates endpoints out of the appropriately annotated methods. The class has a property userService, which is annotated with @AutoWired. The IOC container instantiates that bean and injects it into the UserController bean, inverting the control of the userService bean from the UserController bean to the IOC container.

```
@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping
    public UserDto getUser(@RequestParam String userId) {
        return userService.getUser(userId);
    }
}
```

Listing 1: Defining a RestController bean and injecting properties

Reflection is a JVM feature that allows Java code to dynamically examine and invoke its classes, methods, fields and properties during runtime [69]. Spring uses reflection to configure the metadata for beans and to scan the metadata for annotated classes in order to invoke them during runtime [68] [68]. Reflection is mostly used during startup to initialise and configure beans, but it can also be used during runtime to dynamically alter the behaviour of beans. When optimising a Spring application for AOT and GraalVM native images, the usage of reflection should be hinted for GraalVM, since native images have a partial support for reflection. When optimising executables, the usage of reflection should be minimised, since it hints at more runtime initialisations and increased memory usage. [68]

Dependency Injection is a very central part of the Spring Boot framework and it directly affects startup time, memory usage and CPU usage. The *resource usage* then directly affects pod density in a containerised microservices deployment. And

startup time directly affects the speed of elasticity as discussed in Section 2.1.3, we can see that both *resource usage* (OC1) and *startup time* (OC2) are both OCs that are affected by the framework and contribute towards the scalability and elasticity of microservices.

3.2.2 Concurrency and I/O stacks

Spring Boot supports two web stacks; servlet through Spring MVC and reactive through Spring WebFlux. They represent two fundamentally different concurrency and I/O models. [70]

The servlet stack uses a thread per request model to handle incoming HTTP requests inside an embedded servlet container [71]. Configuring the size of the thread pool directly affects how many requests can be processed concurrently before they queue, thus directly affecting throughput [70].

The reactive stack through Spring WebFlux is free of the servlet API and is completely asynchronous and non-blocking. In contrast to the servlet stacks large thread pool, WebFlux uses the event loop concurrency model with a fixed number of non-blocking threads that each serve many connections with asynchronous I/O.[70] [68] Additionally, the Spring MVC has the option for virtual threads, available since JDK 21. They are very lightweight alternatives to the default platform thread and when enabled can potentially increase throughput dramatically. [70] [72]

While the reactive stack tends to use resources more efficiently, it does not make applications run faster. It requires more work to do things in a non-blocking way. The benefit is that reactive stacks scale better with a small number of threads and less memory. This allows applications to scale in a more predictable way, especially in microservices environments where services fan out requests and there is a mix of slow and unpredictable I/O a. [68] As its clear that the design choice of reactive and non-blocking I/O aims to improve *throughput* (OC3) when scaling, *throughput*

is an OC that is affected by the framework's design choices.

3.2.3 Build and Containerisation

Spring Boot offers several strategies for building an application for deployment. They directly affect the startup time, memory footprint, performance and scalability of the built application. For microservices optimised builds, Spring Boot offers two important approaches; running on JVM with AOT processing and compiling the application to a GraalVM native image. Both are first class approaches for cloud deployments and integrate well with OCI-compliant image creation. [70] [68]

Spring Boot's AOT processing analyses the application at build time, computes configurations for beans and the IOC container and generally moves most of the initialisations from the runtime to build time. This leads to a reduced application startup time and reduced memory footprint. However, AOT processing might not be applicable for all cases, since it assumes that the classpath is fixed and bean graphs will not change during runtime. In Spring, this prevents enabling beans based on application properties and changing beans during runtime. [70]

If an application is eligible for AOT processing, then it also is eligible for GraalVM native compilation. A GraalVM native image is a standalone platform specific executable produced with static analysis. Native images already come with the benefits of AOT processing and additionally they do not depend on the JVM, leading to much smaller startup times and memory footprints. Because the static analysis in native image compilation is not aware of reflection and lazy loaded resources, Spring's AOT processing provides Runtime hints for reflection, resource loading and proxies in order to retain features that are required at runtime. [70] For microservices specific deployments, GraalVM native images are the best option, since the built executables are the smallest with the smallest memory footprints.

Most of the optimisation techniques in the build processes aim to decrease the

startup time and resulting container's *image size*, we can say that *image size* (OC4) is another OC that is important for a JVM framework to optimise, since *image size* affects a pod's startup time as discussed in Section 2.3.2

3.3 Quarkus

3.3.1 Dependency Injection

Quarkus' DI implementation is called ArC and is based on the Jakarta Contexts and Dependency Injection (CDI) specification. Quarkus does not implement the whole specification, but just the CDI Lite. This choice stems from Quarkus' architectural choices and goals. [73] ArC differs fundamentally from Spring's IOC container as it moves most of the initialisations and metadata configurations of beans to the build time, which means that *startup time* and *resource usage* are reduced, thus improving OC1 and OC2. [73] [65].

To improve memory footprint, Quarkus aggressively eliminates unused code from being included in executables, thus lowering memory usage and improving OC1. ArC detects any unused dependencies and removes them by default during build time. This makes the bean graph smaller and reduces the amount of metadata during runtime. [73]

Very integral part of initialising dependencies during build time is build time augmentation. Build Step Processors analyse the applications annotations and descriptors, and generate recorded bytecode that instantiates and wires runtime services directly. The metadata of dependencies are generated at build time, thus avoiding reflection at startup. Quarkus can even serialize pre-initialised state into native images by booting the application in the build step and writing the booted state into a native image, thus lowering *startup time*. [74]

To further avoid runtime initialisations, ArC tries to replace reflection with gen-

erated bytecode wherever possible. This makes injection and proxying reflection free. Instead of dynamic runtime proxies, Quarkus generates custom proxies free of reflection. [73] [65] In Quarkus, a bean is not proxyable if it is final, has final methods, private members or does not have a non-private zero-argument constructor. However, by default Quarkus rewrites these properties to make classes proxyable and ultimately reflection free, thus lowering resource. [73]

3.3.2 Concurrency and I/O stacks

Quarkus at its core is reactive by design and it means that every Quarkus application is reactive. However the code still can be imperative or reactive, offering versatility. [65] [75] The reactive HTTP layer is implemented by RESTEasy Reactive and uses a non-blocking event loop model. However, there is an option to use the imperative blocking servlet stack. [76]

Quarkus handles HTTP requests on the I/O thread with the event loop model in order to maximise *throughput*. If an endpoint requires blocking work, it cannot be run on the I/O thread. In that case the endpoint can declare blocking work with @Blocking, thus offloading the work to a separate worker thread and avoiding blocking of the I/O thread. This approach unifies imperative and reactive programming styles in Quarkus, as code can be either reactive or imperative while the runtime is reactive. [76]

While the reactive I/O model is the default and there is an option for the blocking servlet stack, Quarkus comes with a third option [65]. Just like Spring Boot can leverage virtual threads as discussed in Section 3.2.2, Quarkus can too. Imperative blocking endpoints can offload work to virtual threads with @RunOnVirtualThread. This creates a new concurrency strategy for blocking imperative style code, where each invocation uses a new virtual thread, thus avoiding blocking the I/O thread and the overhead of spawning and switching between heavy platform threads. [76]

3.3.3 Build and Containerisation

Quarkus' build processes are explicitly tailored towards container deployments as AOT does not need to be explicitly enabled as per Quarkus' design philosophy of moving most of the work to build time [77]. Additionally, Quarkus even provides Kubernetes extensions that enable single-step Kubernetes deployments [78]. Even for JVM deployments, the default fast-jar packaging minimises *startup time* and memory footprint by pre-indexing dependencies, thus avoiding wide and expensive classpath scans and ultimately improving OC1 and OC2 [79].

For native executables, Quarkus can use a local installation of GraalVM or a container image to build the executable. GraalVM emits a 64-bit Linux binary, which is then copied into a minimal container image. Quarkus offers several one-line commands for flows that combine native compilation and container image creation. Additionally Quarkus comes with ready-made Dockerfiles for native runtimes with varying utilities and memory footprints, such as the native and native-micro Dockerfiles. [80] The support for optimising *image size* is first class in Quarkus.

3.4 Micronaut

3.4.1 Dependency Injection

Similar to Quarkus, Micronaut's DI also emphasises moving as much work as possible to the build time. In Micronaut, annotation processors generate BeanDefinition classes and metadata at build time via bytecode generation. At startup the framework already knows every injection point, so there is no need to scan all classes, fields, methods and constructors leading to reflection free startups and thus reduced *startup time* and improved *resource usage*. [66]

Micronaut also allows conditional wiring and replacement of beans without reflection, which is something that Spring Boot with AOT compilation does not even

allow. Beans are chosen by type and qualifier. And for conditional loading, the @Requires annotation is used and for substituting beans, the @Replaces annotation is used. [66]

3.4.2 Concurrency and I/O stacks

Micronaut's HTTP stack is non-blocking, and the documentation does not list servlet or any other blocking alternatives. Similar to Quarkus, the event loop concurrency model is used to maximise *throughput*. Controller methods are executed on the same thread that handles the network I/O, unless offloading blocking operations to another thread is required. This approach allows imperative style code with the event loop model, while allowing blocking for localised operations. [66]

Micronaut does also allow offloading work to virtual threads. The framework detects if virtual threads are available and if the code offloads work to blocking executors, the work is offloaded to virtual threads. Additionally, the event loop can be made to use the virtual threads. This approach improves performance by avoiding context switches between the event loop and the virtual threads. However, this feature is experimental. [66]

Similar to Quarkus, Micronaut also embraces reactive programming through Project Reactor or RxJava. Methods can be wrapped with reactive types in order to avoid blocking operations and let the runtime work on the event loop, only using explicit thread offloading when blocking operations are needed. [66] Micronaut's data access module; Micronaut Data, also supports reactive query executions by returning a reactive type and reserving blocking operations to the application's configured I/O thread pool [81].

3.4.3 Build and Containerisation

Micronaut already shifts most of the initialisation work to build time, like Quarkus does. But it comes with an additional Micronaut AOT module, that pre-parses configuration, pre-computes bean requirements and substitutes classes with environment specific optimised versions. These environments being either JVM or GraalVM. [82]

For native executables, Micronaut also supports GraalVM. The build process creates a native binary executable and then packages it into an OCI image. The container tooling offers several options for packaging applications, in startup time slowest to fastest: plain JVM, AOT optimised, CRaC, native and AOT optimised native. Unsurprisingly, the AOT optimised native image is the fastest to start in milliseconds while the plain JVM image can take seconds. [66] [83] [84] Thus Micronaut offers similar paths for optimising *startup time* and *image size*.

3.5 Differences in Operational Characteristics

This chapter maps the different *operational characteristics* that influence scalability and elasticity in deployed microservices and summarises how the frameworks contribute to each *operational characteristic*.

All three frameworks have different options for influencing the *operational characteristics* of deployed microservices. *Startup time* is mainly affected by how much work is done at build time, thus AOT is central in this. *Resource usage* is affected by how much CPU and memory an application consumes, which is again affected by AOT and the use of reflection. Throughput is affected by the I/O model used. And finally container image sizes are mostly affected by GraalVM native images and the different build optimisations of each framework. As we can see from Table 3.1, most of the characteristics are affected by AOT and reflection. These in turn are affected

Table 3.1: Operational Characteristics in JVM frameworks

Operational Characteristic	**Spring Boot**	**Quarkus**	**Micronaut**
Resource usage (OC1)	AOT	AOT, minimal use of reflection, custom proxies	AOT, minimal use of reflection
Startup time (OC2)	AOT	AOT, minimal use of reflection, elimination of dead code	AOT, minimal use of reflection
Throughput (OC3)	Servlet (default), reactive	reactive (default), servlet	reactive
Image size (OC4)	GraalVM native images	GraalVM native images	GraalVM native images

mainly by the different DI implementations of the frameworks.

4 Literature review

This chapter includes the literature review for examining existing research about scalability in microservices context. Key research goals are to investigate the drivers of scalability, user-perceived performance under load and cost implications of scalability in microservices. The purpose of the literature review is to give a theoretical context for analysing the results later in Chapter 5 and identify established findings for answering the research questions.

4.1 Methodology

4.1.1 Research Target

To build a focused evidence based foundation for our research questions and the empirical research in Chapter 5, a systematic literature review was conducted. This literature review consisted of three searches targeting each of the research questions. The resulting articles were then screened based on their relevancy to the research questions. The relevancy of an article was determined by personal judgment and thus it is subjective. The goal was to understand which *operational characteristics* affect scalability and elasticity in containerised microservices deployments (**RQ1**), how user-perceived performance differs between different levels of scalability (**RQ2**) and how cloud TCO is affected between different levels of scalability (**RQ3**).

Our research questions are:

- **RQ1**: What kind of operational characteristics drive scalability in containerised cloud deployments?
- **RQ2**: How different levels of scalability affect user-perceived performance in microservices under heavy load
- **RQ3**: How different levels of scalability affect the total cost of ownership of deployed microservices?

For clarity, this thesis distinguishes scalability and elasticity from each other as explained in Chapter 2.4.1, however this literature review assumes that they might appear as synonyms and might be used interchangeably in the literature. In this thesis user-perceived performance means latency percentiles and throughput, and cloud TCO means the costs related to running and operating microservices in the cloud.

While the empirical research in Chapter 5 focuses on analysing the different levels of scalability and their effects between JVM frameworks, the research questions and thus the literature review are framework agnostic. The scope is focused on container-orchestrated microservices deployments and their scalability and elasticity. Programming language or framework benchmarks are excluded unless the results can be generalised to orchestration-level behaviour.

4.1.2 Search Queries

One targeted search query was conducted for each research question across IEEE Xplore, ACM Digital Library and Google Scholar. IEEE and ACM provide peer-reviewed articles and conference papers from systems and software venues while Google Scholar widens the search pool.

The search queries were crafted by a set of keywords that are relevant to microservices and their scalability, and another set of keywords that targeted each of

the research question. The search queries were:

- **SQ1**: *microservice* AND container* AND (scalability OR elasticity) AND (determinant* or driver* OR factor* OR characteristic*) AND (empirical OR measurement OR benchmark OR evaluation OR "case study")*
- **SQ2**: *microservice* AND container* AND latency AND (benchmark OR "heavy load" OR stress OR spike OR burst OR overload)*
- **SQ3**: *microservice* AND container* AND (cost OR TCO) AND (autoscal* OR elasticity OR scalability OR "scale to zero")*

These search queries are numbered respective to the research questions they target. **SQ1** aims to find articles that investigate scalability optimisation in deployed microservices i.e. which *operational characteristics* of microservice containers improve scalability and elasticity of the whole microservices deployment. **SQ2** aims to find articles that investigate how optimising microservices deployments affects user-perceived performance. **SQ3** aims to find articles that investigate how optimising microservices deployments affects cloud TCO.

4.1.3 Inclusion and Exclusion Criteria

Articles were searched with the search queries from the previous section with a publication date filter between 1st of January, 2020 and 17th of October, 2025. The search only included research articles, conference papers and omitted any abstract only entries. Additionally any duplicate results and non-English literature were excluded. It is also worth mentioning, that access to some literature is restricted, so naturally only articles that had open access or were available through the University of Turku were included.

To further ensure the relevance of the resulting literature to the literature review, screenings based on titles, abstracts and full text reviews were conducted. The

literature had to be relevant to the topic of microservices, scalability and/or elasticity and to the specific research question. For an article to be included, it had to consider microservices or container-orchestrated deployments and their optimisations. The relevance screening was divided into two separate screenings. First by title and abstract and then by a full text review.

The total list of screening criteria were:

- **Relevance**: Entries had to be directly relevant to the **RQ** in question.
- **Redundancy**: Entries found in a previous search of the same **SQ** were omitted.
- **Non-English**: Included only English literature.
- **Open access**: Entries had to be open-access or available through the University of Turku.
- **Publication date**: A publication date filter from 1st of January to 17th of October was applied to the search queries.

4.1.4 Literature Review Process

The literature review process can be divided into distinct phases based on the initial search and the following screenings:

- Initial search
- Pre-screening collection
- Title and abstract Screening
- Full text screening

Initial Search: First, three searches were conducted with the previously mentioned search queries and a publication date filter was applied. SQ1 generated 30, 1394 and 474 results in IEEE Xplore, ACM Digital Library and Google Scholar respectively. SQ2 generated 57, 1106 and 9706 results. And SQ3 generated 109, 1414 and 3940 results.

Table 4.1: Initial search

Search query (SQ)	IEEE Xplore	ACM Digital Library	Google Scholar
SQ1	30	1394	474
SQ2	57	1106	9706
SQ3	109	1414	3940

Pre-Screening Collection The results of the initial search were sorted by relevance and up to 40 results of each search were collected for the next phase. In order for an entry to be included to the next stage it had to be open access, written in English, a full article, and not a duplicate from a previous search of the same **SQ**. **SQ1** yielded all of the 30 results from the initial search in IEEE Xplore and **SQ2** yielded only 2 results from IEEE Xplore, since the rest of the articles had restricted access. Rest of the searches yielded 40 results each.

Table 4.2: Pre-screening collection

Search query (SQ)	IEEE Xplore	ACM Digital Library	Google Scholar
SQ1	30	40	40
SQ2	2	40	40
SQ3	40	40	40

Title and abstract screening After the Pre-screening collection, articles were screened based on the relevancy of their titles and abstracts. **SQ1** generated 16, 10 and 13 results in IEEE Xplore, ACM Digital Library and Google Scholar respectively. **SQ2** generated 0, 7 and 10 results. And **SQ3** generated 9, 13 and 12 results.

Table 4.3: Title and abstract screening

Search query (SQ)	IEEE Xplore	ACM Digital Library	Google Scholar
SQ1	16	10	13
SQ2	0	7	10
SQ3	9	13	12

Full text review After title and abstract screening, the results were screened by reading the introduction and conclusion chapters and skimming through the rest of the text to determine the entry's relevancy to the **RQ** in question. **SQ1** resulted in 5, 0 and 2 results in IEEE Xplore, ACM Digital Library and Google Scholar respectively. **SQ2** resulted in 0, 4 and 3 results. And **SQ3** resulted in 0, 2 and 7

results.

Table 4.4: Full text review

Search query (SQ)	IEEE Xplore	ACM Digital Library	Google Scholar	Total
SQ1	4	0	2	6
SQ2	0	4	3	7
SQ3	0	2	6	8

Out of the three targeted search queries and the following screenings 21 articles resulted in total. **SQ1** resulted in 6 articles in total after the screenings, **SQ2** resulted in 7 articles total and **SQ3** resulted in 8 articles total. Table 4.5 lists all the resulting articles. The articles are labeled as A1, A2, A3 and so forth. The results of each **SQ** are presented in the next section.

Table 4.5: Resulting articles

Art. (A)	Title	Author(s)
A1	"A Study of Response Time Instability of Microservices at High Resource Utilization in the Cloud"	Wang et al.
A2	"Advancing Web Development: A Comparative Analysis of Modern Frameworks for Rest and Graphql Back-end Services"	Zanevych

continued on next page

Table 4.5 – *continued from previous page*

Art. (A)	Title	Author(s)
A3	"Analysis, Evaluation, and Assessment for Containerizing an Industry Automation Software"	Sarkar et al.
A4	"Event-Driven Architecture (EDA) vs API-Driven Architecture (ADA): Which Performs Better in Microservices?"	Rahmatulloh et al.
A5	"Extending Microservices Performance Optimization Through Horizontal Pod Autoscaling: A Comprehensive Study"	Buzato and Goldman
A6	"Performance Optimization During HP-UX to Cloud Container Shifts"	Simha
A7	"Designing Microservices That Handle High-Volume Data Loads"	Guntupalli and Ch
A8	"Development Frameworks for Microservice-based Applications: Evaluation and Comparison"	Dinh-Tuan et al.
A9	"Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices"	Jia and Witchel
A10	"Optimization of Microservices Architecture Performance in High-Load Systems"	Iurchenko

continued on next page

Table 4.5 – *continued from previous page*

Art. (A)	Title	Author(s)
A11	"Optimizing Cold Start Latency in Serverless Computing: A Comprehensive Review of Techniques and Emerging Solutions"	Chaudhary et al.
A12	"PathFence: Reducing Cross-Path Dependencies in Microservices"	Gu and Wang
A13	"StatuScale: Status-aware and Elastic Scaling Strategy for Microservice Applications"	Wen et al.
A14	"An Auto-Scaling Approach for Microservices in Cloud Computing Environments"	ZargarAzad and Ashtiani
A15	"Application deployment using containers with auto-scaling for microservices in cloud environment"	Srirama et al.
A16	"Burst-Aware Predictive Autoscaling for Containerized Microservices"	Abdullah et al.
A17	"Cdascaler: a cost-effective dynamic autoscaling approach for containerized microservices"	Shafi et al.
A18	"CEMA: Cost Effective Multi-Layered Autoscaling for Microservice based Applications"	Shafi et al.
A19	"Erlang: Application-Aware Autoscaling for Cloud Microservices"	Sachidananda and Sivaraman

continued on next page

Table 4.5 – *continued from previous page*

Art. (A)	Title	Author(s)
A20	"Self-adaptive, Requirements-driven Autoscaling of Microservices"	Karol Santos Nunes et al.
A22	"SLO and Cost-Driven Container Autoscaling on Kubernetes Clusters:"	Marchese and Tomarchio

4.2 Literature Review Results

This section presents the results of each of the search queries. Each subsection includes a table that lists the results of the search query and their relevant findings. Analysis of the results are conducted in Section 4.3.

4.2.1 Search Query 1

SQ1 targeted *operational characteristics* that drive scalability in containerised microservices. **SQ1** resulted in 6 articles after the screenings.

Table 4.6: Results of Search Query 1

Art. (A)	Summary	Findings
A1	The article investigated the leading cause of response time instability in microservices under load. The leading cause was a long chain of service dependencies where a bottleneck can trigger a queueing effect from downstream services to upstream services.	To mitigate response time instability of bursty workloads the article proposes: - Adding more threads to handle more requests. - Asynchronous architecture for high *throughput* with smaller *resource usage* - Dynamic autoscaling
A2	The article compared leading web development frameworks. The comparison evaluated performance, scalability, usability and community support.	Node.js' scalability is highlighted thanks to its non-blocking event-driven architecture and Spring Boot's scalability is also highlighted because of its cloud-native abilities.
A3	The article presents a case study of migrating a distributed software to a containerised MSA and offers recommendations for migrating monolithic applications to MSA.	Asynchronous message passing does not incur any significant performance overhead and increases the availability of services compared to synchronous TCP/IP communication. Endpoints should be made asynchronous wherever possible.
A4	The article compared the performance of synchronous data communication and asynchronous data communication.	Asynchronous architecture in microservices led to 30% better response times, 7% reduced error rates and 4% reduced CPU usage.

continued on next page

Table 4.6 – *continued from previous page*

Art. (A)	Summary	Findings
A5	The article investigated the benefits of integrating HPA to microservices.	Horizontal Pod Autoscaling (HPA) led to significant improvements in scalability. However the effectiveness of HPA depends on the speed of provisioning and resource usage of pods.
A6	The article presents a framework for optimising performance when migrating legacy systems to cloud containers.	*Throughput* was a key optimisation metric when containerising a legacy system. It was important to optimise the *image size* and *startup time*.

Table 4.6 lists articles that focused on optimising microservices deployments and reported *operational characteristics* that affected scalability. For each article the key recommendations and considerations were evaluated as whether they signalled an *operational characteristic*. It was found that the most cited *operational characteristics* were *throughput* and *resource usage*. Also *image size* and *startup time* were mentioned by A6 to be important to optimise.

4.2.2 Search Query 2

SQ2 focused on user-perceived performance under load and how much scalability optimisations can improve performance. This search query resulted in 7 articles after the screenings.

Table 4.7: Results of Search Query 2

Art. (A)	Summary	Findings
A7	The article investigated how to design microservices that handle high volumes of data and presented a case study, where an e-commerce platform under synchronous restful microservices was migrated to asynchronous event-driven architecture with Kafka.	The old architecture had major performance issues under 1 million orders per minute. The new architecture had 10x the ordering processing capacity and handled 2 million orders per minute comfortably. End-to-end latency decreased by 60%, 99th percentile latency was reduced by 500 ms and uptime rose to 99.99%.
A8	The article investigated different development frameworks for developing microservices.	Frameworks with various levels of scalability differed in their end-to-end latencies from roughly 200 ms to roughly 400 ms in a benchmark application. Poor scalability can lead to even 2x latency.
A9	The article presented a serverless function runtime called Nightcore. Compared to baseline runtimes, Nightcore moves the runtime overhead from millisecond-scale to microsecond-scale.	In a serverless microservices deployment, optimising the cold start duration led to 1.36x—2.93x higher *throughput* and up to 69% reduction in tail latency.

continued on next page

Table 4.7 – *continued from previous page*

Art. (A)	Summary	Findings
A10	The article investigated strategies for optimising MSA under load.	A microservice deployment was optimised with Command and Query Responsibility Segregation, Caching, autoscaling and asynchronous message passing with Kafka. Compared to the baseline the average response time was halved and error rate decreased from 1.2% to 0.5%.
A11	The article addresses the problem of cold starts in serverless deployments and presented techniques for mitigating cold starts.	In a serverless microservices deployment the cold start latency was reduced from roughly 700 ms to roughly 150 ms by using lightweight runtimes.
A12	The article addresses the problem of cross-path dependencies in microservices, where a single bottleneck can cause response time instability. The article proposed a framework called PathFence to reduce the impact of cross-path dependencies.	The solution prevented from requests queueing up, thus reducing the 99th latency percentile by up to 80% and decreasing the number of dropped requests by more than 90% across multiple benchmarks.
A13	The article presented a custom resource allocation framework for Kubernetes to mitigate the impact of unexpected bursty demand.	Compared to other state-of-the-art methods, the proposed framework reduced the average response time by roughly 10% with smaller *resource usage*.

Table 4.7 lists the articles that reported end-user metrics in different scalability levels. The articles reported various metrics such as *throughput*, average response times, latency percentiles and end-to-end latencies. From each article the end-user facing metrics metrics were extracted and whether an scalability optimisation resulted in improved user-perceived performance. Also the techniques that improved user-perceived performance were extracted if explicitly stated. All of the articles in Table 4.7 report improved user-perceived performance after optimisations.

4.2.3 Search Query 3

SQ3 examined how different levels of scalability affected costs and SLA adherence. This search query resulted in 8 articles after the screenings.

Table 4.8: Results of Search Query 3

Art. (A)	Summary	Findings
A14	The article presented a custom autoscaling approach for Kubernetes, where a multi-criteria decision-making method to determine the optimal resources for a service.	A custom autoscaling approach for microservices led to an average improvement of 40.74%, 20.28% and 28.85% of resource utilisation in three distinct datasets. This led to cost reductions of 1.64%, 1.89% and 1.67% respectively.

continued on next page

Table 4.8 – *continued from previous page*

Art. (A)	Summary	Findings
A15	The article proposed a new container-aware application scheduling strategy, where applications are deployed to the best-fit lightweight containers. Additionally containers are deployed in a way to minimise the number of physical machines.	The proposed method improved the CPU and memory utilisation by 9—15% and 10—18% in two datasets. Overall the strategy reduced the processing costs of microservices by 12—20%.
A16	The article presented a custom autoscaling method that predicts bursts and schedules resources ahead of time with workload forecasting and resource prediction.	The approach decreased SLO violations by 5.17x times while incurring 0.767x times more costs.
A17	The article presented a custom autoscaling method that employs machine learning to dynamically allocate the optimal amount of CPU to microservices during autoscaling events.	The solution reduced the number of unprocessed requests and SLO violations by 7.55x to 97.77x, while incurring 1.09x to 8.18x less costs in a benchmark application.

continued on next page

Table 4.8 – *continued from previous page*

Art. (A)	Summary	Findings
A18	The article presents a cost-effective multilayered autoscaling approach that predicts workloads and dynamically adjusts the number of containers and VMs. Additionally the approach moves containers from underutilized VMs to those with available capacity in order to minimise the number of VMs.	The proposed approach achieved 1.37x to 1.63x better cost-efficiency than baseline strategies with less than half of the SLO violations.
A19	The article presented Erlang, a custom autoscaling approach that employs machine-learning to minimise the number of resources used while sustaining SLOs.	Erlang provides 19.3% reduced costs on average in 63 different workloads. It meets the SLO targets in 53 workloads and is the most cost-effective policy compared to several utilisation and machine-learning based autoscaling approaches in 48 out of the 53 workloads.
A20	The article presented a custom SLO-driven self-adaptive autoscaling solution. It made scaling decisions based on SLOs and assigned the minimal amount of resources to achieve those SLOs.	The solution outperformed the Kubernetes baseline HPA and achieved zero SLO violations with 50% less CPU time, 87% less memory and 90% fewer replicas compared to HPA, thus potentially reducing costs significantly.

continued on next page

Table 4.8 – *continued from previous page*

Art. (A)	Summary	Findings
A21	The article presented a custom autoscaling approach that integrates SLOs with a cost-driven policy. The approach makes efficient decisions by balancing SLOs with operational costs.	The approach outperformed the Kubernetes HPA by having roughly half the 90th latency percentile at roughly half the costs under load.

Table 4.8 lists the articles that optimised scalability or elasticity and connected them to business outcomes. From each article we extracted the optimisation method, which was a custom autoscaling method in each case, and whether the article reported indications of change in SLO violations or costs. The indications were cost deltas, cost efficiency, resource usage, resource efficiency and SLO violation deltas. All of the articles in Table 4.8 reported improved positive benefits to costs, resource usage or SLA adherence after optimisations.

4.3 Discussion of Results

4.3.1 Drivers of Scalability (RQ1)

The results from **SQ1** listed in Table 4.6 consistently talk about optimising *throughput* through asynchronous and non-blocking request handling. Articles A1, A2, A3 and A4 found asynchronous request handling increases *throughput* and manages bursty workloads better than synchronous request handling with smaller *resource usage*. A4 quantifies the benefit: asynchronous architecture led to roughly 30% better response times, 4% less CPU usage and with 7% reduced error rate. To optimise

throughput, asynchronous architecture limits queue growth and tail-latency under load with smaller *resource usage*.

The results also emphasised the positive impact of autoscaling to scalability. In this thesis that means elasticity. However the effectiveness of elasticity depends on the speed of provisioning and application *startup time*. The speed of provisioning and *startup time* time are also largely affected by *image size* as discussed in Section 2.3.2.

Additionally, one of the findings of A6 was that optimising *image size* and *startup time* are important for scalability as it improves the elasticity of the microservices deployment. A2 promoted the cloud-native abilities of Spring Boot, but it was unclear what is meant with cloud-native abilities and what these abilities provide. It is possible that cloud-native in that article means the same AOT compilation and GraalVM native images as discussed in Section 3.2.3.

The results also consistently mention optimising the *resource usage* of microservices. This means that OC1 is central as *resource usage* largely determines pod density and thus how many pods can be fitted into a node. In other words doing more with less.

4.3.2 User-Perceived Performance Under Load (RQ2)

The results of **SQ2** in Table 4.7 consistently show that microservices deployments can receive major benefits to user-perceived performance from improved scalability and elasticity. The articles proposed various ways of improving scalability with custom autoscaling, asynchronous architecture, *cold start* optimisations, caching and various architectural patterns. The benefits reported were considerable.

A7, A8, A10, A12 and A13 all showed that scalability improvements decrease average response time considerably. The improvements ranged from roughly 10% to roughly 60%. A12 reported that a deployment was optimised with custom concur-

rency strategies combined with asynchronous communication and the 99th latency percentile dropped by up to 80% and the number of dropped requests decreased by more than 90%. The findings of A7 were particularly interesting because a synchronous RESTful microservices architecture was migrated into an asynchronous event-driven architecture with major benefits. The synchronous version struggled with 1 million orders per minute, while the new architecture handled 2 million orders per minute easily. End-to-end latency decreased by 60%, 99th percentile decreased by 500 ms and uptime rose to 99.99%.

The findings of A9 and A11 come from serverless microservices deployments, which differs from Kubernetes deployments as *cold start* is more prevalent. The findings do note considerable benefits from optimisations to *startup time* and the speed of provisioning. *Throughput* was increased by up to 2.93x and tail latency was reduced by 69% by using lightweight runtimes and smaller *image sizes.* While the findings come from a serverless architecture, the mechanisms of lightweight runtimes and smaller images should similarly reduce latency during scale-out in Kubernetes-orchestrated deployments.

4.3.3 Cost and SLA Impact of Elasticity (RQ3)

The results of **SQ3** in Table 4.8 show that by improving scalability and elasticity, cloud costs and SLO violations can be reduced considerably. While the results of **SQ3** only consider elasticity optimisations through custom autoscaling strategies, optimising the scalability of individual microservices brings similar benefits, since elasticity is driven by scalability as discussed in Section 2.4. The benefits in the results are primarily gained from better resource utilisation, fewer pod replicas and faster and more accurate provisioning.

A14 and A15 reported improvements to resource utilisation from 9% to even 40.74%. A20 introduced an SLO-driven self-adaptive autoscaling strategy that

achieved zero SLO violations with 50% less CPU time, 87% less memory usage and 90% fewer replicas. While an improved resource usage is not a direct evidence of less costs, it does hint at better cost-efficiency.

The listed cost reductions in the results are considerable. Cost reductions ranged from roughly 2% to roughly 88% less costs in various experiments. A17 reported 1.09x to even 8.18x less costs and A18 reported 1.63x better cost-efficiency. On average the results showed a cost reduction of roughly 25%. At scale 25% less costs can yield major economical benefits to a CSU.

In addition to cost reductions, the custom autoscaling strategies of the results managed to decrease SLO violations. A16, A17, A18 and A20 all reported reduced SLO violations. A16 and A17 reported 5.17x to 97.77x less SLO violations. A18 reported more than half reduced SLO violations. The solution of A20 achieved even zero SLO violations.

4.3.4 Cross-cutting Synthesis and Limitations

The results of **SQ1**, **SQ2** and **SQ3** form an evidence driven picture. The scalability optimisations from asynchronous request handling and inter-service communication improve user-perceived performance and TCO. Fast provisioning and *startup time* makes services become ready quickly during scale-out. From **SQ1**, the results emphasise asynchronous and non-blocking request handling as the primary way to increase and sustain throughput during burst with less resources. One article reported 30% faster response times with less CPU usage and error rates.

SQ2 shows that asynchronous architecture also leads to improved user-perceived performance. Migrating from a synchronous request handling to an event-drive request handling increased headroom and thus better scalability. End-to-end latency decreased by 60% and 99th latency percentile dropped by hundreds of milliseconds. The number of dropped requests were decreased as request queues are prevented

from forming. Faster time to ready through lightweight runtimes and smaller images increases throughput significantly and decreases tail latency during scale-out.

The cost and SLA results of **SQ3** also shows that improved elasticity brings substantial benefits. Custom autoscaling strategies provided more accurate and faster provisioning with less resources and SLO violations. The results reported 9—41% higher resource utilisation and 25% less costs on average. At the same time SLO violations are decreased and one result even showed zero SLO violations.

While these results show that improved scalability can yield major benefits, some limitations suppress these results. The nature of experiments in the articles spanned heterogenous settings. The results mixed serverless and container-orchestrated deployments, synthetic and production environments and various shapes and sizes of workloads, thus the absolute numbers are not directly comparable. All of the articles from **SQ3** optimising autoscaling policies instead of pod level characteristics, thus they do not isolate how service level optimisations affect cloud TCO. The used metrics for the results varied, for example latency percentiles, average latencies, end-to-end latencies, cost-efficiency and absolute cost deltas, which limits result normalisation. And bias is plausible, since all of the articles show positive results. We treat the results as indicative, optimising throughput with asynchronous architecture and fast provisioning improves scalability and scalability and elasticity improvements bring considerable benefits to user-perceived performance, cloud TCO and SLA adherence.

5 Scalability Experiments

This thesis aims to research how different *operational characteristics* affect scalability and how different levels of scalability affect user-perceived performance and cloud costs. To support this research, three controlled experiments were conducted in a simulated environment with simulated traffic. This chapter describes the experiments, their execution and results.

5.1 Methodology

This empirical research aims to provide answers for the three research questions. For **RQ1** we aim to examine how *throughput*, *resource usage*, *image size* and *startup time* affect scalability. For **RQ2** we aim to investigate how much different levels of scalability affect the end-user facing metrics such as latency percentiles (p50, p90 and p99). For **RQ3** we aim to examine how different levels of scalability affect resource usage and cloud costs.

Our research questions are:

- **RQ1**: What kind of operational characteristics drive scalability in containerised cloud deployments?
- **RQ2**: How different levels of scalability affect user-perceived performance in microservices under heavy load

- **RQ3**: How different levels of scalability affect the total cost of ownership of deployed microservices?

Table 5.1: The three experiments and their target research questions

Experiment (E)	**Research question(s)**
E1	RQ1
E2	RQ1, RQ2, RQ3
E3	RQ3

The three experiments aim to provide answers for one or more research questions. Table 5.1 shows which **RQs** each experiment (**E**) focuses on. Experiment **E1** compares the *startup times* and provisioning times of different benchmark applications with AOT and GraalVM native images enabled or disabled. **E2** compares the synchronous and asynchronous concurrency and I/O models in Spring Boot and inspects what is the effect to scalability, resource usage, latency and error rates. And finally **E3** compares the resource usage and computing costs of the development frameworks when run in a microservices deployment in AWS. For each experiment we are going to have multiple variants of a benchmark application. The variants vary in their development frameworks and configurations. The design of each experiment is described in Section 5.1.2. The design of the benchmark applications are described in the next section.

This empirical research employs the JVM frameworks presented in Section 3. In **E1** and **E2** the benchmark application is a single service application. The aim is to study *operational characteristics* and how improved scalability affects provisioning time and user-perceived performance. For that we do not yet need a multi-service application. For **E3** the benchmark applications consists of 4 services, which allows for inspecting elasticity and resource usage in a simulated microservices environment with inter-service dependencies. For each experiment the business logic and APIs stay constant, while different scalability optimisations create different levels of

scalability.

5.1.1 Benchmark application designs

E1 and **E2** use a single-service benchmark application, since **E1** examines the time to provision a single pod replica and **E2** examines the throughput, latency, resource usage and error rates of a single service. **E3** uses a multi-service benchmark application that is used to examine elasticity and resource usage in a simulated microservices deployment. All of the source code of the experiments are available in a public Github repository. See Appendix B for more.

Single-service application

The application consists of a single service that exposes an endpoint that represents I/O intensive work. The endpoint takes query parameters that the application then uses to fetch information from an external stub service. A single endpoint does not require a lot of logic in an application with a really limited scope, but that is why the application is split into multiple layers in order to introduce additional classes. The application logic consists of a controller layer, service layer and data layer classes for both endpoints. The classes use annotations heavily to utilise beans for configuration, dependency injection, validation and mapping.

Multi-service application

The multi-service application consists of 4 stateless microservices with their own responsibilities. The benchmark application models a simplified product catalog system with clearly defined subsystems that form hierarchical request flows. The services return dummy data and are stateless without any caches or databases. This design choice ensures that the observed behaviour of the services stem solely from the *operational characteristics* of the applications and not from external dependencies.

The high level architecture of the product catalog system is presented in Figure 5.1 and consists of the following services:

- ProductService
- MetadataService
- CatalogService
- AttributeService

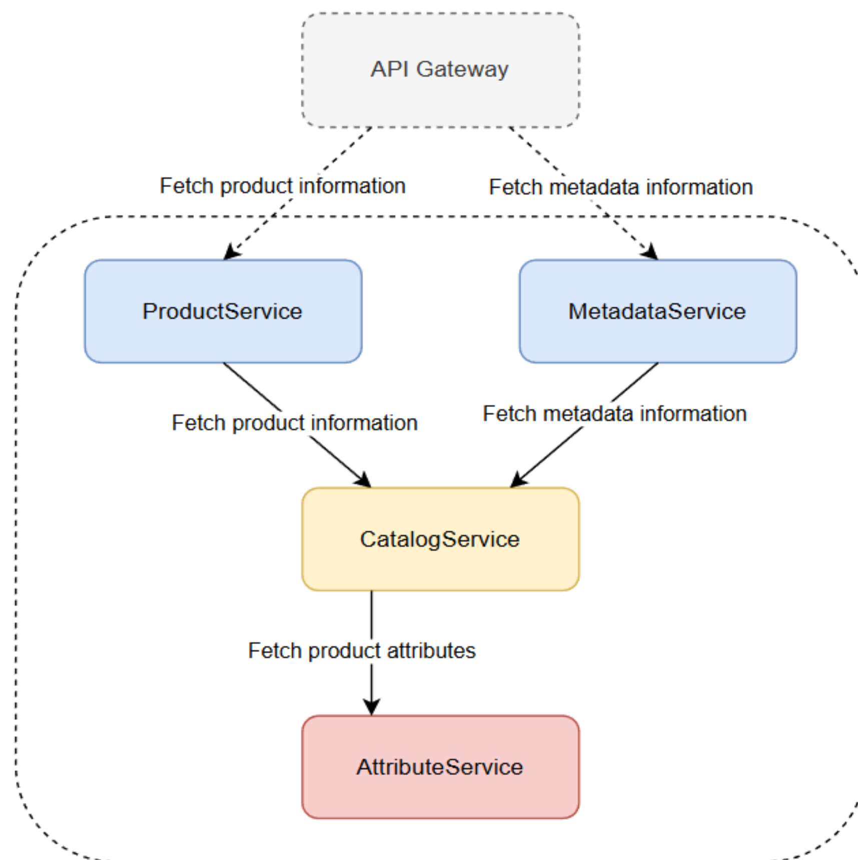


Figure 5.1: The multi-service application architecture

The *ProductService* acts as an entry point for fetching product information and it represents an I/O intensive workload. The service forwards requests to the *Cata-*

logService, *CatalogService* in turn forwards them to *AttributeService* and *AttributeService* creates artificial latency and returns a JSON response. The request fan-out pattern and long dependency chain model a microservice workflow where tail latency increases as a result of multiple internal calls.

The *MetadataService* acts as an entry point for fetching metadata about the products and represents CPU intensive work. The *MetadataService* fan-out requests to the *CatalogService* that simulates CPU work by doing synthetic CPU calculations. This workflow models services that perform local processing after fetching lightweight data for example, score calculation or rule evaluation.

The *CatalogService* is at the middle layer of all the services and is being depended on by the *ProductService* and *MetadataService*. It forwards requests to *AttributeService* and performs CPU intensive work for *MetadataService*. This service is intentionally designed to be a bottleneck and it allows for observing scalability and elasticity reactions in a microservice hotspot.

The *AttributeService* is only used for the I/O intensive workflow and it simulates data fetching from downstream services. It incurs an artificial delay and returns a JSON response.

5.1.2 Experiment Designs

Experiment E1

This experiment's goal is to compare the time to provision a pod replica when the image is cached in the node and when it needs to be pulled from a registry. Additionally the relationship between *image size* and *startup time* is investigated. Metrics to be extracted are image size, average pod startup time, average application startup time and average Kubernetes overhead. The pod startup time is extracted by measuring the time a pod takes to scale from 0 to 1 and the pod logs that the application is ready. The application startup time is extracted from the frameworks

startup logs. The Kubernetes overhead is measured by deducting the application startup time from the pod startup time.

The experiment uses a test script that creates a Kubernetes cluster and deploys all the application variants to pods. These pods are then scaled to 0 and then to 1 to collect the startup time metrics. For each application variant the scale up operation is iterated 50 times. The script runs this setup for two scenarios; images are cached in the node and images need to be pulled from a registry. This allows us to inspect if the image size introduces overhead to the pod startup time.

The benchmark application variants for this experiment are:

- Spring Boot JVM
- Spring Boot native
- Quarkus JVM
- Quarkus native
- Quarkus native micro
- Micronaut JVM
- Micronaut native

Each variant has an individual service. The goal is to inspect a single pod's startup time, thus no additional services are needed. The images are compiled with the framework's default options and thus even large differences between image variants are expected even though the business logic for the applications are near identical. The goal is not to inspect, which framework can potentially produce the smallest image, but what is the implication of various image sizes to provisioning time.

Experiment E2

This experiment's goal is to compare the synchronous and asynchronous concurrency and I/O models in Spring Boot. The goal is to examine, which model provides the best *throughput* and *resource usage* with the same amount of resources assigned to a pod. To investigate SLA adherence, latency percentiles and error rates are also tracked. Additionally the potentially reduced resource usage of native images is investigated, since they do have lightweight runtimes. The benchmark application variants for this experiment are:

- Spring Boot synchronous JVM
- Spring Boot asynchronous JVM
- Spring Boot asynchronous native

The image of each variant is used to build and place corresponding pods in a Kubernetes cluster. The benchmark application exposes two endpoints that simulate CPU intensive work and an I/O operation. In this experiment, only the I/O endpoint is used with a fixed latency parameter of 100ms. The benchmark application delegates the request to the stub service that creates the artificial latency. This way the benchmark application only concentrates on handling requests and not on CPU intensive work, allowing for a clear comparison of the concurrency models.

To generate requests, K6 is used, which is a load testing tool for Kubernetes. To collect metrics of the pods, Prometheus is used. Prometheus exposes telemetry endpoints in the benchmark applications, which are then queried at 5 second intervals. K6 creates virtual users (VUs) that query the endpoint, while waiting for 0.5 seconds between queries. The K6 script in this experiment ramps up the number of VUs from 0 to 500 in 1 minute. It stays at 500 VUs for 50 minutes. This way we can inspect the metrics at a steady state. After 5 minutes, K6 ramps up to 3500 VUs in

the span of 10 minutes or if one of the pre-configured threshold conditions are met. The threshold conditions are that the p95 latency should not exceed 2000ms or the error rate of requests should not exceed 1%. Each benchmark application's pod is assigned a resource limit of 1 CPU and 2GB memory. The stub service's pod is assigned a limit of 4 CPU's and 4 6GB of memory to ensure it is not the bottleneck.

Experiment E3

This experiment aims to evaluate scalability and elasticity of four benchmark application variants in a multi-service deployment in AWS. The goal is to observe how much the different applications with different levels of scalability use resources in the end and how much they cost.

The multi-service benchmark application described in Section 5.1.2 is deployed to AWS, where Elastic Container Service (ECS) and Fargate is used to orchestrate and run the containers. Fargate is a serverless container runtime, that allows running containers without provisioning EC2 instances(AWS virtual machines), which act as nodes. Fargate is billed by CPU and memory usage, which allows for much more granular cost tracking, whereas EC2 is billed by the instance type and how long they are used for. ECS is the container orchestrator, which abstracts away autoscaling algorithms and resource provisioning and does them automatically. Each service container is allocated identical amount of resources of 2 CPU and 4GB memory.

Elastic Kubernetes Service (EKS) and EC2 instances for nodes would have been much more closer to an actual Kubernetes deployment, but they would have required too much time and resources. The combination of ECS and Fargate already gives us the scaling behaviour of Kubernetes with an HPA and allows for much more granular cost tracking. In addition to ECS and Fargate, AWS CloudWatch (an observability tool) is used to gather metrics from the Fargate pods and to display them in a dashboard. Also cost tags are applied to the Fargate resources by variant,

so that costs of each variant can be compared.

The benchmark application exposes two request workflows that simulate CPU intensive work and I/O intensive work. Calls to *ProductService* fan out requests to *CatalogService*, which then fan out requests to *AttributeService*, where an artificial delay is incurred. This represents the I/O intensive work. Calls to *MetadataService* fan out requests to *CatalogService*, where a synthetic CPU calculation is conducted, thus simulating CPU intensive work. The benchmark application variants for this experiment are:

- Spring Boot baseline
- Spring Boot optimised
- Quarkus optimised
- Micronaut optimised

In these variants baseline means the development framework defaults. Optimised means that all scalability optimisations are enabled, which are full asynchronous concurrency and I/O, AOT optimisations and GraalVM native image compilation. We have optimised variants for each framework, but only one baseline variant for Spring Boot. This decision is taken to reduce time and resources used to create and run six application variants. Also the goal is to demonstrate the magnitude of difference of metrics between the lowest scalability application and the highest scalability application. And Spring Boot with default settings is expected to have the lowest scalability, since it uses reflection and runtime initialisations the most when compared to Micronaut and Quarkus. For each application variant the same load test is conducted.

The load test in this experiment creates 2000 VUs at the start. Each VU generates one request to *ProductService* and one request to *MetadataService*, while

waiting one to two seconds between requests. In 20 minutes intervals a burst of VUs is injected for the duration of five minutes. The first burst is 4000 VUs and every burst adds 4000 more VUs. This test lasts for 2 hours and the last burst adds 24000 VUs. This load test allows inspecting how much resources each application variant requires resources and how well they satisfy the demand.

5.2 Experiment E1

5.2.1 Results

E1 investigated pod startup time by scaling pods from 0 to 1 for each application variant and collected related metrics such as image size, average pod startup time, average application startup time and average Kubernetes overhead. The scale up for each application variant was iterated for 50 times and averages were extracted.

The image sizes are displayed in Table 5.2. The JVM variants have the largest image sizes and the native variants have considerably lower image sizes. Out of the JVM variants Quarkus has the largest image size. Micronaut native and Quarkus native micro have the smallest image sizes of 85MB and 88MB respectively.

Table 5.2: Image sizes of the single-service application variants

Application variant	**Image size**
Micronaut native	85MB
Quarkus native micro	88MB
Spring Boot native	131MB
Quarkus native	167MB
Spring Boot JVM	273MB
Micronaut JVM	300MB
Quarkus JVM	442MB

E1 also investigated the pod startup time, which consists of launching a container from an image and starting the application. We extracted pod startup time by polling for readiness of the pods, the framework reported application startup

Table 5.3: Pod startup time with image cached

App. variant	Avg. Pod startup	Avg. App. startup	Avg. Kubernetes overhead	Image sizes
Quarkus Native	1059ms	11ms	1048ms	167MB
Quarkus Native Micro	1073ms	9ms	1064ms	87.5MB
Micronaut Native	1082ms	12ms	1070ms	85MB
Spring Boot Native	1061ms	47ms	1013ms	131MB
Quarkus JVM	1987ms	1194ms	793ms	442MB
Micronaut JVM	2235ms	1294ms	941ms	300MB
Spring Boot JVM	5061ms	3906ms	1155ms	273MB

time and the Kubernetes overhead by subtracting the application startup time from the pod startup time. From Table 5.3 we can see that the native variants reported roughly the same Kubernetes overhead, but the JVM variants had larger differences in Kubernetes overhead. The native variants have almost instant application startup, and with roughly 1000ms Kubernetes overhead the native pods are ready in roughly 1000ms. The JVM variants were slower, with the fastest being Quarkus JVM (1987ms) and the slowest being Spring Boot JVM (5061ms). In Spring Boot's case the native compilation reduced the application startup time from 3096ms to 47ms and reduced the pod start up time by roughly 80%. However there is no correlation to be seen between the image sizes and any metric. That is expected, since in this scenario the images are cached in the node.

Table 5.4: Pod startup time when image is pulled from a registry

App. variant	**Avg. Pod startup**	**Avg. App. startup**	**Avg. Kubernetes overhead**	**Image sizes**
Quarkus Native	2395ms	11ms	2383ms	167MB
Quarkus Native Micro	2309ms	10ms	2299ms	87.5MB
Micronaut Native	2338ms	12ms	2326ms	85MB
Spring Boot Native	2253ms	59ms	2196ms	131MB
Quarkus JVM	4926ms	1229ms	3697ms	442MB
Micronaut JVM	4319ms	1274ms	3044ms	300MB
Spring Boot JVM	6905ms	3827ms	3078ms	273MB

Table 5.4 shows the results of the scaling scenario when images do not live in the node and need to be pulled from a registry. The applications startup times are roughly the same as in Table 5.3, but the Kubernetes overhead and the pod startup times are greater. From the data we can see that the Kubernetes overhead now correlates with the image size. Quarkus JVM with the greatest image size (442MB) has the greatest Kubernetes overhead (3697ms) and is followed by Spring boot JVM and Micronaut JVM. Figure 5.2 shows the correlation, when the data is plotted.

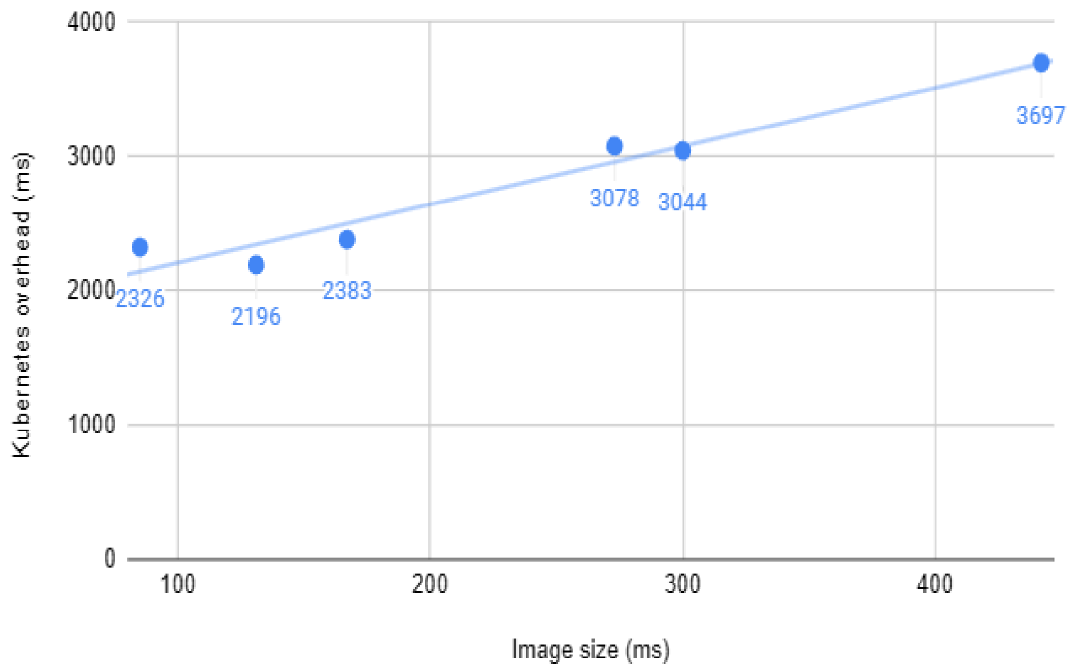


Figure 5.2: The correlation of image size and Kubernetes overhead plotted

5.2.2 Analysis of Results

The results of **E1** listed in Section 5.2.1 show that by optimising a microservice's *image size* and the build artifacts, the *startup time* can be lowered considerably. The slowest pod in **E1** was the Spring Boot JVM variant and took 5061ms on average in a cached scenario to startup, while the native counterpart's startup time was 1061ms on average. In fact, the native compiled applications were the fastest to start with average pod startup times ranging from 1059ms to 1082ms. If Kubernetes overhead is ignored, the application startup time is almost instant, ranging from 9ms on average to 47ms on average.

Out of the JVM variants, Quarkus and Micronaut pods had similar startup times at 1987ms on average and 2235ms on average respectively. They outperformed the startup time of Spring Boot by roughly 3000ms, which was to be expected since Quarkus and Micronaut move almost all of the dependency initialisations to build time and avoid reflection. However, it is important to note that the reported Kuber-

netes overheads vary considerably, even after 50 test iterations. The native variants had similar overhead at roughly 1000ms, but the Quarkus JVM variant had roughly 800ms. The overhead calculation might not be reliable, as application startup time is used to calculate the Kubernetes overhead. The frameworks might measure the application startup time from different points of time and some frameworks might not include everything. The variance of the overhead can also be explained by the container being able to start the application process earlier for JVM variants, since application resources might be loaded lazily in JVM and native Linux executables might load resources eagerly adding to more overhead. The native variants all have roughly the same Kubernetes overhead, which is expected. They do not have any JVM initialisations in the application startup time and have similar build artifacts as they all include a Linux executable.

There was no correlation between *image size*, application startup time or pod provisioning time in the cached test scenarios. But in the uncached scenario, where the image needs to be pulled from a registry, a correlation can be observed. The native variants, which had the smallest images, also had the smallest Kubernetes overhead from 2196ms on average to 2383ms on average. The JVM images were larger and also had larger Kubernetes overhead. The Quarkus JVM variant had the greatest *image size* of 442MB and also the greatest Kubernetes overhead of 3697ms. Table 5.4 and Figure 5.2 demonstrate the correlation. However, it is important to consider that the reported Kubernetes overhead might not be entirely reliable as discussed previously.

The results of **E1** demonstrate the variability in *startup times* and *image sizes*. The provisioning time of pods can differ by seconds and in high volume bursty workloads the impact can be considerable. By optimising *startup time* and *image sizes*, seconds can be saved from provisioning time, thus potentially avoiding SLO violations. Especially in large systems where containers live in multiple nodes or

serverless systems, where container images have to be fetched from a registry every time a container is launched. It is also important to note that the benchmark applications in this experiment were relatively small and comprised of around 10 to 20 classes. In real world applications that comprise of far more classes, the magnitude can be potentially larger.

5.3 Experiment E2

5.3.1 Results

E2 compared the synchronous and asynchronous concurrency and I/O models in Spring Boot. Additionally the potential benefits to resource usage by native images was also investigated. A load testing scenario was run with K6 to inspect metrics at a steady RPS generated by VUs and when ramping up the number of VUs to reach a threshold condition. The threshold conditions were that the p95 latency should be less than 2000ms and error rate should be less than 1%. The metrics that were monitored were CPU usage, memory usage, request throughput and p95 latency. In the experiment, the synchronous JVM variant failed at 2179 VUs due to p95 latency exceeding the limit of 2000ms, while the error rate stayed at 0.0%. The asynchronous JVM variant failed at 3009 VUs due to the error rate reaching 1.00% and the asynchronous native variant failed at 2989 VUs due to error rate reaching 1.05%.

Figure 5.3 shows that the CPU usage of the synchronous JVM variant is the highest. When the number of VUs is kept at 500, the CPU usage of the synchronous JVM variant is roughly 0.95 cores and it reached 0.90 cores around the 110 seconds mark. This variant also reaches the failure point fastest around 700 seconds. Out of the benchmark application variants, the asynchronous JVM has the lowest CPU usage. During the steady RPS at 500 VUs, the CPU usage of the variant is roughly

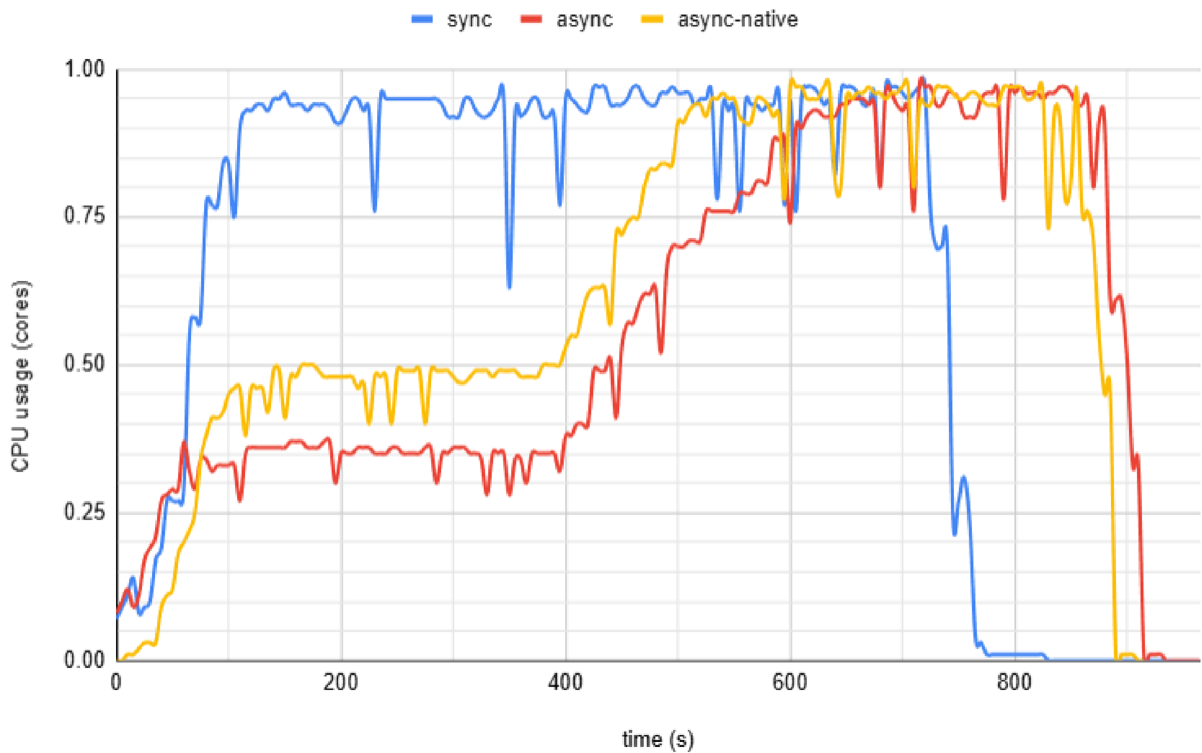


Figure 5.3: The CPU usage of the benchmark application variants

0.35 cores and it reaches 0.90 cores at around 610 seconds. The CPU usage of the native asynchronous variant is slightly higher. During the 5 minutes at 500 VUs, the CPU usage is near 0.50 cores and the variant reaches 0.90 cores around the 500 seconds mark. Both of the asynchronous variants reach the failure point at around 850 seconds.

The memory usage of the synchronous JVM variant is the highest as shown in Figure 5.4. During the 5 minutes with constant 500 VUs, the memory usage is around 350MB. And towards the failure point, the memory usage peaks at around 475MB. The asynchronous JVM variant has considerably lower memory usage. With 500 VUs, the memory usage is around 220MB and it peaks at roughly 340MB near the failure point. The asynchronous native variant initially has significantly lower memory usage compared to the JVM variants. With 500 VUs, the memory usage is around 60MB. But when the VUs are ramped up, the memory usage exceeds that

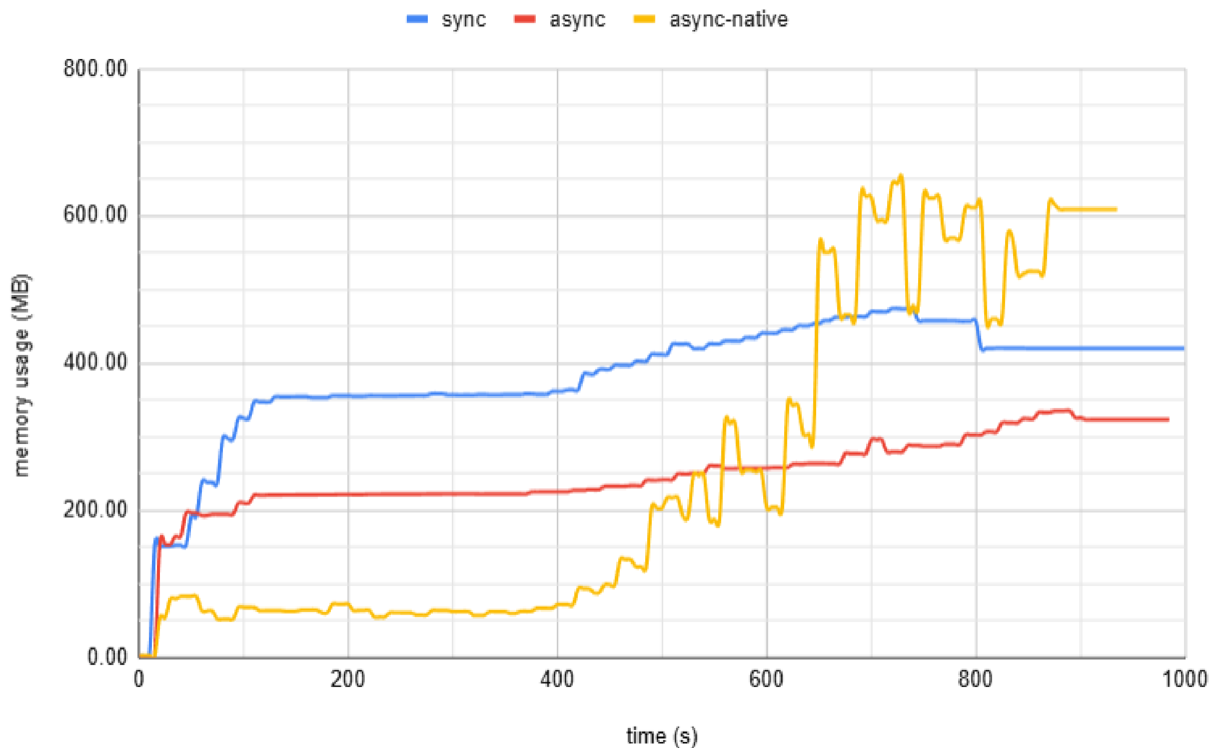


Figure 5.4: The memory usage of the benchmark application variants

of the JVM variants and peaks at roughly 650MB.

Figure 5.5 shows that the asynchronous variants had roughly the same request throughput and both reached considerably higher throughput as the synchronous JVM variant. With 500 VUs, all variants had identical throughput at roughly 790 req/s. But when the VUs are gradually increased, the throughput of the synchronous JVM variant is not going up. It stays at roughly 790 req/s and starts decreasing near the failure point. The asynchronous variants both reach a considerably higher throughput. When the VUs are gradually increased, the throughput of the asynchronous variants gradually go up and both reach the peak throughput at roughly 2800 req/s before failure.

The p95 latency of the asynchronous variants are similar, while the synchronous JVM variant has considerably higher latency as shown in Figure 5.6. With 500 VUs, the p95 latency of the asynchronous variants is near 100ms constantly, which is the

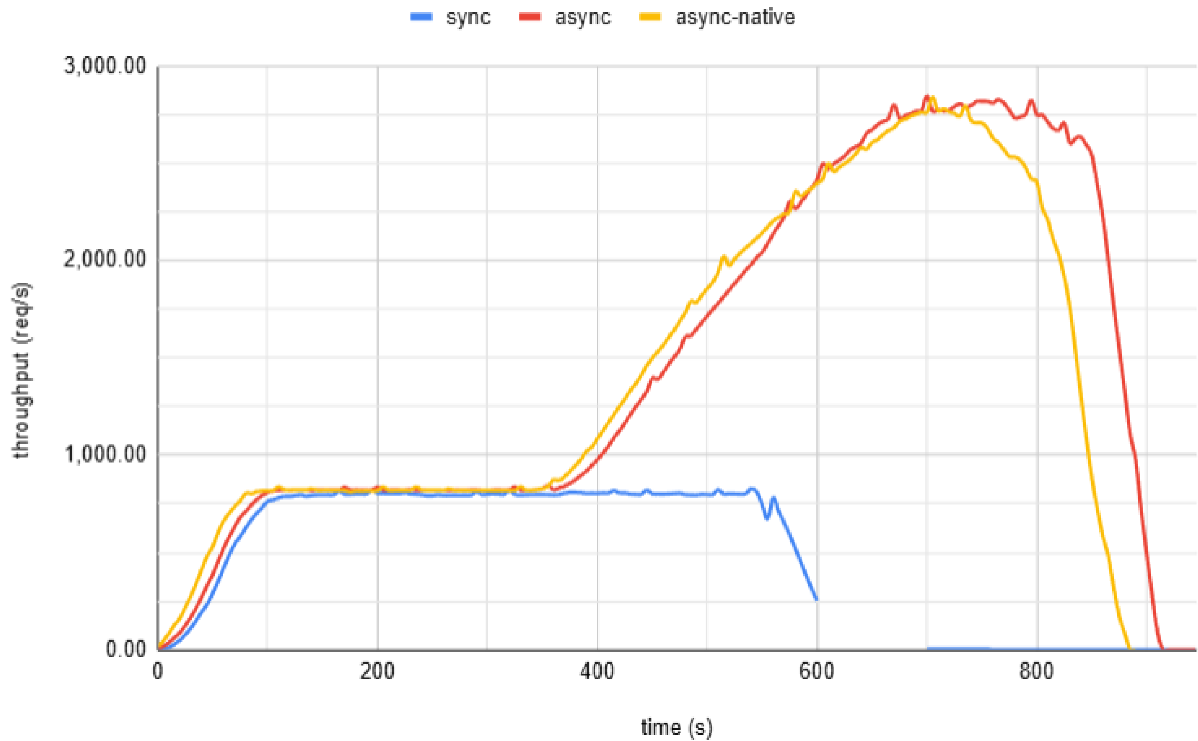


Figure 5.5: The request throughput of the benchmark application variants

artificial latency created by the stub service. The p95 latency of the synchronous variant with 500 VUs is roughly 130ms, which is a bit slower than the asynchronous variants. When the number of VUs is gradually increased before the 400 second time mark, the p95 latency of the synchronous variant increases more aggressively than the asynchronous variants and the p95 latency is more than thrice of the asynchronous variants. There is a gap of empty data for the synchronous variant after the 600 seconds time mark, which is the point when the synchronous JVM variant reached failure due to the p95 latency exceeding 2000ms. The p95 latency of the asynchronous variants when the number of VUs is gradually increased, but around the 750 seconds time mark the asynchronous native variant's p95 latency starts increasing slightly faster. Around the failure point of the asynchronous variants at 850 seconds, the p95 latency of the native variant is roughly 1700ms and the JVM variant's is roughly 900ms.

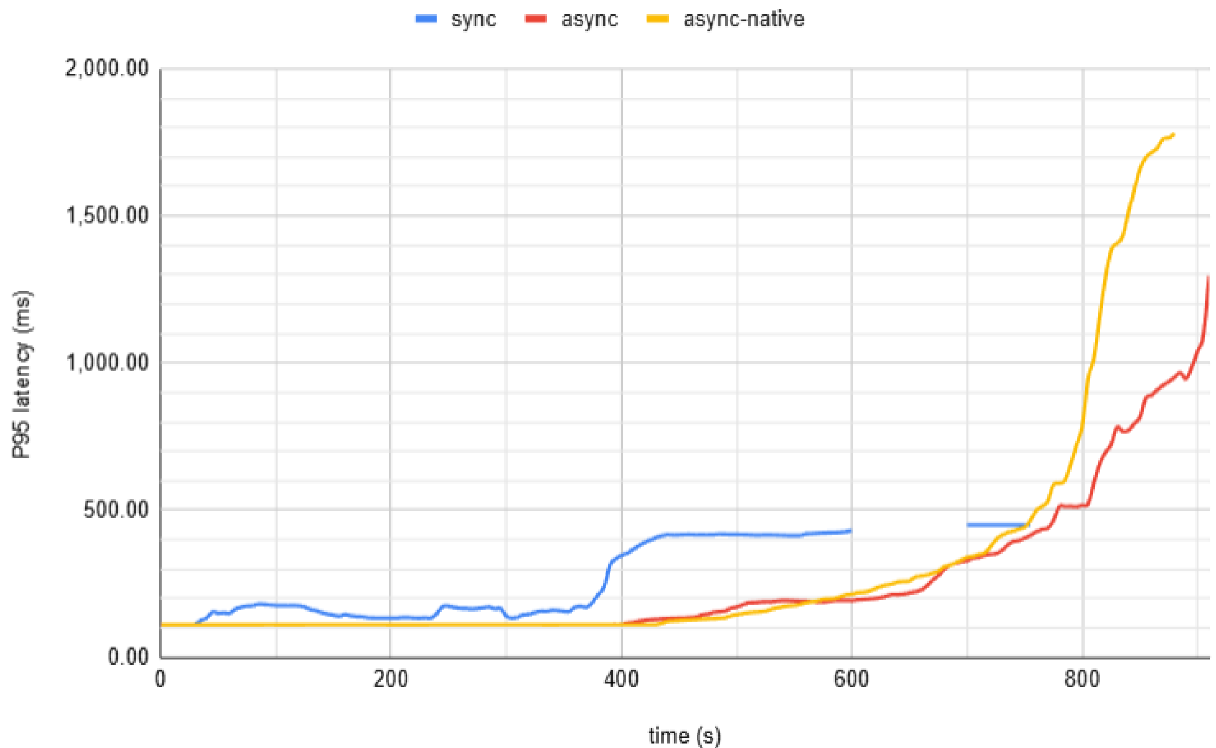


Figure 5.6: The p95 latency of the benchmark application variants

5.3.2 Analysis of Results

The results of **E2** in the previous section show that asynchronous concurrency and I/O has significant benefits when compared to the synchronous alternative. The synchronous JVM variant reached failure with 2179 virtual users after 700 seconds with an error rate of 0.0%. The asynchronous native variant reached failure with 2989 virtual users after 850 seconds and the asynchronous JVM variant reached failure with 3009 virtual users after 850 seconds while both of they had an error rate at roughly 1%. The asynchronous variants can potentially incur SLO violations at high load, since they had errors and the synchronous variant did not. However it is important to not that they did have better availability, because of their high throughput. And usually in production deployments, autoscaling would aim to prevent high stress, which leads to errors. However in bursty demands the application would be exposed to high stress, thus leading to errors. But it is also important to

note that during steady demand, the p95 latency of the asynchronous variants were almost identical to that of the artificial latency created inside the application. The synchronous variant added an overhead of 10 to 100ms to the p95 latency. Which might sound insignificant, but if in a microservices application with a long chain of services all services added overhead to the latency, the end to end latency would potentially violate SLOs.

The CPU usage was lowest with the asynchronous JVM variant. With 500 VUs the CPU usage was roughly 0.35 cores and the asynchronous native variant used roughly 0.50 cores. The CPU usage of the synchronous variant at 500 VUs already was using more than 0.90 cores. With asynchronous concurrency the CPU usage can be more than halved during steady demand, with the synchronous JVM variant having 61% less CPU usage. The more lightweight runtime of the native variant did not yield any benefits in CPU usage and it was a bit higher than the JVM alternative.

While the native variant did not yield benefits to CPU usage, it did yield benefits in memory usage. With 500 VUs the memory usage for the synchronous JVM variant was roughly 350MB, for the asynchronous JVM variant it was roughly 225MB and for the asynchronous native variant it was roughly 60MB, which is 83% smaller than the synchronous JVM variants memory usage. However, when the number of VUs were scaled up, the memory usage of the native variant quickly exceeded the JVM variants and reached over 650MB of memory usage. **E2** only investigated the Spring Boot framework and it is possible that the different native images of the Micronaut and Quarkus frameworks, specifically the distroless images, would have even less memory usage.

When the request throughput of the benchmark application variants were compared, the asynchronous variants had significantly higher maximum throughput before failure. Both of the asynchronous variants reached a throughput of roughly 2800

req/s. There was no difference between the asynchronous JVM and asynchronous native variants throughput. The synchronous JVM variant already reached its maximum throughput of roughly 790 req/s during the 5 minutes at 500 VUs. When the number of VUs were gradually increased, the throughput of the synchronous variant did not increase, meaning the requests were already queueing up. The results show that with asynchronous request handling, the request throughput can be increased by even 254%.

Asynchronous request handling also has significant benefits to latency. With 500 VUs the p95 latency of the asynchronous variants were almost identical at near 100ms. The synchronous variant incurs a slight overhead and the p95 latency is roughly 130ms at 500 VUs. When the number of VUs is gradually increased, the P95 latency is more than double than that of the asynchronous variants. Both the request throughput and the p95 latency of the synchronous JVM variant show that requests start queueing up. When the number of VUs are gradually increased, throughput does not go up and the p95 latency is increasing eventually to the point that there are too many requests and the p95 latency exceeds 2000ms. There is an empty gap of data for the synchronous variant around the time of failure. This means that the requests to the metrics endpoint of Prometheus are timing out. K6 measures the p95 latency differently and detects that the p95 latency is over 2000ms and flags a failure. The p95 latency of the asynchronous variants increase far slower and they never reach the 2000ms threshold. The native variant has similar p95 latency to the asynchronous JVM variant, after 750 seconds the latency increases more aggressively. Near the failure point the p95 latency for the native variant is already at 1700ms, which is 800ms higher than the asynchronous JVM variant.

Experiment **E2** showed that asynchronous request handling has significant benefits to CPU usage, memory usage, request throughput and latency. However, the asynchronous variants had error rates at roughly 1.0%, while the synchronous vari-

ant 0.0% during high stress. However, this is probably prevented in real production deployments by scaling the applications up, before they reach 90% CPU usage. Additionally, the lightweight runtime of a native image did not yield any significant benefits. The only metric that it seemed to improve was memory usage. It is also important to note, that the benchmark application was relatively small. In a real world application the magnitude of difference between memory usages could be quite different. Also the benchmark application did not do any CPU intensive work apart from request handling, instead every request delegated all work to the external stub service. In a real world application, a single request might require more intensive CPU work from the application and a request might fan out multiple requests to multiple external services. The benefits might be different in a real world application and depend on the nature of the application, but **E2** still indicates major improvements.

5.4 Experiment E3

5.4.1 Results

E3 evaluated the behaviour of four microservices application variants under a two hour load test. The goal was to observe how different levels of steady-state scalability manifested in a microservices deployment with inter-service dependencies, where requests propagated downstream. Each application variant consisted of 4 services, which were *ProductService*, *MetadataService*, *CatalogService* and *AttributeService*. The business logic and APIs are identical across variants. The services were deployed independently with identical resource allocations to AWS.

In Figure 5.7 we can see the CPU utilisation of each application variant in each service. The CPU utilisation goes up when a burst is injected and goes down shortly after as containers are scaled up due to high CPU utilisation. The graph shows the

CPU utilisation for all container replicas at the same time, and the steepness of the line graph shows how aggressively the containers use CPU. The figure shows that the CPU utilisation of the Micronaut optimised variant seems to be the lowest especially in *AttributeService*. The other variants seem to all have pretty similar CPU utilisation graphs and it hard to notice from the figure alone, which variant had the highest CPU utilisation. However, at times it seems that the CPU utilisation of the Spring Boot optimised variant seems to rise the fastest.

In Figure 5.8 it is clear that the memory usage of the Spring Boot baseline variant is the highest in all services except for *AttributeService* where Micronaut uses the most memory. Otherwise the optimised variants use similar amounts of memory. As the containers are scaled up when the memory utilisation or CPU utilisation reaches 70%, it can be said that the containers in this experiment are only scaled up because of high CPU utilisation since the memory utilisation does not reach 70% for any variant in any service.

Figure 5.9 shows the task count for each variant's service, which is increased when the resource usage of a service reaches the 70% threshold. The Spring Boot variants required the most tasks to match supply with demand and the Spring Boot optimised variant required the most out of all variants, while having over three times the tasks at times compared to the Micronaut optimised variant.

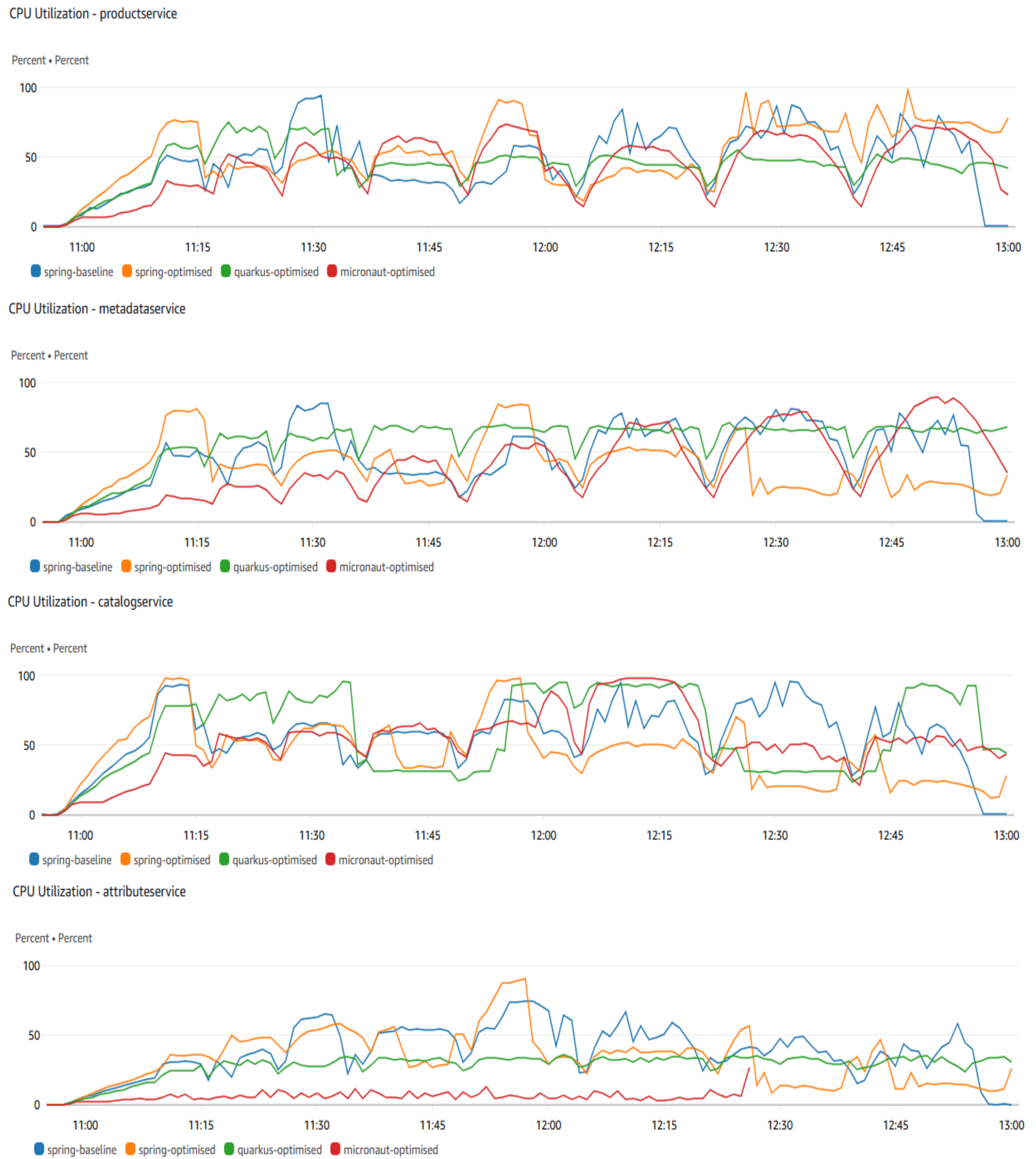
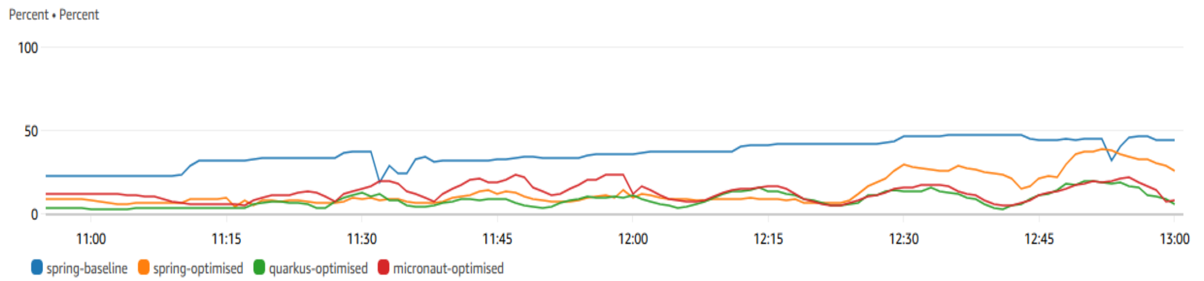
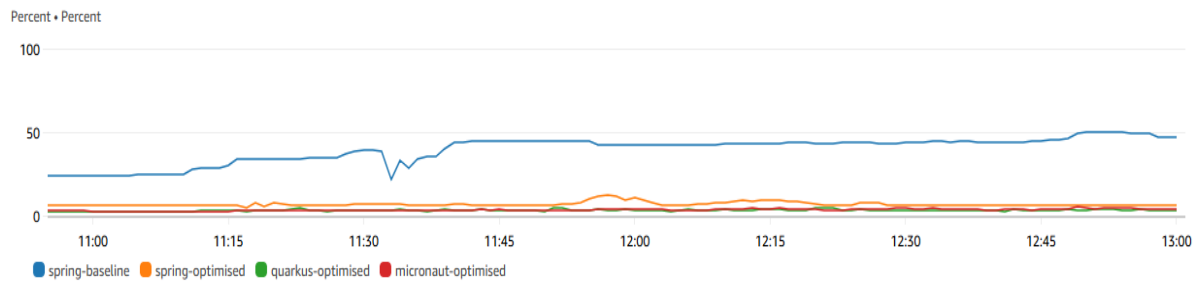


Figure 5.7: The CPU utilisation of each application variant

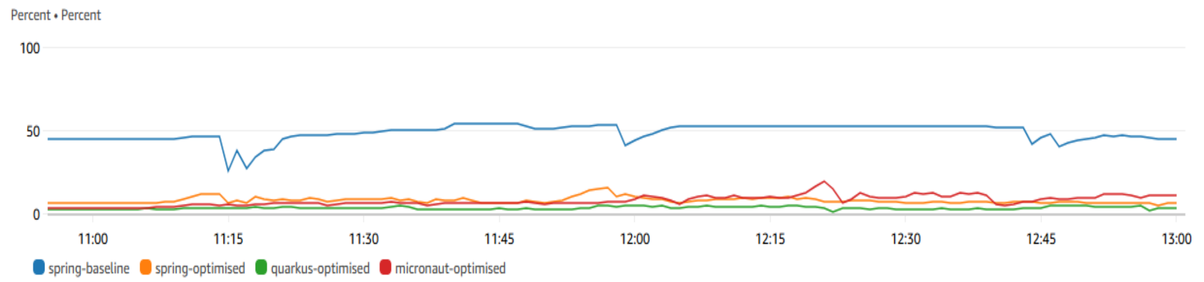
Memory Utilization - productservice



Memory Utilization - metadataservice



Memory Utilization - catalogservice



Memory Utilization - attributeservice

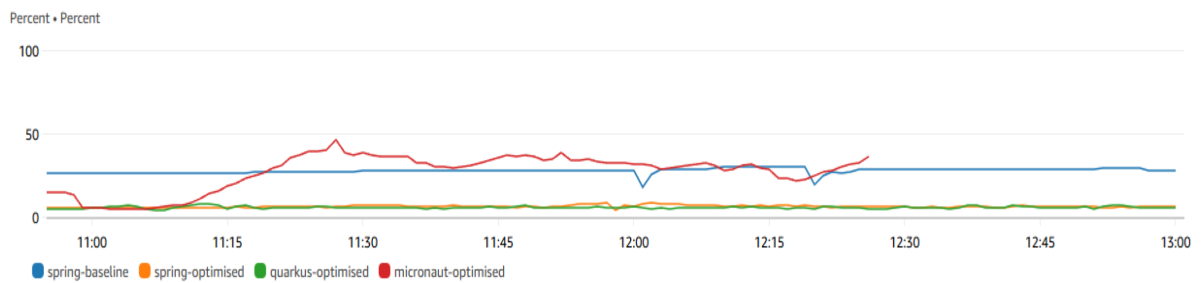


Figure 5.8: The memory utilisation of each application variant

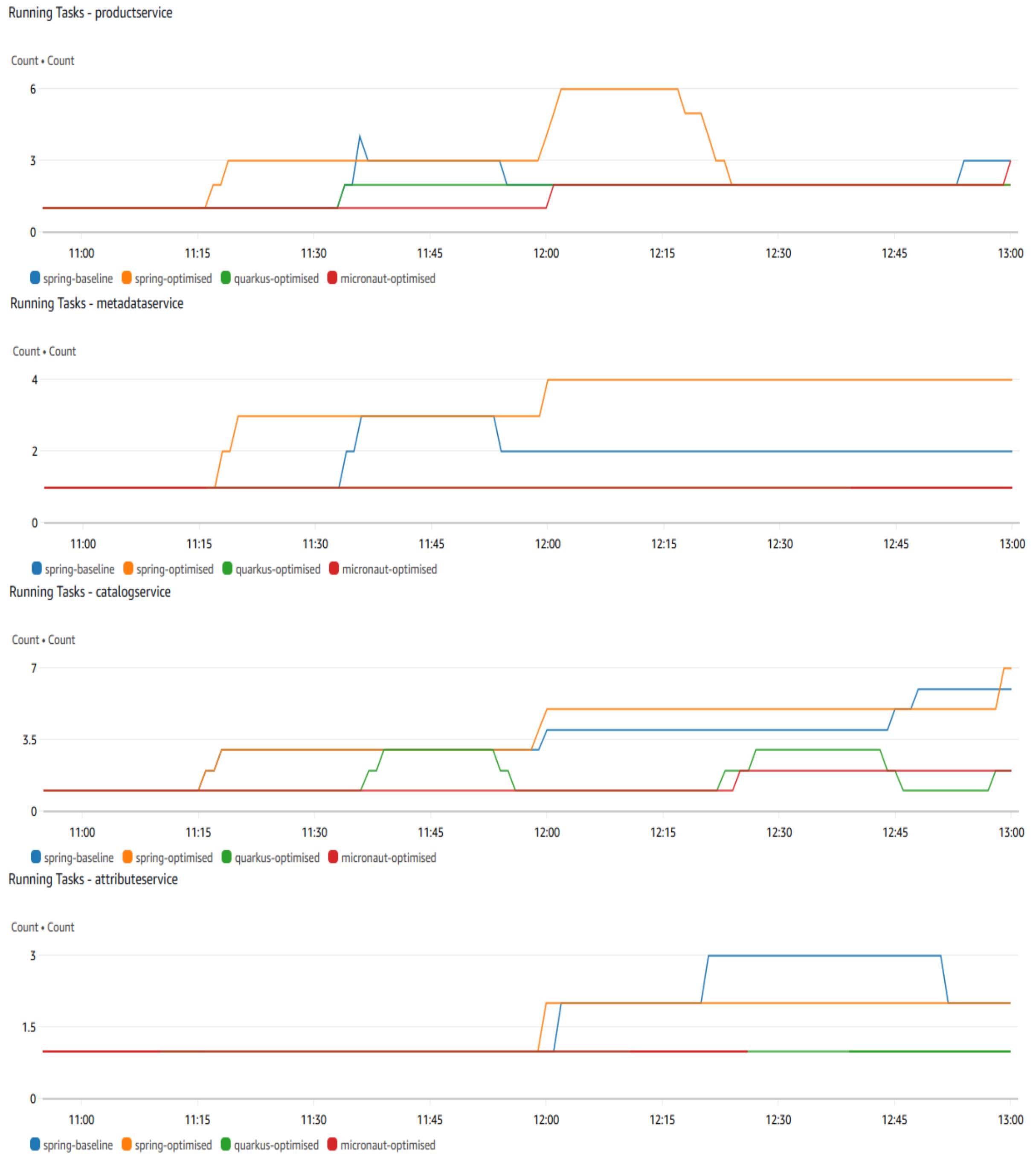


Figure 5.9: The task count of each application variant

The Figure 5.10 shows the cost differences between the application variants after the two hour load test. The resources were billed by the CPU and memory allocated to each container. The detailed pricing for the CPU usage was 0.04656$ per vCPU

per hour and for memory usage 0.00511$ per GB per hour. Each service container was allocated 2 CPUs and 4GB memory. The final costs for the Spring Boot baseline variant were 4.05$, for Spring Boot optimised 4.35$, for Quarkus optimised 2.16$ and for Micronaut optimised 1.95$.

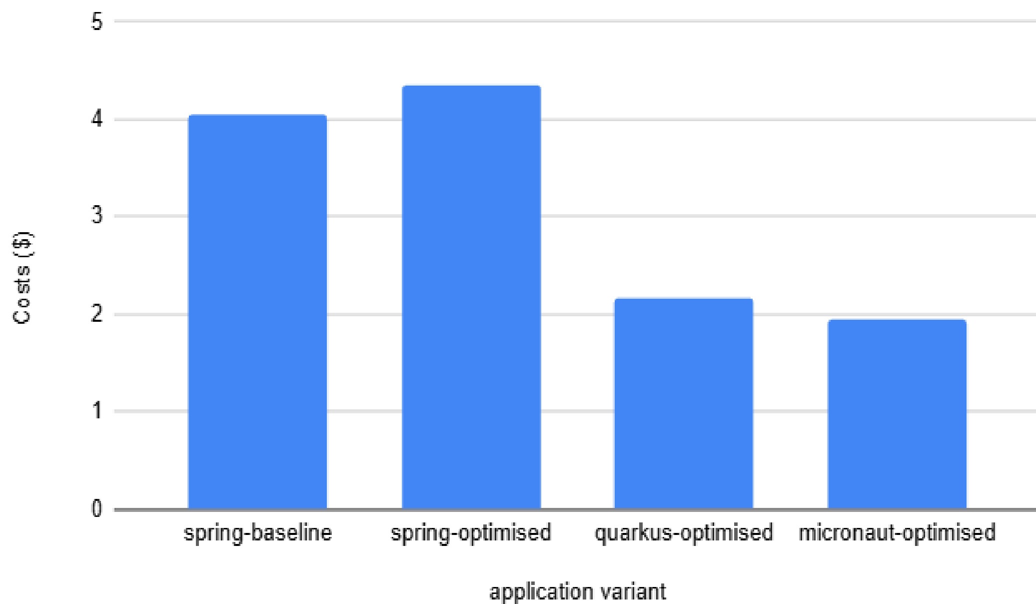


Figure 5.10: The cost of each application variant

5.4.2 Analysis of Results

E3 focused on RQ3 and investigated how differences in scalability affected resource usage and cloud costs. The experiment compared the resource usage and cloud costs of 4 different microservices application variants with various levels of scalability when deployed to AWS.

The CPU utilisation results show differences between the variants and the Micronaut optimised variant had the lowest CPU utilisation on average, but it is unclear from Figure 5.7 to, which variant had the highest CPU utilisation out of the variants. But it does seem that the Spring Boot optimised variant had the highest CPU utilisation, which is supported by the fact that it also had the highest task count. It

is unclear why this happens, since **E2** clearly demonstrated that Spring Boot with optimisation settings should use less CPU than with default settings. It might be possible that the autoscaling configurations in the experiment were not optimal for the variant, since at times the variant used a lot of CPU compared to the other variants, but at times it used half the CPU compared to other variants. This indicates that the containers were scaled up, but failed to scale down. But even this does not explain why the containers scaled so high in the first place.

The memory utilisation results showed that the Spring Boot baseline variant used the most memory with roughly 30-50% utilisation on average throughout the experiment. The optimised variants all used roughly 5-10% memory in all services, except for Micronaut optimised which had even 60% memory utilisation during a burst. The memory usage was independent of the load intensity, indicating that the CPU usage is the dominant cost driver and triggers scaling under load. Overall the results show that scalability optimisations can lead to roughly 4 times smaller memory usage.

The task count or container replica results represent the total resource usage of the application variants and they can be used to predict the total costs of running the application variants, since the resources were billed by the allocated CPU and memory for each task. The task count for the Spring Boot variants had the highest task count, which indicates they had the highest CPU utilisation. The Micronaut optimised and Quarkus optimised variants both had low task counts and Micronaut optimised had the lowest. As the CPU utilisation predicted, the task count of the Spring Boot optimised was higher than the Spring Boot baseline, which was surprising. At times the task count of Spring Boot optimised was even three times greater than the Micronaut optimised variant.

The cost results highlighted a significant benefit from improved scalability. After the two hour load test, the costs for the Spring Boot baseline were 4.05$, for Spring

Boot optimised were 4.35$, for Quarkus optimised 2.16$ and for Micronaut optimised 1.995$. As the task count predicted, the costs for the Spring Boot optimised were the highest and lowest for the Micronaut optimised.

E3 showed clear difference in scaling behaviour, resource usage and cloud costs. Two out of the three microservices applications with scalability optimisations lead to roughly 4 times smaller memory usage, roughly three times smaller CPU usage and roughly half the costs. But it is important to note that the optimised Spring Boot application had higher CPU usage and thus costs compared to the baseline. This conflicts with the results of **E2**, since it explicitly demonstrated that scalability optimisations in Spring Boot halved the CPU usage.

6 Discussion

This chapter discusses and synthesises the results of the literature review in Chapter 4 and the empirical scalability experiments in Chapter 5. The goal is to interpret the results in the context of the scalability of individual microservices and elasticity of microservices deployments to answer the research questions in Chapter 1. Instead of focusing on implementation technologies, the discussion in this chapter abstracts the results to generalisable scalability characteristics applicable to containerised microservices. Additionally, the limitations of this thesis are discussed and future work is proposed.

6.1 Operational Characteristics Driving Scalability (RQ1)

The results of the literature review and the scalability experiments showed that scalability is created by a combination of interrelated *operational characteristics*. These characteristics determine how well an individual microservice utilises computing resources and how well it supports elastic scaling in cloud environments. The results show that *startup time*, *image size*, *resource usage* and *throughput* are the most influential drivers of scalability and their combination determines the effectiveness of elastic scaling.

The most important *operational characteristics* for elastic scaling are *startup*

time and *image size*, since they both determine how fast a container replica can be provisioned. *Image size* determines how fast the image can be fetched from a registry to the node or the machine that is responsible for running the application and *startup time* determines when the application is ready for incoming traffic. The literature consistently mentioned the startup latency as a critical factor for elastic scaling. The experiments demonstrated that AOT and/or native compilation significantly reduced the *startup time* from seconds to milliseconds and the *image size* affected the time to fetch images to a node. Both of these factors can effectively lower the whole provisioning time of containers by seconds and that is an critical improvement especially with applications with bursty or unpredictable demand.

Resource usage, especially in terms of memory and CPU usage, was another important *operational characteristic* together with *throughput*. Microservices that require less resources allow for higher pod density, which improves the overall resource utilisation. Higher *throughput*, means microservices can do more with the same amount of resources. When these two characteristics are combined, it means that an individual service can do more with less. This means that services can handle increased demand before scaling out, thus reducing scaling actions, that add scaling latency or provisioning new containers, which add to the total resource usage. A microservice with bad *resource usage* and low *throughput* requires more scaling actions and container replicas to handle increased demand, thus introducing more scaling latency and costs. It was found that AOT compilation, native compilation, lightweight runtimes and asynchronous request handling were the key methods to to improve *resource usage* and *throughput*.

Importantly, the findings show that these operational characteristics do not act independently. *Startup time*, *image size*, *throughput* and *resource usage* are interrelated and reinforce each other. For example, a service with high *resource usage* has lower *throughput* per container, which means that it requires more scaling actions

adding more scaling latency. Microservices with low *resource usage*, high *throughput* and low overall provisioning time benefit from high effective capacity, smooth elastic scaling and lower costs.

6.2 Impact of Scalability on User-Perceived Performance (RQ2)

The user-perceived performance of server-side applications is primarily determined by response latency. In addition to latency, end-users can perceive timeouts or occasional failures, which are all captured by latency-focused SLIs. The literature review and scalability experiments showed that scalability has an impact on user-perceived performance, particularly under heavy or rapidly changing load.

Latency percentiles, such as p95 latency, emerged as the most sensitive indicators of scalability from the end-user's perspective. While average latency and latency percentiles can remain stable under moderate load, increased demand exposes inefficiencies in resource utilisation and request handling. When a microservice approaches their maximum capacity, resource contention increases and requests begin to queue up causing a rise in tail-latency. The results of the scalability experiments showed that a service with weaker scalability experienced a rise in p95 latency, well before total saturation occurred. Meaning that the service experienced noticeable overhead while remaining operational. In a microservices deployment with long service chains, these latencies add up and thus end-to-end latency can increase significantly due to weak scalability.

Scalability also affects how microservices behave during demand fluctuations as discussed in the previous section. In elastic cloud deployments, demand can be unpredictable and change often. Autoscaling requires time to provision new container replicas to match supply with demand. Services that are slow to provision introduce

scaling delays, during which the service is under-provisioned and the response times can increase considerably leading to latency spikes, response timeout or even response failures. Services with strong throughput, low resource usage and are fast to provision, reduce the duration and severity of these latency spikes or even prevent them overall, thus resulting in a smoother user experience overall during scaling events.

6.3 Impact of Scalability on Cloud Costs and SLA Adherence (RQ3)

Scalability of microservices has a direct and measurable impact to cloud costs and SLA adherence. The economic outcomes of scalability are primarily dictated by how well microservices utilise resources to maintain the required throughput. Both the literature review and the scalability experiments showed that poor scalability increases recurring operational costs and forces trade-offs between cost efficiency and SLA adherence, while strong scalability enables predictable and cost-effective operations.

One of the primary cost drivers associated with scalability is resource efficiency. Services with lower effective capacity per instance, determined by *throughput* and *resource usage*, require a greater number of container replicas to sustain acceptable performance levels. This leads to increased overall resource usage, namely CPU and memory usage. As a result, recurring infrastructure costs grow aggressively with load. The findings from the scalability experiments support findings from the literature, that low scalability leads to underutilised resources and reduced packing efficiency. Both of these directly increase TCO in cloud environments. The scalability experiments found that improved scalability lead to over two times smaller computing costs.

The provisioning times of containers also has a significant role in cloud costs. Microservices with slow provisioning times amplify the risks of under-provisioning during scaling actions. This encourages CSUs to maintain a level of over-provisioning as a safety margin to mitigate latency spikes and SLO violations, while increasing the infrastructure costs. In contrast, microservices with fast provisioning times and high effective capacity reduce required scaling actions and time spent under-provisioned. This allows for much tighter capacity planning and more aggressive scale-down policies, thus lowering infrastructure costs while maintaining SLA adherence.

6.4 Limitations

This thesis provides several insights into the effects of scalability to user-perceived performance, cloud costs and how to optimise microservices for scalability. However, several limitations and methodological choices must be acknowledged.

An important limitation of the scalability experiments is their synthetic nature. The experiments were positivist in nature, meaning that they were designed to be controlled and repeatable in order to isolate the effects of scalability related characteristics. While this approach worked well for systematic and clear interpretation of the results, it does not reflect the complexity and variability of real-world production deployments.

Additionally, some variability and inconsistent results were observed in some of the scalability experiments. Differences in observed orchestration overhead were observed even in scenarios, where there should not be any variability in Section 5.2. And in Section 5.4 a scalability optimised application variant performed worse than the baseline variant, which conflicts with the results of Section 5.3. The underlying causes of these inconsistencies could not be determined within the scope of this thesis.

Another limitation is the technological scope of the thesis. The scalability ex-

periments were conducted using JVM-based microservice frameworks, which were selected to present various levels of scalability. While the research questions and analysis of results aimed to be framework and language agnostic, the results may not generalise to microservices implemented with different programming languages or runtimes, since different ecosystems may have distinct startup behaviours and concurrency models that could influence scalability in different ways and the magnitude of benefits might differ.

The chosen cloud environment in Section 5.4 also introduces constraints on generalisability. The scaling behaviours and infrastructure differs between cloud providers and the cost model also differs even inside the chosen cloud provider. This fact undermines the generalisability of the results in Section 5.4. Additionally, autoscaling strategies were limited to default reactive strategies, which might not reflect the behaviour of modern production systems with advanced predictive scaling strategies.

Additionally, the analysis of cloud costs and TCO relied entirely on simplified cost models that focused on computing costs. This approach worked well for analysing the cost implications of scalability, but it does not account for fixed infrastructure costs or indirect costs from development or long-term maintenance.

Despite these limitations, the methodology of this thesis does provide a systematic foundation for analysing the relationship between scalability, user-perceived performance and cloud costs. The limitations discussed in this section does not invalidate the findings, but rather indicates where caution is required when applying the findings of this thesis.

6.5 Future Work

The findings of this thesis together with the discussed limitations in the previous section highlight multiple directions for future research. The presented results focused on general scalability characteristics in controlled environments, and further

studies could broaden the scope to real-world production deployments.

One important direction for future work is the inclusion of additional programming languages and runtime ecosystems. Extending the analysis from the JVM ecosystem to include microservices implemented in other ecosystems, such as Python, Node.js or Go, would allow more extensive comparison of startup behaviour, resource efficiency, latency characteristics and concurrency models. Such studies could help investigate the generalisability of the identified scalability drivers.

Future research could also investigate more realistic and heterogeneous workloads. A case study of optimising a real-world production system could provide deeper insight into how the different *operational characteristics* drive scalability and what is the magnitude of benefits. Future research with a more realistic workload could also combine the scalability optimisation with alternative autoscaling strategies, particularly with predictive autoscaling.

Future research could also refine the economic analysis by having more extensive cost models. Including additional cost factors such as storage, databases, networking, observability, development and maintenance would provide a more holistic view of TCO. Also evaluating the impacts of scalability across multiple cloud providers and pricing models could strengthen the generalisability of the results.

7 Conclusion

The research problem addressed in this thesis was to analyse how different levels of scalability in microservices affect user-perceived performance and cloud costs. This thesis examined scalability as a general trait of systems relevant to containerised microservices deployed to cloud environments.

To address this problem, this thesis combined a structured literature review with a conceptual analysis of microservice implementation technologies and controlled empirical experiments conducted in a container orchestration environment. The conceptual analysis investigated what kind of *operational characteristics* microservice frameworks target for improved scalability. The literature review examined scalability concepts, mechanisms and implications. The controlled experiments evaluated different microservices benchmark applications with various levels of scalability in order to observe how the different *operational characteristics* manifested under load. JVM-based microservice frameworks were used to compare implications of scalability in a common ecosystem, while the analysis and conclusions remained framework and language agnostic.

The first research question examined which *operational characteristics* drive scalability in containerised microservices. The results showed that scalability is driven by *startup time*, *image size*, *throughput* and *resource usage*. These characteristics are interrelated and scalability manifests from their combined effect rather than from isolated optimisations. The literature review found multiple commonly used

methods for optimising scalability, such as AOT optimisation, native compilation, lightweight runtimes and asynchronous request handling. The empirical experiments showed that these methods can be used to improve scalability in the used workloads.

The second research question examined how scalability affects user-perceived performance. Poor scalability manifested as increased tail-latency, latency spikes during scale-out and unpredictable response times. The literature review results showed that scalability optimisations can lead to up to 60% reduced average latency and several hundred milliseconds reductions in p95 latency in some contexts. The empirical experiments showed that poor scalability can incur additional latency, which may be amplified in a microservices deployment with long service chains.

The third research question investigated the implications from improved scalability to cloud costs and SLA adherence. The results showed that poor scalability leads to increased computing costs in a cloud environment due to inefficient resource utilisation and the need for over-provisioning resources. In contrast, microservices with higher scalability utilise resources more effectively and have less or no need for over-provisioning, which leads to smaller resource usage and smaller costs. The empirical experiments demonstrated that improved scalability leads to even two times smaller computing costs. Also, since improved scalability brings more stable response times, less required scale-out actions and overall smaller end-to-end latency, SLO violations are reduced.

The research outcomes of this thesis demonstrate that scalability plays a central role in user-perceived performance and in cloud costs in cloud-native microservices. The results showed that scalability of individual microservices should be treated as an important design goal in microservices, since it has a direct impact to user experience and business outcomes. By analysing scalability from technical and economical perspectives, this thesis adds to a more holistic understanding of microservice behaviour in high demand workloads in elastic cloud environments.

References

[1] M. S. Hamzehloui, S. Sahibuddin, and K. Salah, "A systematic mapping study on microservices", en, F. Saeed, N. Gazem, F. Mohammed, and A. Busalim, Eds., Cham: Springer International Publishing, 2019, pp. 1079–1090, ISBN: 978-3-319-99007-1. DOI: 10.1007/978-3-319-99007-1_100.

[2] F. Aydemir and F. Başçiftçi, "Building a performance efficient core banking system based on the microservices architecture", en, *Journal of Grid Computing*, vol. 20, no. 4, p. 37, Nov. 2022, ISSN: 1572-9184. DOI: 10.1007/s10723-022-09624-z.

[3] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A kubernetes controller for managing the availability of elastic microservice based stateful applications", English, *JOURNAL OF SYSTEMS AND SOFTWARE*, vol. 175, p. 110 924, May 2021, Web of Science ID: WOS:000623099500013, ISSN: 0164-1212, 1873-1228. DOI: 10.1016/j.jss.2021.110924.

[4] G. Galante and L. C. E. de Bona, "A survey on cloud computing elasticity", in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, Nov. 2012, pp. 263–270. DOI: 10.1109/UCC.2012.30. [Online]. Available: https://ieeexplore.ieee.org/document/6424959.

[5] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza, "Elasticity in cloud computing: A survey", en, *annals of*

telecommunications - annales des télécommunications, vol. 70, no. 7, pp. 289–309, Aug. 2015, ISSN: 1958-9395. DOI: 10.1007/s12243-014-0450-7.

[6] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges", *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, Mar. 2018, ISSN: 1939-1374. DOI: 10.1109/TSC.2017.2711009.

[7] Ł. Wyciślik, Ł. Latusik, and A. M. Kamińska, "A comparative assessment of jvm frameworks to develop microservices", en, *Applied Sciences*, vol. 13, no. 33, p. 1343, Jan. 2023, ISSN: 2076-3417. DOI: 10.3390/app13031343.

[8] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*, en. Sep. 2011. DOI: 10.6028/NIST.SP.800-145. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/145/final.

[9] Y. Liu and T. Liimatainen, "Technology diffusion of cloud computing", vol. 5, no. 1, pp. 80–87, Mar. 2014.

[10] S. Zhang, S. Zhang, X. Chen, and X. Huo, "Cloud computing research and development trend", in *2010 Second International Conference on Future Networks*, Jan. 2010, pp. 93–97. DOI: 10.1109/ICFN.2010.58. [Online]. Available: https://ieeexplore.ieee.org/document/5431874.

[11] R. Younis, M. Iqbal, K. Munir, M. A. Javed, M. Haris, and S. Alahmari, "A comprehensive analysis of cloud service models: Iaas, paas, and saas in the context of emerging technologies and trend", in *2024 International Conference on Electrical, Communication and Computer Engineering (ICECCE)*, Oct. 2024, pp. 1–6. DOI: 10.1109/ICECCE63537.2024.10823401. [Online]. Available: https://ieeexplore.ieee.org/document/10823401.

[12] M. Salas-Zarate and L. Colombo-Mendoza, "Cloud computing: A review of paas, iaas, saas services and providers", English, *REVISTA DIGITAL LAMP-*

SAKOS, no. 7, pp. 47–57, Jun. 2012, Web of Science ID: WOS:000215658000006, ISSN: 2145-4086.

[13] M. Saraswat and R. Tripathi, "Cloud computing: Analysis of top 5 csps in saas, paas and iaas platforms", Dec. 2020, pp. 300–305. DOI: 10.1109/SMART50582.2020.9337157. [Online]. Available: https://ieeexplore.ieee.org/document/9337157.

[14] J. Dogani, R. Namvar, and F. Khunjush, "Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey", *Computer Communications*, vol. 209, pp. 120–150, Sep. 2023, ISSN: 0140-3664. DOI: 10.1016/j.comcom.2023.06.010.

[15] M.-N. Tran, D.-D. Vu, and Y. Kim, "A survey of autoscaling in kubernetes", in *2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2022, pp. 263–265. DOI: 10.1109/ICUFN55119.2022.9829572. [Online]. Available: https://ieeexplore.ieee.org/document/9829572.

[16] F. Qazi, D. Kwak, F. G. Khan, F. Ali, and S. U. Khan, "Service level agreement in cloud computing: Taxonomy, prospects, and challenges", *Internet of Things*, vol. 25, p. 101 126, Apr. 2024, ISSN: 2542-6605. DOI: 10.1016/j.iot.2024.101126.

[17] J. He and L. Sun, "A review on sla-related applications in cloud computing", in *2018 1st International Cognitive Cities Conference (IC3)*, Aug. 2018, pp. 87–91. DOI: 10.1109/IC3.2018.00027. [Online]. Available: https://ieeexplore.ieee.org/document/8567173.

[18] F. Schulz, "Elasticity in service level agreements", in *2013 IEEE International Conference on Systems, Man, and Cybernetics*, Oct. 2013, pp. 4092–4097. DOI: 10.1109/SMC.2013.698. [Online]. Available: https://ieeexplore.ieee.org/document/6722451.

[19] P. Rosati and T. Lynn, "Measuring the business value of infrastructure migration to the cloud", en, in *Measuring the Business Value of Cloud Computing*, T. Lynn, J. G. Mooney, P. Rosati, and G. Fox, Eds. Cham: Springer International Publishing, 2020, pp. 19–37, ISBN: 978-3-030-43198-3. DOI: 10.1007/978-3-030-43198-3_2. [Online]. Available: https://doi.org/10.1007/978-3-030-43198-3_2.

[20] P. Rosati, F. Fowley, C. Pahl, D. Taibi, and T. Lynn, "Right scaling for right pricing: A case study on total cost of ownership measurement for cloud migration", in 2019, vol. 1073, pp. 190–214, arXiv:1908.04136 [cs]. DOI: 10.1007/978-3-030-29193-8_10. [Online]. Available: http://arxiv.org/abs/1908.04136.

[21] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead", *IEEE Software*, vol. 35, no. 3, pp. 24–35, May 2018, ISSN: 1937-4194. DOI: 10.1109/MS.2018.2141039.

[22] N. Dragoni et al., "Microservices: Yesterday, today, and tomorrow", en, in *Present and Ulterior Software Engineering*, M. Mazzara and B. Meyer, Eds. Cham: Springer International Publishing, 2017, pp. 195–216, ISBN: 978-3-319-67425-4. DOI: 10.1007/978-3-319-67425-4_12. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12.

[23] G. Blinowski, A. Ojdowska, and A. Przybyłek, "Monolithic vs. microservice architecture: A performance and scalability evaluation", *IEEE Access*, vol. 10, pp. 20 357–20 374, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152803.

[24] J. Lewis and M. Fowler. "Microservices: A definition of this new architectural term", Accessed: Dec. 15, 2025. [Online]. Available: https://martinfowler.com/articles/microservices.html.

[25] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges", *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3305687.

[26] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review", in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, pp. 1–7. DOI: 10.1109/SCCC49216.2019.8966423.

[27] L. Weerasinghe and I. Perera, "Evaluating the inter-service communication on microservice architecture", in *2022 7th International Conference on Information Technology Research (ICITR)*, Dec. 2022, pp. 1–6. DOI: 10.1109/ICITR57877.2022.9992918. [Online]. Available: https://ieeexplore.ieee.org/document/9992918.

[28] B. Çiftçi and B. Çiloğlugil, "A review of comparative studies on performance evaluation of communication mechanisms for microservices", en, in *Computational Science and Its Applications – ICCSA 2025*, O. Gervasi et al., Eds., Cham: Springer Nature Switzerland, 2025, pp. 98–113, ISBN: 978-3-031-96962-1. DOI: 10.1007/978-3-031-96962-1_7.

[29] K. Adrio, C. N. Tanzil, M. C. Lianto, and Z. E. Rasjid, "Comparative analysis of monolith, microservice api gateway and microservice federated gateway on web-based application using graphql api", in *2023 10th International Conference on Electrical Engineering, Computer Science and Informatics (EECSI)*, 2023, pp. 654–660. DOI: 10.1109/EECSI59885.2023.10295809. [Online]. Available: https://ieeexplore.ieee.org/document/10295809.

[30] A. Hlybovets and I. Paprotskyi, "Increasing the fault tolerance in microservice architecture", en, *Cybernetics and Systems Analysis*, vol. 60, no. 3, pp. 480–488, May 2024, ISSN: 1573-8337. DOI: 10.1007/s10559-024-00689-0.

[31] H. Hanada and K. Ishibashi, "Empirical study on request timeout and retry for microservices communication", in *2024 IEEE 29th Pacific Rim International Symposium on Dependable Computing (PRDC)*, Nov. 2024, pp. 199–200. DOI: 10.1109/PRDC63035.2024.00037. [Online]. Available: https://ieeexplore.ieee.org/document/10859000.

[32] S. Verreydt, E. H. Beni, E. Truyen, B. Lagaisse, and W. Joosen, "Leveraging kubernetes for adaptive and cost-efficient resource management", in *Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, ser. WOC '19, New York, NY, USA: Association for Computing Machinery, Dec. 2019, pp. 37–42, ISBN: 978-1-4503-7033-2. DOI: 10.1145/3366615.3368357. [Online]. Available: https://dl.acm.org/doi/10.1145/3366615.3368357.

[33] A. Nõu, S. Talluri, A. Iosup, and D. Bonetta, "Investigating performance overhead of distributed tracing in microservices and serverless systems", in *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '25, New York, NY, USA: Association for Computing Machinery, May 2025, pp. 162–166, ISBN: 979-8-4007-1130-5. DOI: 10.1145/3680256.3721316. [Online]. Available: https://dl.acm.org/doi/10.1145/3680256.3721316.

[34] P. Sharma, L. Chaufournier, P. Shenoy, and Y. C. Tay, "Containers and virtual machines at scale: A comparative study", in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16, New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 1–13, ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988337. [Online]. Available: https://dl.acm.org/doi/10.1145/2988336.2988337.

[35] I. Mavridis and H. Karatza, "Performance and overhead study of containers running on top of virtual machines", in *2017 IEEE 19th Conference on Busi-*

ness Informatics (CBI), vol. 02, 2017, pp. 32–38. DOI: 10.1109/CBI.2017.69. [Online]. Available: https://ieeexplore.ieee.org/document/8012937.

[36] Accessed: Jul. 25, 2025. [Online]. Available: https://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment.

[37] S. Fu, J. Liu, X. Chu, and Y. Hu, "Toward a standard interface for cloud providers: The container as the narrow waist", *IEEE Internet Computing*, vol. 20, no. 2, pp. 66–71, Mar. 2016, ISSN: 1941-0131. DOI: 10.1109/MIC.2016.25.

[38] A. Malviya and R. K. Dwivedi, "A comparative analysis of container orchestration tools in cloud computing", in *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, Mar. 2022, pp. 698–703. DOI: 10.23919/INDIACom54597.2022.9763171. [Online]. Available: https://ieeexplore.ieee.org/document/9763171.

[39] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/autoscaling/.

[40] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/.

[41] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[42] R. Furnadzhiev, "An experimental evaluation of latency-aware scheduling for distributed kubernetes clusters", *Engineering Proceedings*, vol. 100, no. 11, p. 25, 2025, ISSN: 2673-4591. DOI: 10.3390/engproc2025100025.

[43] N. Zhao et al., "Large-scale analysis of docker images and performance implications for container storage systems", *IEEE Transactions on Parallel and*

Distributed Systems, vol. 32, no. 4, pp. 918–930, Apr. 2021, ISSN: 1558-2183. DOI: 10.1109/TPDS.2020.3034517.

[44] M. Straesser et al., "An empirical study of container image configurations and their impact on start times", in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, May 2023, pp. 94–105. DOI: 10.1109/CCGrid57682.2023.00019. [Online]. Available: https://ieeexplore.ieee.org/document/10171550.

[45] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Slacker: Fast distribution with lazy docker containers", in *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, ser. FAST'16, USA: USENIX Association, Feb. 2016, pp. 181–195, ISBN: 978-1-931971-28-7.

[46] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[47] "Autoscaler/vertical-pod-autoscaler at 9f87b78df0f1d6e142234bb32e8acbd71295585a · kubernetes/autoscaler", Accessed: Jul. 25, 2025. [Online]. Available: https://github.com/kubernetes/autoscaler/tree/9f87b78df0f1d6e142234bb32e8acbd71 vertical-pod-autoscaler.

[48] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/.

[49] V. Dakić, J. Redžepagić, M. Bašić, and L. Žgrablić, "Performance and latency efficiency evaluation of kubernetes container network interfaces for built-in and custom tuned profiles", en, *Electronics*, vol. 13, no. 1919, p. 3972, Jan. 2024, ISSN: 2079-9292. DOI: 10.3390/electronics13193972.

[50] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/storage/persistent-volumes/.

[51] L. Mercl and J. Pavlik, "Public cloud kubernetes storage performance analysis", en, in *Computational Collective Intelligence*, N. T. Nguyen, R. Chbeir, E. Exposito, P. Aniorté, and B. Trawiński, Eds., Cham: Springer International Publishing, 2019, pp. 649–660, ISBN: 978-3-030-28374-2. DOI: 10.1007/978-3-030-28374-2_56.

[52] Accessed: Jul. 25, 2025. [Online]. Available: https://kubernetes.io/docs/concepts/security/.

[53] Z. Morić, V. Dakić, and T. Čavala, "Security hardening and compliance assessment of kubernetes control plane and workloads", en, *Journal of Cybersecurity and Privacy*, vol. 5, no. 22, p. 30, 2025, ISSN: 2624-800X. DOI: 10.3390/jcp5020030.

[54] A. B. Bondi, "Characteristics of scalability and their impact on performance", in *Proceedings of the 2nd international workshop on Software and performance*, ser. WOSP '00, New York, NY, USA: Association for Computing Machinery, 2000, pp. 195–203, ISBN: 978-1-58113-195-6. DOI: 10.1145/350391.350432. [Online]. Available: https://dl.acm.org/doi/10.1145/350391.350432.

[55] J. Dean and L. A. Barroso, "The tail at scale", *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, ISSN: 0001-0782. DOI: 10.1145/2408776.2408794.

[56] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications", no. arXiv:1901.04946, Jan. 2019, arXiv:1901.04946 [cs]. DOI: 10.48550/arXiv.1901.04946. [Online]. Available: http://arxiv.org/abs/1901.04946.

[57] Accessed: Sep. 16, 2025. [Online]. Available: https://survey.stackoverflow.co/2025/technology#most-popular-technologies-language.

[58] D. Lu, J. Wu, Y. Sheng, P. Liu, and M. Yang, "Analysis of the popularity of programming languages in open source software communities", in *2020 International Conference on Big Data and Social Sciences (ICBDSS)*, Aug. 2020, pp. 111–114. DOI: 10.1109/ICBDSS51270.2020.00033. [Online]. Available: https://ieeexplore.ieee.org/document/9434501.

[59] Ł. Wyciślik, Ł. Latusik, and A. M. Kamińska, "A comparative assessment of jvm frameworks to develop microservices", en, *Applied Sciences*, vol. 13, no. 3, p. 1343, Jan. 2023, ISSN: 2076-3417. DOI: 10.3390/app13031343.

[60] M. Viggiato, R. Terra, H. Rocha, M. T. Valente, and E. Figueiredo, "Microservices in practice: A survey study", no. arXiv:1808.04836, Aug. 2018, arXiv:1808.04836 [cs]. DOI: 10.48550/arXiv.1808.04836. [Online]. Available: http://arxiv.org/abs/1808.04836.

[61] S. Gehwolf. "Java 17: What's new in openjdk's container awareness", Accessed: Sep. 16, 2025. [Online]. Available: https://developers.redhat.com/articles/2022/04/19/java-17-whats-new-openjdks-container-awareness.

[62] M. Basso, A. Prokopec, A. Rosà, and W. Binder, "Improving native-image startup performance", in *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO '25, New York, NY, USA: Association for Computing Machinery, Mar. 2025, pp. 689–703, ISBN: 979-8-4007-1275-3. DOI: 10.1145/3696443.3708927. [Online]. Available: https://dl.acm.org/doi/10.1145/3696443.3708927.

[63] C. Wimmer et al., "Initialize once, start fast: Application initialization at build time", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 184:1–184:29, Oct. 2019. DOI: 10.1145/3360610.

[64] P. Plecinski, N. Bokla, T. Klymkovych, M. Melnyk, and W. Zabierowski, "Comparison of representative microservices technologies in terms of performance for use for projects based on sensor networks", en, *Sensors*, vol. 22, no. 20, p. 7759, Jan. 2022, ISSN: 1424-8220. DOI: 10.3390/s22207759.

[65] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/performance/.

[66] Accessed: Sep. 27, 2025. [Online]. Available: https://docs.micronaut.io/latest/guide/.

[67] Accessed: Sep. 25, 2025. [Online]. Available: https://docs.spring.io/spring-boot/index.html.

[68] Accessed: Sep. 25, 2025. [Online]. Available: https://docs.spring.io/spring-framework/reference/.

[69] Accessed: Sep. 25, 2025. [Online]. Available: https://www.graalvm.org/22.1/reference-manual/native-image/Reflection/.

[70] Accessed: Sep. 25, 2025. [Online]. Available: https://docs.spring.io/spring-boot/reference/.

[71] Accessed: Sep. 25, 2025. [Online]. Available: https://jakarta.ee/specifications/servlet/5.0/jakarta-servlet-spec-5.0.

[72] Accessed: Sep. 25, 2025. [Online]. Available: https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html.

[73] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/guides/cdi-reference.

[74] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/guides/writing-extensions.

[75] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/versatility/.

[76] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/guides/quarkus-reactive-architecture.

[77] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/container-first/.

[78] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/kubernetes-native/.

[79] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/guides/maven-tooling.

[80] Accessed: Sep. 27, 2025. [Online]. Available: https://quarkus.io/guides/building-native-image.

[81] Accessed: Sep. 27, 2025. [Online]. Available: https://micronaut-projects.github.io/micronaut-data/latest/guide/.

[82] Accessed: Sep. 27, 2025. [Online]. Available: https://micronaut-projects.github.io/micronaut-aot/latest/guide/.

[83] Accessed: Sep. 27, 2025. [Online]. Available: https://guides.micronaut.io/latest/micronaut-docker-image-gradle-kotlin.html.

[84] Accessed: Sep. 27, 2025. [Online]. Available: https://guides.micronaut.io/latest/micronaut-creating-first-graal-app-gradle-java.html.

[85] Q. Wang, X. Gu, and C. Pu, "A study of response time instability of microservices at high resource utilization in the cloud", in *2024 IEEE 6th International Conference on Cognitive Machine Intelligence (CogMI)*, Oct. 2024,

pp. 111–116. DOI: 10.1109/CogMI62246.2024.00024. [Online]. Available: https://ieeexplore.ieee.org/document/10835545.

[86] O. Zanevych, “Advancing web development: A comparative analysis of modern frameworks for rest and graphql back-end services”, en, *Grail of Science*, no. 37, pp. 216–228, Mar. 2024, ISSN: 2710-3056. DOI: 10.36074/grail-of-science.15.03.2024.031.

[87] S. Sarkar, A. PP, and S. Ramaswamy, “Analysis, evaluation, and assessment for containerizing an industry automation software”, in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2020, pp. 1972–1979. DOI: 10.1109/SMC42975.2020.9282840. [Online]. Available: https://ieeexplore.ieee.org/document/9282840.

[88] A. Rahmatulloh, F. Nugraha, R. Gunawan, I. Darmawan, E. Haerani, and R. Rizal, “Event-driven architecture (eda) vs api-driven architecture (ada): Which performs better in microservices?”, in *2024 International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*, Aug. 2024, pp. 31–36. DOI: 10.1109/ICoABCD63526.2024.10704326. [Online]. Available: https://ieeexplore.ieee.org/document/10704326.

[89] F. H. L. Buzato and A. Goldman, “Extending microservices performance optimization through horizontal pod autoscaling: A comprehensive study”, in *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2025, pp. 553–562. DOI: 10.1109/IPDPSW66978.2025.00089. [Online]. Available: https://ieeexplore.ieee.org/document/11106131.

[90] A. Simha, “Performance optimization during hp-ux to cloud container shifts”, en, vol. 9, 2022.

[91] B. Guntupalli and S. V. Ch, "Designing microservices that handle high-volume data loads", en, *International Journal of AI, BigData, Computational and Management Studies*, vol. 4, no. 4, pp. 76–87, Dec. 2023, ISSN: 3050-9416. DOI: 10.63282/3050-9416.IJAIBDCMS-V4I4P109.

[92] H. Dinh-Tuan, M. Mora-Martinez, F. Beierle, and S. R. Garzon, "Development frameworks for microservice-based applications: Evaluation and comparison", in *Proceedings of the 2020 European Symposium on Software Engineering*, ser. ESSE '20, New York, NY, USA: Association for Computing Machinery, Dec. 2020, pp. 12–20, ISBN: 978-1-4503-7762-1. DOI: 10.1145/3393822.3432339. [Online]. Available: https://dl.acm.org/doi/10.1145/3393822.3432339.

[93] Z. Jia and E. Witchel, "Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices", ser. ASPLOS '21, New York, NY, USA: Association for Computing Machinery, Apr. 2021, pp. 152–166, ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446701. [Online]. Available: https://dl.acm.org/doi/10.1145/3445814.3446701.

[94] A. Iurchenko, "Optimization of microservices architecture performance in high-load systems", en, *The American Journal of Engineering and Technology*, vol. 7, no. 05, pp. 123–132, May 2025, ISSN: 2689-0984. DOI: 10.37547/tajet/Volume07Issue05-10.

[95] S. Chaudhary, D. K. Somwanshi, V. S. Rajawat, M. Sharma, and P. Verma, "Optimizing cold start latency in serverless computing: A comprehensive review of techniques and emerging solutions", in *Proceedings of the 6th International Conference on Information Management & Machine Intelligence*, ser. ICIMMI '24, New York, NY, USA: Association for Computing Machinery, 2025, pp. 1–8, ISBN: 979-8-4007-1122-0. DOI: 10.1145/3745812.3745825. [Online]. Available: https://dl.acm.org/doi/10.1145/3745812.3745825.

[96] X. Gu and Q. Wang, "Pathfence: Reducing cross-path dependencies in microservices", in *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '25, New York, NY, USA: Association for Computing Machinery, 2025, pp. 1–13, ISBN: 979-8-4007-1869-4. DOI: 10.1145/3731545.3731583. [Online]. Available: https://dl.acm.org/doi/10.1145/3731545.3731583.

[97] L. Wen et al., "Statuscale: Status-aware and elastic scaling strategy for microservice applications", *ACM Trans. Auton. Adapt. Syst.*, vol. 20, no. 1, 4:1–4:25, Mar. 2025, ISSN: 1556-4665. DOI: 10.1145/3686253.

[98] M. ZargarAzad and M. Ashtiani, "An auto-scaling approach for microservices in cloud computing environments", en, *Journal of Grid Computing*, vol. 21, no. 4, p. 73, Nov. 2023, ISSN: 1572-9184. DOI: 10.1007/s10723-023-09713-7.

[99] S. N. Srirama, M. Adhikari, and S. Paul, "Application deployment using containers with auto-scaling for microservices in cloud environment", *Journal of Network and Computer Applications*, vol. 160, p. 102 629, 2020, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2020.102629.

[100] M. Abdullah, W. Iqbal, J. L. Berral, J. Polo, and D. Carrera, "Burst-aware predictive autoscaling for containerized microservices", *IEEE Transactions on Services Computing*, vol. 15, no. 3, pp. 1448–1460, May 2022, ISSN: 1939-1374. DOI: 10.1109/TSC.2020.2995937.

[101] N. Shafi, M. Abdullah, W. Iqbal, A. Erradi, and F. Bukhari, "Cdascaler: A cost-effective dynamic autoscaling approach for containerized microservices", en, *Cluster Computing*, vol. 27, no. 4, pp. 5195–5215, 2024, ISSN: 1573-7543. DOI: 10.1007/s10586-023-04228-y.

[102] N. Shafi, M. Abdullah, W. Iqbal, and F. Bukhari, "Cema: Cost effective multi-layered autoscaling for microservice based applications", *Journal of Network and Computer Applications*, vol. 242, p. 104 266, Oct. 2025, ISSN: 1084-8045. DOI: 10.1016/j.jnca.2025.104266.

[103] V. Sachidananda and A. Sivaraman, "Erlang: Application-aware autoscaling for cloud microservices", in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24, New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 888–923, ISBN: 979-8-4007-0437-6. DOI: 10.1145/3627703.3650084. [Online]. Available: https://dl.acm.org/doi/10.1145/3627703.3650084.

[104] J. P. Karol Santos Nunes, S. Nejati, M. Sabetzadeh, and E. Y. Nakagawa, "Self-adaptive, requirements-driven autoscaling of microservices", in *Proceedings of the 19th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '24, New York, NY, USA: Association for Computing Machinery, 2024, pp. 168–174, ISBN: 979-8-4007-0585-4. DOI: 10.1145/3643915.3644094. [Online]. Available: https://dl.acm.org/doi/10.1145/3643915.3644094.

[105] A. Marchese and O. Tomarchio, "Slo and cost-driven container autoscaling on kubernetes clusters:" en, in *Proceedings of the 15th International Conference on Cloud Computing and Services Science*, Porto, Portugal: SCITEPRESS - Science and Technology Publications, 2025, pp. 72–79, ISBN: 978-989-758-747-4. DOI: 10.5220/0013482100003950. [Online]. Available: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0013482100003950.

Appendix A The Use of AI

Artificial intelligence tools were used in a limited capacity to support the implementation phase of the empirical experiments in Chapter 5. Specifically, the Github Copilot tool for Visual Studio Code was used as a programming assistance tool during the development of the benchmark applications and supporting scripts. The selected LLM model was Claude Sonnet 4.5 and it was used in both chat and agent modes.

The AI tool was used solely to assist with software development tasks, such as replicating benchmark application implementations across frameworks, debugging code and identifying suitable programming libraries for specific frameworks. Additionally, the tool was used to automate repetitive tasks, such as generating PowerShell scripts for building and packaging multiple container images at the same time.

The use of AI tools was strictly limited only to programming assistance. The AI tools were not used in designing the research questions, research problem, experimental methodology, analysing data, interpreting results or writing the contents of the thesis. All research decisions, design of experiments, methodological choices, analysis and conclusions were made solely by the author.

Appendix B Empirical Experiments Source Code

The implementation of the benchmark applications, container configurations, deployment scripts, automation scripts and infrastructure code used in the empirical experiments are available in a public Github repository. The repository includes instructions for building and deploying the applications, and reproducing the experiment results.

Repository: *https://github.com/KaarleJ/microservice-experiments*