Sveučilište u Rijeci
Fakultet informatike
i digitalnih tehnologija

University Graduate Program in Informatics

Antonio Janach

Comparison of Orchestration Systems for Microservices Applications

Master's Thesis

Mentor: adj. assist. prof. dr. Rok Piltaver
Co-mentor: prof. dr. Sanda Martinčić-Ipšić

Rijeka, September 2024

Sveučilište u Rijeci
Fakultet informatike i digitalnih tehnologija

Rijeka, 14.5.2024.

Zadatak za diplomski rad

Pristupnik: Antonio Janach

Naziv diplomskog rada: Usporedba sustava za orkestraciju mikrouslužnih aplikacija

Naziv diplomskog rada na eng. jeziku: Comparison of Orchestration Systems for Microservices Applications

Sadržaj zadatka:

Cilj ovog diplomskog rada je provesti detaljnu usporedbu različitih alata za orkestraciju, uključujući Azure Kubernetes Service, K3S i OpenShift. Usporedba će se temeljiti na studiji slučaja orkestriranja mikroservisne aplikacije srednje složenosti koja se sastoji od 5 do 20 mikroservisa. Usporedit će se barem dva alata za orkestraciju, jedan koji podržava lokalno okruženje (on-premises) i drugi koji podržava rješenje "u oblaku" (cloud). Kvantitativna usporedba će se fokusirati na performanse i isplativost. Mjerenja će biti provedena na infrastrukturi s identičnim resursima, uključujući broj procesora (vCPU), količinu radne memorije (RAM), kapacitet i tip diskova, te broj klastera povezanih s kontrolnom ravninom i radnim čvorovima.

Kvalitativna usporedba identificirat će i analizirati razlike u implementaciji, konfiguraciji, jednostavnosti implementacije i integracijama, upravljanju, sigurnosti i drugim značajkama ovih sustava. Nadalje, istražit će se dostupni alati i tehnike za praćenje performansi i optimizaciju klastera u svakom okruženju. Također, usporedit će se dostupnost i kvaliteta službene podrške te korisnost povezanih online zajednica.

Dva glavna rezultata bit će formalni proces za odabir optimalne orkestracijske platforme i sveobuhvatna usporedba proučenih alata za orkestraciju specifične mikroservisne aplikacije. Optimalni alat za orkestraciju bit će odabran na temelju izmjerenih podataka i analize prikupljenih informacija.

Sadržaj zadatka na eng. jeziku:

The aim of this thesis is to conduct a detailed comparison of different orchestration tools such as Azure Kubernetes Service, K3S or OpenShift. The comparison will be based on a case-study of orchestrating a microservices application of medium complexity consisting of 5 to 20 microservices. At least two orchestration tools will be compared, one supporting on-premises and the other supporting a cloud environment. Quantitative comparison will focus on performance and cost-effectiveness. Measurements will be conducted on infrastructure with identical resources, including the number of processors (vCPU), amount of memory (RAM), capacity and type of system disk, and the number of clusters related to the control plane and worker nodes.

Qualitative comparison will identify and analyse differences in implementation, configuration, ease of deployment and integrations, management, security, and other features of these systems. Furthermore, available tools and techniques for monitoring performance and optimizing clusters in each environment will be explored. Additionally, the availability and quality of official support and usefulness of related online communities will be compared.

The two main results will be a formal process for selecting the optimal orchestration platform and a comprehensive comparison of the studied orchestration tools for the specific microservices application. The optimal orchestration tool will be selected based on the measured data and analysis of the collected information.

Mentor:

nasl. doc. dr. sc. Rok Piltaver

[illegible]

Voditeljica za diplomske radove:

Prof. dr. sc. Ana Meštrović

[illegible]

Komentor:

Prof. dr. sc. Sanda Martinčić-Ipšić

[illegible]

Zadatak preuzet: 14.5.2024.

Antonio Janach

Janach

Preface

The completion of this thesis would not have been possible without the support and encouragement of several important people in my life.

First and foremost, I dedicate this work to my parents, whose unwavering support and belief in me have been the foundation of my achievements. I am deeply grateful to my grandparents for their wisdom and life lessons that have guided me throughout my journey. To my partner, Mia, your love, patience, and understanding have been my strength, motivating me to push forward even during the most challenging times.

I also wish to thank my faculty professors, colleagues, and mentor, whose guidance and knowledge have significantly shaped my academic growth. Their dedication to teaching and their passion for the subject matter have been a constant source of inspiration. A special thanks to my mentors for their insightful guidance and unwavering support throughout this process.

I am also deeply appreciative of Ivica, my leader at work, who provided me with the opportunity to work with these cutting-edge technologies. Your trust and guidance have been instrumental in my professional development. Additionally, I am grateful to SICK Mobilisis for providing the resources necessary to work on this thesis, enabling me to explore and deepen my understanding of these advanced technologies.

This thesis is a testament to the collective support and encouragement of those who have stood by me. Thank you all for being a part of this journey.

Abstract

This thesis provides a comprehensive comparison of Kubernetes orchestration tools, specifically focusing on Azure Kubernetes Service (AKS) and K3S, to determine their suitability for orchestrating a medium complexity microservices application, exemplified by the "Online Boutique" application, which consists of 15 containers. The analysis compares one tool supporting on-premises environments (K3S) with another designed for cloud environments (AKS), emphasizing performance, cost-effectiveness, management complexity, and scalability.

The quantitative analysis was conducted on infrastructure with identical resources, including CPU, memory, and storage, to ensure a fair comparison. AKS demonstrated significant cost advantages over a five-year period, largely due to its integration with the Azure ecosystem, which optimizes resource allocation and reduces operational overhead. However, K3S consistently outperformed AKS in key performance metrics, including CPU speed, memory transfer rate, and request-handling capabilities. These performance differences are partly due to the additional load created by the hypervisor and the extra cloud-specific services running within the AKS cluster.

The qualitative analysis identified differences in implementation, configuration, ease of deployment, integration, and management. AKS excels in cloud environments due to its automated management and seamless integration with Azure, making it suitable for organizations looking to minimize operational overhead. In contrast, K3S offers greater flexibility and customization, particularly for on-premises deployments or scenarios requiring specific configurations. Additionally, K3S is suitable for organizations with existing on-premises infrastructure.

Keywords: Kubernetes, Azure Kubernetes Service, AKS, K3S, microservices, cloud computing, on-premises, container orchestration, performance analysis, cost-effectiveness.

Usporedba sustava za orkestraciju mikrouslužnih aplikacija

Sažetak

Ovaj diplomski rad pruža sveobuhvatnu usporedbu Kubernetes alata za orkestraciju, s posebnim naglaskom na Azure Kubernetes Service (AKS) i K3S, u svrhu utvrđivanja njihove prikladnosti za orkestriranje aplikacija srednje složenosti, pri čemu je korištena "Online Boutique" aplikacija koja se sastoji od 15 kontejnera. Analiza uspoređuje jedan alat koji podržava lokalna okruženja (K3S) s drugim dizajniranim za okruženja u oblaku (AKS), s naglaskom na performanse, isplativost, složenost upravljanja i skalabilnost.

Kvantiativna analiza provedena je na infrastrukturi s identičnim resursima, uključujući procesor, memoriju i pohranu, kako bi se osigurala pravedna usporedba. AKS je pokazao značajne prednosti u troškovima tijekom petogodišnjeg razdoblja, uglavnom zbog integracije s Azure ekosustavom, što optimizira alokaciju resursa i smanjuje operativno opterećenje. Međutim, K3S je u svim ključnim performansama nadmašio AKS, uključujući brzinu procesora, brzinu prijenosa memorije i sposobnost obrade zahtjeva. Ove razlike u performansama djelomično su uzrokovane dodatnim opterećenjem koje stvaraju hipervizor i dodatni servisi specifični za okruženje u oblaku unutar AKS klastera.

Kvalitativna analiza identificirala je razlike u implementaciji, konfiguraciji, lakoći postavljanja, integraciji i upravljanju. AKS se ističe u okruženjima u oblaku zahvaljujući automatiziranom upravljanju i besprijekornoj integraciji s Azureom, čime je prikladan za organizacije koje žele minimizirati operativno opterećenje. Nasuprot tome, K3S nudi veću fleksibilnost i prilagodljivost, osobito za lokalne implementacije ili scenarije koji zahtijevaju specifične konfiguracije. Dodatno, K3S je prikladan za organizacije koje već imaju postojeću on-prem okruženje.

Ključne riječi: Kubernetes, Azure Kubernetes Service, AKS, K3S, mikroservisi, računarstvo u oblaku, lokalno okruženje, orkestracija kontejnera, analiza performansi, isplativost.

Table of Contents

1. Introduction 1

1.1. Background and Motivation 1

1.2. Objectives of the Thesis 3

1.3. Limitation 4

1.4. Structure of the Thesis 5

2. Theoretical Background 7

2.1. Microservices Architecture 7

2.1.1. Definition and Principles 7

2.1.2. Benefits and Challenges of Microservices Architecture 10

2.2. Kubernetes - Orchestration Tool 11

2.2.1. Orchestration Tools in General 11

2.2.2. Overview of the Kubernetes Orchestration Tool 14

2.3. Overview of Selected Orchestration Tools 18

2.3.1. Azure Kubernetes Service (AKS) 18

2.3.2. K3S 19

3. Case Study Design 22

3.1. Description of the Microservices Application 22

3.1.1. Application Architecture 22

3.1.2. Functional and Non-functional Requirements 24

3.2. Experimental Setup 26

3.2.1. Infrastructure Specifications 26

3.2.2. Platform Infrastructure Elements 28

3.2.3. Deployment Scenarios 29

3.2.4. Testing Methodology 29

4. Implementation and Deployment 31

4.1. Deploying Application on Azure Kuberentes Service (AKS) 31

4.1.1. Configuration and Setup of Infrastructure 31

4.1.2. Deployment Process of Application Services 35

4.1.3. Deployment Process of System Services 36

4.1.4. Challenges and Solutions 37

4.2. Deploying on K3S 40

4.2.1. Configuration and Setup of Infrastructure 40

4.2.2. Deployment Process of Application Services 44

4.2.3. Deployment Process of System Services 45

4.2.4. Challenges and Solutions 45

5. Performance and Cost Analysis 47

5.1. Resource Utilization 47

5.1.1. Comparison of Pods and Containers 47

5.1.2. CPU and Memory Usage 48

5.1.3. Storage and Network Performance 49

5.2. Benchmark Results 52

5.2.1. Apache AB Results 53

5.2.2. K6 Results 57

5.2.3. System Benchmarking Results 65

5.3. Cost Analysis 70

5.3.1. Cost Calculation Methodology 71

5.3.2. Comparative Cost Analysis 73

5.3.3. Insights on On-Premises vs Cloud-Based Solutions 80

6. Qualitative Analysis 82

6.1. Implementation and Configuration Differences 82

6.2. Ease of Deployment and Integrations 83

6.3. Management, Monitoring and Security Features 85

6.4. Documentation and Community Support... 88

7. Discussion.. 90

7.1. Summary of Findings 90

7.2. Comparative strengths and Weaknesses of Each Tool....................................... 92

7.3. Recommendations for Selecting an Orchestration Platform 95

7.4. Gartner Magic Quadrant for Container Management.. 96

8. Conclusion.. 99

8.1. Comprehensive Comparison and Final Recommendations............................... 100

8.2. Implications for Future Work 101

References 103

List of Tables.. 107

List of Figures... 108

List of Codes... 110

Appendix 111

1. Introduction

The advent of microservices architecture has transformed the way modern applications are designed and deployed. Microservices offer exceptional flexibility and efficiency by breaking down applications into smaller, independent services that can be developed, deployed, and scaled individually. However, managing these microservices, especially at scale, introduces significant complexity. Orchestration tools like Azure Kubernetes Service (AKS) and K3S have emerged as vital technologies to address these challenges, providing robust solutions for deploying, scaling, and operating containerized applications [1].

This thesis aims to conduct a detailed comparison of various orchestration tools, specifically focusing on AKS and K3S. The comparison is grounded in a case study involving the orchestration of a microservices application of medium complexity. Strengths and weaknesses of each tool are identified in both on-premises and cloud infrastructure deployment environments by examining quantitative and qualitative aspects. The goal is to provide methodology for a comprehensive analysis that can guide organizations in selecting the most suitable orchestration platform for their specific needs.

1.1. Background and Motivation

The rapid advancement of technology has led to the widespread adoption of microservices architectures. These architectures offer significant benefits, such as scalability, maintainability, and agility. Unlike monolithic architectures, where all components are interdependent, microservices break down applications into smaller, independent services [2]. These services can be developed, deployed, and scaled individually, allowing for greater flexibility and faster iteration. This modular approach enables organizations to respond more quickly to changing business requirements and market demands.

Containers have become essential for implementing microservices architectures. They provide a lightweight, portable, and consistent environment for running applications, regardless of the underlying infrastructure [3]. By encapsulating an application and its dependencies into a single package, containers ensure that microservices can be deployed and run reliably across different environments. This isolation enhances security and simplifies the process of scaling and managing applications.

However, managing these microservices efficiently poses substantial challenges. The complexity of orchestrating multiple services, ensuring their seamless interaction, and maintaining their health and performance can be daunting without the right tools and strategies.

Orchestration tools have emerged as essential components for managing the deployment, scaling, and operation of containerized applications in distributed environments. These tools provide a framework for automating the deployment, scaling, and operation of application containers across clusters of hosts [3]. They offer built-in mechanisms for service discovery, load balancing, and automated rollouts and rollbacks.

Microservices applications can be deployed in one of three environments: cloud, on-premises, or hybrid. Each environment presents unique challenges and opportunities for orchestration. Organizations are increasingly adopting cloud-native technologies to leverage the flexibility and scalability of cloud environments [1]. Cloud-native applications are designed to fully exploit the advantages of the cloud computing model, such as elastic scalability, resilience, and the ability to run applications in a highly distributed manner.

Simultaneously, many enterprises maintain on-premises infrastructure to meet specific regulatory, performance, or cost requirements. Regulatory requirements may mandate that certain data must remain on-premises, while performance considerations may require low-latency access to data and services. Cost constraints can also play a role, as maintaining on-premises infrastructure may be more economical for certain workloads.

To address these diverse requirements, a hybrid approach, combining on-premises and cloud environments, is often necessary. Hybrid deployments enable organizations to leverage the benefits of both environments, providing the flexibility to optimize resources, meet compliance requirements, and achieve cost efficiencies [4]. By seamlessly integrating on-premises systems with cloud services, hybrid strategies offer a robust solution for modern application deployment and management.

Selecting an optimal orchestration tool that can effectively integrate with both on-premises and cloud environments is crucial. Such a tool must provide consistent management and operational capabilities across these environments, ensuring that microservices can be deployed and managed with the same ease and efficiency, regardless of where they run. This requires robust support for hybrid and multi-cloud deployments, including features such as unified monitoring and management, consistent security policies, and seamless workload portability.

Platforms like Azure Kubernetes Service (AKS) and K3S stand out among the appropriate orchestration tools due to their widespread adoption and comprehensive feature sets. AKS, a managed Kubernetes service by Microsoft Azure, offers integrated CI/CD, monitoring, and security features that simplify container management [5]. K3S, a lightweight Kubernetes distribution by Rancher Labs, is optimized for resource-constrained environments, making it ideal for edge computing and IoT applications [6].

This thesis is motivated by the need to provide a detailed and practical comparison of leading orchestration tools, facilitating informed decision-making for organizations looking to optimize their microservices management. We first identify the strengths and weaknesses of each tool in different deployment scenarios by analyzing the performance, cost-effectiveness, and qualitative features of AKS and K3S. Based on the comparison, we propose a formal process for selecting the most suitable orchestration platform that will enable organizations to achieve their operational and strategic objectives.

1.2. Objectives of the Thesis

This thesis aims to compare Azure Kubernetes Service (AKS) and K3S as orchestration tools through a case study of a medium complexity microservices application. The application itself comprises 15 microservices, each running individual pods within the Kubernetes Cluster. Additionally, 7 system services are deployed as a part of kube-prometheus-stack, providing essential monitoring and management capabilities. The findings from this thesis will promote best practices in managing cloud-native applications and provide valuable resources for practitioners and researchers alike. The specific objectives are as follows:

Quantitative Comparison:

To evaluate the performance and cost-effectiveness of at least two orchestration tools, focusing on Azure Kubernetes Service (AKS) and K3S, one supporting cloud environments deployment and the other supporting on-prem. To conduct measurements on infrastructure with identical resources, including the number of processors (vCPU), amount of memory (RAM), capacity, and type of system disk, as well as the number of clusters related to the control plane and worker nodes. To analyze performance metrics such as response time, throughput, and resource utilization to determine how each tool handles the orchestration of microservices.

Qualitative Comparison:

To identify and analyze differences in implementation, configuration, ease of deployment, integrations, management, security, and other relevant features of the orchestration tools. To examine the setup and configuration processes, the user-friendliness of each tool, the integration capabilities with existing systems and services, and the security features they offer. Additionally, to explore available tools and techniques for monitoring performance and optimizing clusters to understand how each platform supports ongoing operations and maintenance.

Support and Community:

To compare the availability and quality of official support and the usefulness of related online communities for each orchestration tool. To evaluate the documentation, official support channels, and the strength and activity level of user communities. The goal is to understand how accessible and effective the support mechanisms are for each tool, and how community resources can aid in troubleshooting and optimizing the use of the orchestration platforms.

Selection Process:

To develop a formal process for selecting the optimal orchestration platform based on the measured data and the analysis of collected information. To design framework, i.e. a structured system for evaluating both quantitative and qualitative factors, which organizations can use to evaluate orchestration tools based on their specific needs and constraints. To provide an associated set of guidelines and recommendations to help organizations choose the most suitable orchestration tool for their microservices applications.

1.3. Limitation

While this study provides valuable insights into the performance and suitability of Azure Kubernetes Service (AKS) and K3S for orchestrating microservices, it is essential to consider the following limitations:

- **Application Specificity**: The case study application is of medium complexity and the results may not be directly applicable to applications with significantly different complexity levels. Organizations with simpler or more complex applications might need to conduct additional evaluations to determine the best orchestration tool for their specific needs.

- **Selected Orchestration Tools**: This study focuses on AKS and K3S due to their widespread adoption and relevance. While OpenShift is mentioned for context, the practical part of this thesis does not cover it. Other orchestration tools, such as Google Kubernetes Engine (GKE) or Amazon Elastic Kubernetes Service (EKS), are also available but not included in this comparison.
- **Deployment Environment**: The comparison is limited to the specified environments: on-premises and cloud. The study does not account for hybrid or multi-cloud deployments, which may have different requirements and challenges.
- **Resource Constraints**: Measurements are conducted on infrastructure with identical resources, including the number of processors (vCPU), amount of memory (RAM), capacity and type of system disk, and the number of clusters related to the control plane and worker nodes. While this ensures a fair comparison, it may not represent all possible deployment scenarios or resource configurations.
- **Temporal Relevance**: The rapidly evolving nature of technology means that the findings of this thesis may become outdated as new features and improvements are introduced to the orchestration tools. Regular updates and new releases from AKS and K3S could impact the relevance and accuracy of the results over time.

1.4. Structure of the Thesis

The following is an overview of the thesis structure:

The first chapter introduces the topic of the thesis, providing background information and the motivation behind the study. It outlines the thesis's objectives, scope, and limitations and presents the overall structure.

The second chapter provides the foundational knowledge necessary for understanding microservices architecture. It begins with an exploration of microservices, covering their definition, principles, benefits, and challenges. The chapter then delves into orchestration tools, offering an overview of their purpose, key features, and functions. Specific attention is given to Azure Kubernetes Service (AKS) and K3S as the selected orchestration tools.

The third chapter describes the case study that serves as the basis for the comparison. This includes a detailed description of the microservices application, its architecture, and both its functional and non-functional requirements. The experimental setup is explained, covering infrastructure specifications, deployment scenarios, and the testing methodology.

The fourth chapter shifts the focus to the practical implementation and deployment of the microservices application using the selected orchestration tools. This includes a step-by-step guide on configuring and setting up Azure Kubernetes Service (AKS) and K3S, detailing the deployment processes, and addressing any challenges encountered along the way.

The fifth chapter evaluates the performance and cost-effectiveness of the orchestration tools. Performance testing results are presented, followed by a detailed analysis of the performance metrics. The cost analysis includes the methodology for cost calculation and a comparative analysis of costs. Resource utilization, including CPU, memory, storage, and network performance, is also discussed.

In the sixth chapter, a qualitative analysis examines differences in implementation and configuration, ease of deployment, integrations, management, and security features of the orchestration tools. Tools and techniques for monitoring and optimization are explored, and the quality of community support and documentation is assessed.

The seventh chapter synthesizes the findings from the quantitative and qualitative analyses. It summarizes the strengths and weaknesses of each orchestration tool, offers recommendations for selecting an orchestration platform, and discusses the implications of the research findings for future work.

The concluding chapter summarizes the entire thesis, highlighting key contributions and findings.

2. Theoretical Background

This chapter provides a comprehensive theoretical background on microservices architecture, delving into its definition, principles, benefits, and challenges. It then explores orchestration tools, detailing their key features, functions, and importance in managing microservices. Finally, the chapter provides an in-depth overview of two selected orchestration tools: Azure Kubernetes Service (AKS) and K3S, highlighting their unique features, benefits, and suitable use cases.

2.1. Microservices Architecture

2.1.1. Definition and Principles

Microservices architecture is a software design approach that decomposes an application into a collection of loosely coupled, independently deployable services [1]. Each service corresponds to a specific business capability and can be developed, deployed, and scaled independently. This modular approach is in sharp contrast to traditional monolithic architectures, where all functionalities are tightly interconnected in a single, large application [7].

Key Principles of Microservices Architecture [8]**:**

1. **Single Responsibility Principle:** Each microservice should focus on a single piece of business functionality. This principle promotes a clear separation of concerns, making the system easier to understand, develop, and maintain.
2. **Independence:** Services should be independently deployable and scalable. This independence allows for continuous delivery and deployment, enabling faster time-to-market.
3. **Decentralized Data Management:** Each service manages its own database or data storage, ensuring that services remain decoupled. This approach simplifies the data model and enhances data consistency within each service.
4. **Inter-Service Communication:** Microservices communicate with each other via lightweight protocols such as HTTP/REST or messaging queues. This communication must be robust and reliable to ensure the smooth operation of the system.
5. **Automation through CI/CD:**
    a. **Continuous Integration (CI):** This practice involves frequently merging code changes from multiple developers into a central repository, where automated

builds and tests are run. The primary goal of CI is to quicky detect and address bugs, improve software quality, and reduce the time it takes to validate and release new software updates.

b. **Continuous Deployment (CD):** Extends CI by automatically deploying all code changes from the repository to the production environment after passing set tests. This means changes are automatically and reliably made live, providing a faster pathway to addressing user needs and improving the application.

Significance of CI/CD in Microservices:

CI/CD enables organizations to handle the inherent complexity of managing multiple microservices by automating the integration and deployment processes. This automation supports frequent updates and consistent system behaviour across different environments [9]. For microservices, where multiple services must work in concert yet be developed and scaled independently, CI/CD provides a systematic approach to rapid and reliable software delivery. This is crucial for maintaining system integrity and agility in the dynamic operational landscapes where microservices thrive [10].

Microservices are often deployed using containerization technologies such as Docker, which encapsulates a service and its dependencies in a container [2]. This encapsulation ensures consistent operation across various environments, further supported by CI/CD pipelines that automate the testing and deployment of these containers.

Role of Containers in Microservices:

Containers are a lightweight form of virtualization that encapsulate an application and its dependencies into a self-contained unit that runs consistently across any computing environment. This technology is pivotal for microservices architectures due to its ability to isolate software from its surroundings, thus ensuring that it works uniformly despite differences in infrastructure or location.

In a microservices architecture, each service is deployed as a separate container, allowing individual services to run independently of others [3]. This independence is crucial for implementing the core principles of microservices such as agility, scalability, and resilience.

- **Consistency and Portability:** Containers provide a consistent environment for applications from development through production. This portability simplifies the deployment process across different computing environments, from a developer's local machine to the production servers.

- **Rapid Provisioning:** Containers can be created and destroyed in seconds, providing the agility needed to deploy, scale, and terminate services swiftly according to demand.
- **Resource Efficiency:** Unlike traditional virtual machines that each require a full operating system, containers share the host system's kernel. This design significantly reduces overhead, making it possible to run more services on the same hardware.

As the deployment of microservices scales, understanding and managing the lifecycle of the underlying cloud infrastructure becomes critical. By comprehensively addressing each phase of the lifecycle, organizations can ensure that their microservices architecture remains robust, efficient, and capable of meeting evolving demands. The following section details the lifecycle of cloud infrastructure within the context of microservices deployment.

Advantages of Cloud Infrastructure [3]**:**

1. **Agility:** Cloud infrastructure enables rapid deployment and scaling of services, allowing organizations to respond quickly to changing business needs.
2. **Cost-Efficiency:** The pay-as-you-go model of cloud services minimizes upfront capital expenses and aligns costs with actual usage, promoting financial efficiency.
3. **Resilience:** Built-in redundancy and geographic distribution of cloud resources enhance the reliability and availability of applications, minimizing downtime and ensuring business continuity.

Lifecycle of Cloud Infrastructure [11]**:**

The lifecycle of cloud infrastructure in a microservices context involves several critical stages, each essential for maintaining an efficient, reliable, and scalable system:

1. **Planning and Design:** This initial stage involves defining the architecture, selecting appropriate technologies (e.g., Kubernetes for orchestration, Docker for containers), and establishing best practices for service design.
2. **Development:** During development, each microservice is built to conform to the defined architecture and principles. Continuous Integration ensures that new code is automatically tested and integrated into the existing codebase, reducing integration issues.

3. **Deployment:** Services are deployed using Continuous Deployment pipelines, allowing for automated releases to production environments. This stage leverages orchestration tools to manage service distribution across nodes, ensuring high availability.
4. **Monitoring and Scaling:** Once deployed, services are continuously monitored for performance and reliability using tools like Prometheus and Grafana. These insights drive automated scaling actions, ensuring resources are allocated efficiently based on demand.
5. **Maintenance and Optimization:** Regular updates and optimizations are performed to improve performance, patch vulnerabilities, and add new features. This stage is critical for maintaining system health and adapting to changing user needs.
6. **Retirement:** As services become obsolete or are replaced by newer solutions, they are systematically decommissioned to free up resources and reduce complexity.

Integrating these stages into lifecycle management ensures that cloud infrastructure can adapt to and support the evolving demands of modern applications.

2.1.2. Benefits and Challenges of Microservices Architecture

Understanding the benefits and challenges of microservices architecture is essential for organizations considering or currently implementing microservices architecture. This understanding helps in making informed decisions and optimizing their architecture for better performance and reliability. The Table 1 below summarizes the key benefits and challenges associated with microservices architecture.

Table 1. Summary of Benefits and Challenges in Microservices Architecture. Source [2].

Category	Aspect	Description
Benefits	Scalability	Microservices allow each service to be scaled independently, optimizing resource utilization. This flexibility is particularly valuable in handling varying loads and improving application performance.
	Flexibility	Developers can use different technologies and frameworks for different services, depending on what is best suited for the task. This polyglot approach enables teams to leverage the best tools for each job.
	Resilience	Fault isolation is improved as failures in one service do not necessarily affect others. This resilience enhances the overall availability and reliability of the application.
	Faster Deployment	Teams can deploy services independently, leading to more frequent and faster updates. This agility is crucial for responding to changing business requirements and market conditions.
	Better Organization	Microservices align with modern DevOps practices and encourage a more agile and collaborative approach to software

		development. This alignment fosters a culture of continuous improvement and innovation.
Challenges	Complexity	Managing numerous services can become complex, particularly in terms of deployment, monitoring, and maintenance. This complexity requires sophisticated tooling and robust practices to handle effectively.
	Inter-Service Communication	Ensuring reliable communication between services can be challenging, especially when network issues arise. Implementing robust communication patterns and handling failures gracefully are critical to maintaining system stability.
	Data Management	Maintaining consistency across decentralized databases can be difficult. Techniques such as eventual consistency and distributed transactions are often necessary but add complexity.
	Deployment Overhead	Setting up and managing CI/CD pipelines for multiple services requires significant effort. Automation and orchestration tools are essential to streamline these processes and reduce manual intervention.
	Latency	Cross-service communication can introduce latency, impacting performance. Optimizing communication paths and minimizing data transfer can help mitigate these effects.

2.2. Kubernetes - Orchestration Tool

2.2.1. Orchestration Tools in General

This chapter explores what orchestration tools are, their purposes, and the details of several prominent orchestration tools used in managing microservices.

Orchestration tools are crucial for the management of microservices architectures, as they provide a software platform to automate the deployment, scaling, and operation of containerized applications. These tools ensure that services run smoothly, remain highly available, and dynamically scale based on demand. By automating and coordinating the processes involved, orchestration tools facilitate the efficient operation of complex applications composed of multiple, interdependent services.

Key Features and Functions of Kubernetes

Table 2 summarizes the key features and functions of Kubernetes, highlighting how they contribute to the effective orchestration and management of containerized applications:

Table 2. Summary of Key Features and Functions in Orchestration Tools. Source [3].

Category	Feature/Function	Description
Key Features	Declarative Configuration	Users define the desired state of the system using configuration files. The orchestration tool then ensures the system matches this state. This approach simplifies management and reduces errors.
	Self-Healing	Orchestration tools automatically replace failed containers and ensure that the desired number of instances is always

		running. Self-healing improves the resilience and availability of applications.
	Rolling Updates and Rollbacks	These tools facilitate seamless updates by gradually replacing old versions with new ones. They also allow rolling back to previous versions if issues are detected, minimizing downtime.
	Horizontal Scaling	Orchestration tools automatically adjust the number of running instances of a service based on demand. This capability helps maintain performance and handle varying loads.
	Persistent Storage	Managing storage resources and ensuring data persistence in case of container restarts. Persistent storage solutions enable applications to retain state and data integrity.
Functions	Deployment	Orchestration tools automate the deployment of applications, ensuring containers are launched with the correct configurations. This automation reduces manual effort and increases consistency.
	Load Balancing	Distributing incoming traffic across multiple instances of a service to ensure high availability and reliability.
	Management	Orchestration tools provide a centralized platform for managing all aspects of containerized applications, including configuration, storage, and networking. Centralized management simplifies operations and enhances control.
	Resource Management	Efficiently allocating resources such as CPU and memory to different services based on their needs.
	Monitoring	Continuous monitoring of the health and performance of services, with alerts and insights, helps maintain optimal operation. Monitoring tools enable proactive issue resolution and performance optimization.
	Security	Enforcing security policies and best practices to protect applications and data is a key function of orchestration tools. Security features include encryption, access control, and network segmentation.

In Table 3 are listed key use cases for Kubernetes: managing complex applications, supporting modern development practices, and handling large-scale data processing.

Table 3. Use Cases for Kubernetes

Enterprise Applications	Large enterprises use Kubernetes to manage complex, multi-tier applications. Its scalability and resilience are crucial for maintaining the performance and availability of business-critical applications.
Cloud-Native Applications	Kubernetes is ideal for developing and deploying cloud-native applications that leverage microservices architectures. It provides the tools needed to manage the dynamic nature of these applications.
DevOps and Continuous Integration/Continuous Deployment (CI/CD)	Kubernetes streamlines the CI/CD pipeline by automating the deployment and scaling of applications. It integrates well with DevOps tools, enabling rapid development and deployment cycles.
Hybrid and Multi-Cloud Deployments	Kubernetes supports hybrid and multi-cloud environments, allowing organizations to deploy applications across on-premises and cloud infrastructures. This flexibility helps

	avoid vendor lock-in and enhances disaster recovery capabilities.
Big Data and Machine Learning	Kubernetes is used to manage big data and machine learning workloads, providing the scalability and resource management needed for processing large datasets and training complex models.

Table 4 presents key orchestration tools for microservices:

Table 4. Overview of Key Orchestration Tools for Microservices

Orchestration Tool	**Description**	**Use Case**	**Created By**
Kubernetes	Developed in 2008 by Google, Kubernetes is an open-source container orchestration tool known for its vast library of functionalities and automated deployment features. It was handed over to the Cloud Native Computing Foundation in 2014.	Large-scale enterprise applications, cloud-native applications, multi-cloud and hybrid cloud environments.	Google
K3S	Developed by Rancher Labs, K3S is a lightweight Kubernetes distribution designed for easy installation and low resource use, making it ideal for edge, IoT, and low-capacity environments.	Edge computing, IoT deployments, small to medium enterprises, and environments with limited resources.	Rancher Labs
OpenShift	OpenShift by Red Hat is an enterprise-grade hybrid platform that expands Kubernetes functionalities. It offers both Container-as-a-Service (CaaS) and Platform-as-a-Service (PaaS) models, with a small learning curve and support for creating databases and application services.	Enterprise environments, hybrid cloud deployments, and application services development.	Red Hat
Nomad	Developed by HashiCorp, Nomad supports container and non-container workloads, integrating with other HashiCorp tools like Consul, Vault, and Terraform. It supports macOS, Windows, and Linux, and offers GPU support, scalability, and multi-cloud deployments.	Mixed workload environments, organizations using HashiCorp tools, and highly dynamic, scalable applications.	HashiCorp
Docker Swarm	Docker Swarm is Docker's native support for container orchestration, providing portable and agile applications with seamless load balancing and high service availability in a distributed environment.	Small to medium-sized deployments, development and testing environments, Docker-native applications.	Docker, Inc.
Apache Mesos	Apache Mesos is an open-source cluster management tool that performs container orchestration and allows resource sharing and allocation across distributed frameworks, suitable for large-scale clustered environments.	Large-scale data processing, high-performance computing, multi-framework environments.	Apache Software Foundation
Helios	Developed by Spotify, Helios is a tool for orchestrating Docker containers,	Managing Docker containers, fitting into	Spotify

	known for its practicality and ability to fit seamlessly with most developer workflows, supporting single-node and multi-node instances.	existing developer workflows, single-node and multi-node deployments.	

2.2.2. Overview of the Kubernetes Orchestration Tool

Kubernetes is an open-source platform designed to automate the deployment, scaling, and operation of containerized applications. Originally developed by Google in 2008 Table 4, Kubernetes has become the industry standard for container orchestration. In 2014, Google handed over Kubernetes to the Cloud Native Computing Foundation (CNCF), ensuring its continued development and widespread adoption. [1]

Kubernetes operates using a cluster-based architecture composed of distributed physical or virtual servers, known as nodes. These nodes are categorized into two main types: master and worker nodes, which together form a cluster. The components installed on a node determine its functionality and identify it as either a master or worker node.

Cluster architecture

A Kubernetes cluster consists of a control plane and one or more worker nodes. The control plane is responsible for managing the overall state of the cluster, while the worker nodes execute the tasks assigned by the control plane. The architecture of a Kubernetes cluster is designed to ensure fault-tolerance and high availability, with the control plane managing the worker nodes and the pods they host.

Control plane components:

- **"API Server:** The API server (kube-apiserver) is the core component of the Kubernetes control plane that exposes the Kubernetes API. It acts as the front end for the Kubernetes control plane, handling all RESTful requests and updates to the cluster. For example, when you deploy an application on Kubernetes, the API server processes the request and updates the cluster state accordingly.
- **etcd:** A consistent and highly available key-value store that serves as the primary data storage mechanism for all cluster configuration data and state. This component is critical for maintaining the cluster's state and ensuring that configuration data is reliably stored and accessible. In a production environment, etcd is typically configured with high availability, often running as a distributed system across multiple nodes.
- **Scheduler:** The scheduler (kube-scheduler) is responsible for assigning workloads to nodes based on resource availability, performance requirements, and other constraints.

It ensures that pods are efficiently distributed across the nodes in the cluster. For instance, if one node is under heavy load, the scheduler will allocate new pods to a less utilized node to balance the workload.

- **Controller Manager:** The controller manager (kube-controller-manager) runs various controllers that regulate the state of the cluster. These controllers include the node controller, which monitors node health, and the deployment controller, which manages application updates and scaling. The deployment controller, for example, ensures that the correct number of pod replicas are always running." [4]
- Cloud Controller Manager: Specific to cloud deployments, the cloud controller manager integrates the cluster with cloud provider APIs, managing cloud-specific resources such as load balancers, storage, and networking. [5] This component is essential for leveraging cloud-specific features, such as automatically provisioning a cloud load balancer when a service of LoadBalancer type is created.

"Worker node components:

- **Kubelet:** The kubelet is an agent running on each worker node that ensures containers are running in a pod. It monitors the state of the containers and intervenes to maintain the desired state as specified by the control plane. For example, if a container in a pod crashes, the kubelet will automatically restart it.
- **Container Runtime:** The container runtime, such as Docker or containerd, is responsible for running the containers within pods. It abstracts system-level operations, allowing Kubernetes to manage containers uniformly across different environments. The container runtime ensures that containers are isolated and have the necessary resources to run.
- **Kube-proxy:** Kube-proxy is a network proxy that runs on each node, maintaining network rules and managing communication between services in the cluster. It ensures that network traffic is properly routed to and from pods. For example, kube-proxy helps in load balancing network traffic to different pod replicas providing the same service." [4].

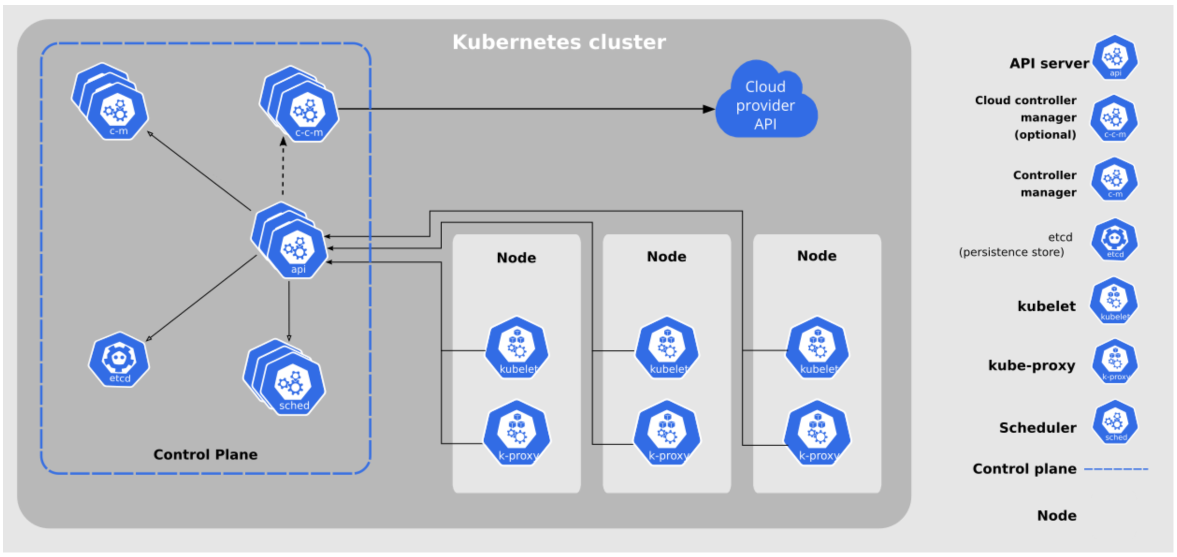


Figure 1. The Components of a Kubernetes Cluster. Source: [4].

“Kubernetes resources play a vital role in defining and managing the state and behaviour of applications within a cluster. These resources presented in Figure 1. The Components of a Kubernetes Cluster. Source: .ensure that applications are efficiently deployed, scaled, and maintained across the cluster.

- **Pods:** A pod is the smallest deployable unit in Kubernetes, encapsulating one or more containers that share the same network namespace and storage. Pods are fundamental to Kubernetes applications, allowing for easy scaling and management. They are scheduled on worker nodes to ensure efficient resource utilization. For instance, a web application might consist of multiple pods, each hosting a web server container.
- **Controllers:** Controllers in Kubernetes maintain the desired state of the cluster by managing pods and other resources. They automate essential tasks such as deployment, scaling, and self-healing. Examples include:
    - **Deployments:** These manage the rollout of new application versions, ensuring that a specified number of replicas are always running. Deployments facilitate updates and rollbacks, enhancing application management.
    - **StatefulSets:** Designed for stateful applications, StatefulSets ensure that each pod has a unique identity and stable storage, making them ideal for databases and other applications requiring consistent data access.
    - **DaemonSets:** These ensure that a copy of a pod runs on all or specific nodes in the cluster, typically used for services like logging and monitoring which need to be present on every node.

- **Services:** Services define a logical set of pods and a policy for accessing them, providing a stable network endpoint for communication between components. Services also facilitate load balancing, ensuring that traffic is evenly distributed across pods. For example, a service might expose a set of backend pods to external traffic, distributing the load evenly across all pods.
- **Ingress:** Ingress resources manage external access to services within a cluster, offering advanced load balancing, SSL termination, and name-based virtual hosting. Ingress enhances security and provides a unified entry point for accessing applications. A common use case for ingress is to route HTTP and HTTPS traffic to different services based on the request URL." [4].

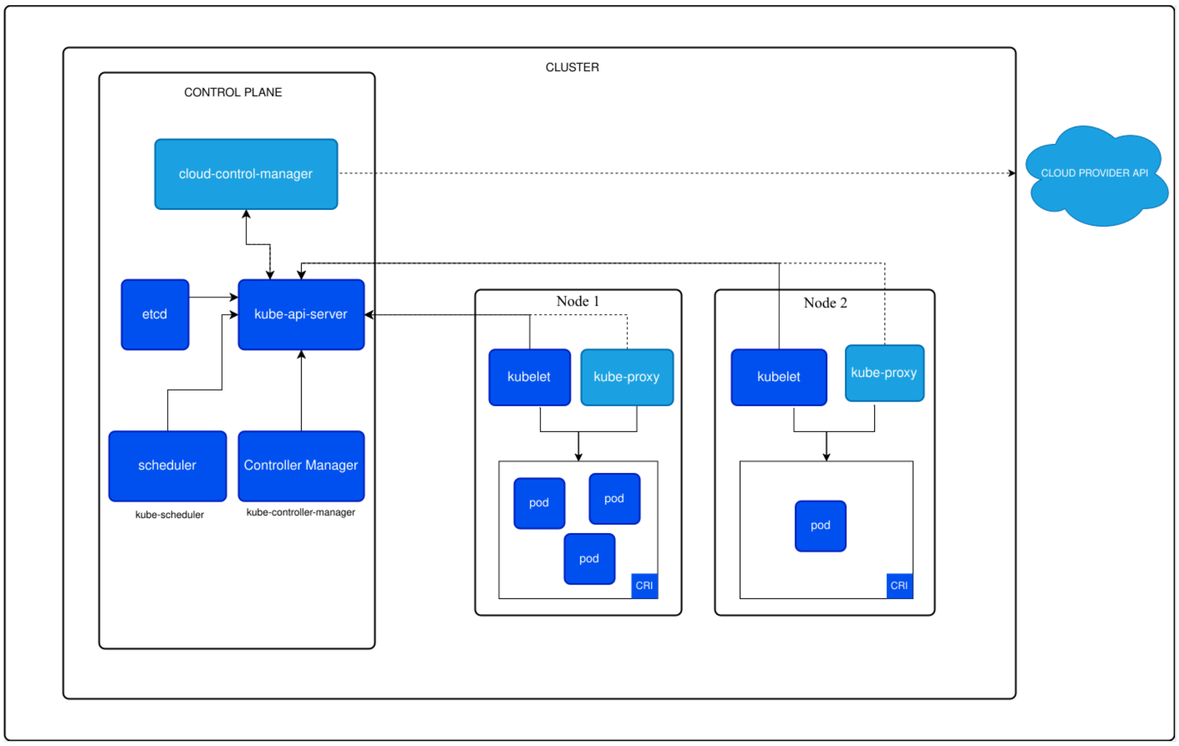


Figure 2. Kubernetes Cluster Architecture. Source: [4].

Deploying Kubernetes involves setting up the master and worker nodes, configuring networking, and ensuring that the components can communicate securely and efficiently. Figure 2 illustrates the architecture of a Kubernetes cluster, showing the relationship between the master and worker nodes, and how they handle workloads, networking, and communication between components. In

Table 5 are listed the primary methods for deploying Kubernetes:

Table 5. Deployment Methods for Kubernetes

Deployment Method	**Description**	**Tools/Services**
On-Premises Deployment	Kubernetes can be deployed on physical or virtual machines in an on-premises data center. This method involves setting up and managing clusters locally.	Kubeadm, K3S, OpenShift
Cloud-Based Deployment	Many cloud providers offer managed Kubernetes services that simplify deployment and management. These services handle much of the infrastructure setup and maintenance.	Google Kubernetes Engine (GKE), Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS)
Hybrid Deployment	Organizations can deploy Kubernetes in a hybrid environment, combining on-premises and cloud resources. This approach offers flexibility, scalability, and better disaster recovery capabilities.	OpenShift, K3S
Local Development	For development and testing purposes, Kubernetes can be deployed locally using tools that create and manage Kubernetes clusters on local machines.	Minikube, Kind (Kubernetes IN Docker), K3D

2.3. Overview of Selected Orchestration Tools

In the rapidly evolving field of container orchestration, two tools stand out: Azure Kubernetes Service (AKS) [12] and K3S [6]. They both simplify the management of containerized applications, but each is tailored to different needs.

Azure Kubernetes Service (AKS) is a fully managed service from Microsoft, which is built on the open-source Kubernetes system. It provides a robust and scalable solution for enterprise needs.

K3S is a lightweight Kubernetes distribution developed by Rancher Labs That is optimized for resource-constrained environments and IoT applications. It offers a streamlined and efficient Kubernetes experience.

This section provides an overview of AKS and K3S and sets the stage for a detailed examination of their features, benefits, and use cases.

2.3.1. Azure Kubernetes Service (AKS)

Azure Kubernetes Service (AKS) is a managed container orchestration service that simplifies the deployment, scaling, and operation of containerized applications [5]. By automating essential infrastructure tasks such as updates and scaling, AKS allows development teams to concentrate on application development.

Key Features and Advantages of AKS [5]:

- **Managed Infrastructure:** AKS automates management tasks, such as updates and scaling, reducing the operational overhead on DevOps engineers and improving efficiency.
- **Integrated Security:** AKS offers enhanced security by integrating with Azure's security services, providing robust identity management and access controls to protect sensitive data.
- **Cost-Effective Scalability:** AKS supports a flexible pricing model that allows businesses to optimize costs based on resource usage. Its scalability features enable applications to efficiently manage varying loads, making it suitable for dynamic applications.
- **High Availability and Reliability:** AKS ensures high availability with features like automated node health monitoring and self-healing capabilities, maintaining consistent performance and uptime.
- **Hybrid and Multi-Cloud Flexibility:** AKS supports deployments across hybrid and multi-cloud environments, offering businesses flexibility and preventing vendor lock-in.

Setting up AKS involves the following steps [12]:

1. **Create an AKS Cluster:** Use the Azure portal, CLI, or Azure Resource Manager templates to create a new Kubernetes cluster.
2. **Configure the Cluster:** Set up node count, networking options, and other configurations.
3. **Deploy Applications:** Use kubectl to deploy containerized applications to the cluster.
4. **Scale and Manage:** Utilize AKS features for scaling and managing applications, such as autoscaling and monitoring tools.

2.3.2. K3S

K3S is a lightweight Kubernetes distribution optimized for resource-constrained environments and IoT applications [6]. It simplifies the Kubernetes setup to offer a streamlined experience while ensuring compatibility with the broader Kubernetes ecosystem. DevOps engineers manage their entire cluster, including control plane nodes, and are responsible for updates, patches, and maintenance. This level of control allows flexibility but can introduce complexity, especially in larger deployments. Managing multiple clusters or complex infrastructure might require advanced skills and additional tools.

K3S provides a production-ready Kubernetes environment with minimal resource usage, making it ideal for edge computing and scenarios where resources are limited. It maintains essential Kubernetes functionalities but without the typical overhead of traditional distributions.

Some of the key use-cases where K3S performs well [6]:

- **Edge Computing:** K3S is ideal for managing applications on edge devices, where resources are limited, and simplicity is crucial for efficient operations.
- **IoT Deployments:** K3S is well-suited for efficiently managing fleets of IoT devices, offering streamlined operations with minimal resource consumption. Its ability to run on ARM architecture means it can be deployed directly onto a wide range of IoT devices.
- **Development and Testing:** K3S provides a lightweight environment that is perfect for developers to create and test Kubernetes applications.
- **Small-Scale Deployments:** K3S offers simplicity and efficiency, making it particularly beneficial for organizations with limited resources.
- **Hybrid Deployments:** K3S is also capable of being used in hybrid environments, effectively combining resources from edge devices and cloud setups for integrated operations.

K3S integrates with a wide range of DevOps tools to enhance its functionality and streamline development workflows. It supports Kubernetes package management, allowing developers to easily manage and deploy applications using pre-configured charts. K3S is also compatible with various continuous integration and continuous deployment (CI/CD) platforms, enabling automated pipelines that build, test, and deploy applications directly to K3S clusters. For monitoring and observability, K3S works with tools that provide insights into application performance and health. Additionally, it is well-suited for integration with configuration management systems, which automate the provisioning and management of K3S clusters. This flexibility makes K3S a versatile platform that aligns well with modern DevOps practices, supporting a comprehensive toolchain for efficient software lifecycle management.

Reasons to choose K3S include its simplicity, as it offers a straightforward installation process and minimal dependencies that make it easy to set up and manage. Additionally, it features a reduced binary size and low resource requirements. K3S also simplifies storage management by supporting local storage without the need for a dedicated backend. This makes it particularly efficient and allows for rapid deployment and scaling across various environments.

Initiating a K3S deployment involves the following steps:

1. **Install K3S:** Execute the installation script on the target machine to automatically set up and configure K3S. During installation, define the roles of each node, setting up control plane and agent nodes as needed, with customizable options available via environment variables for specific requirements like network plugins or Kubernetes versions.
2. **Configure Settings:** After installation, adjust additional settings such as network policies and storage options to tailor the cluster to specific operational needs.
3. **Deploy Applications**: Use kubectl to deploy containerized applications using YAML files that define the necessary configurations for images, replicas, and network settings.
4. **Manage and Scale:** Employ K3S's tools to monitor, manage, and dynamically scale applications according to demand, ensuring efficient performance and resource utilization.

3. Case Study Design

This chapter outlines the design of the case study for evaluating Azure Kubernetes Service (AKS) and K3S using the "microservices-demo" application developed by Google, also known as "Online Boutique" [13]. It includes a detailed description of the microservices application used for the evaluation, the experimental setup, and the specific configurations for both AKS and K3S deployments. Additionally, this chapter will explain the functional and non-functional requirements of the application, the infrastructure specifications, the deployment environment, and the monitoring setup.

3.1. Description of the Microservices Application

3.1.1. Application Architecture

"The "Online Boutique" is a cloud-native microservices demo application showcasing a modern application architecture. It consists of an e-commerce website where users can browse items, add them to the cart, and purchase them. The application is designed to demonstrate best practices for running microservices on Kubernetes.

The "Online Boutique" application is built using a collection of microservices, each designed to perform specific functions while working together to create a seamless e-commerce experience:

1. **Frontend**: This is the web-based user interface where customers can browse items, add them to their cart, and proceed to checkout. There is no login or signup; a session ID is generated automatically for all users.
2. **Product Catalog Service**: This service manages the store's product inventory. It provides a list of products from a JSON file, allows users to search for products, and retrieve details about individual items.
3. **Cart Service**: Responsible for managing the user's shopping cart. It stores items in the cart using Redis and retrieves them as needed.
4. **Checkout Service**: This service handles the entire checkout process. It retrieves the user's cart, prepares the order, and coordinates payment, shipping, and email notifications.
5. **Payment Service**: This service processes payments. It charges the provided credit card information (mocked) with the specified amount and returns a transaction ID.

6. **Shipping Service**: Manages the shipping of purchased items. It provides shipping cost estimates based on the items in the cart and ships items to the given address (mocked).
7. **Ad Service**: Displays advertisements to users. It delivers text ads based on the provided context words.
8. **Currency Service**: Offers currency conversion rates, converting amounts from one currency to another using real-time data fetched from the European Central Bank. This service handles the highest number of queries per second.
9. **Recommendation Service**: Suggests products to users based on the contents of their cart, enhancing the shopping experience with personalized recommendations.
10. **Email Service**: Sends order confirmation emails to users (mocked), ensuring they receive notification about their purchases.
11. **Load Generator**: Simulates user traffic for testing purposes, continuously sending requests to mimic realistic user shopping behaviors on the frontend.

As shown in Figure 3, the “microservices-demo” interconnection of elements microservices illustrates the architecture, demonstrating how different microservices interact and communicate within the system.” [13]

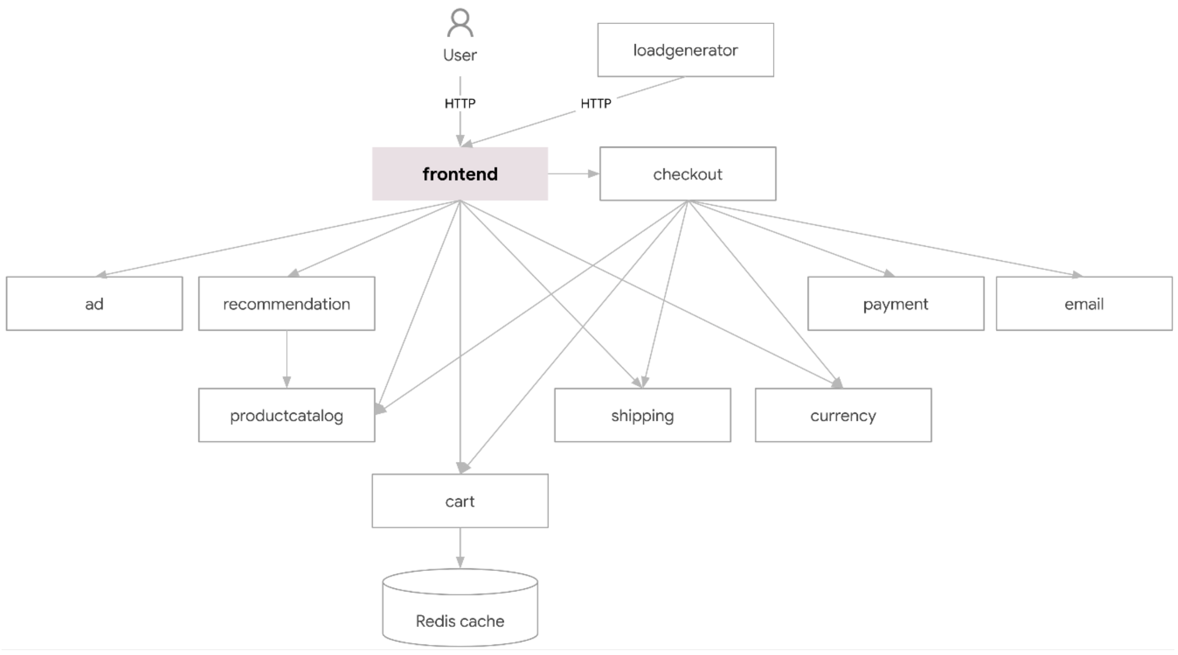


Figure 3. The "microservices-demo" interconnection of microservices. Source: [13].

Technology Stack for the 'Online Boutique' Application: The Online Boutique application leverages a diverse and robust technology stack to deliver a cloud-native, microservices-based architecture. This stack includes multiple programming languages, containerization technologies, orchestration, messaging protocols, and monitoring solutions to ensure high availability, scalability, and efficient communication between services.

- **Programming Languages**: Various, including Go, Java, Node.js, Python, C#,
- **Databases**: Redis for caching,
- **Containerization**: Docker for packaging microservices,
- **Orchestration**: Kubernetes for managing containerized applications,
- **Messaging Protocols**: HTTP for REST APIs, gRPC for internal service communication,
- **Monitoring**: Prometheus and Grafana for system monitoring and visualization.

3.1.2. Functional and Non-functional Requirements

Functional Requirements

The functional requirements of the "Online Boutique" application are inherently described in the Application Architecture section, where each service is responsible for specific application functionalities.

Non-Functional Requirements and SLOs

The system's non-functional requirements set expectations related to efficiency, reliability scalability and other considerations. These requirements are expressed as Service Level Objectives (SLOs) to provide clear, measurable goals for the system's operation:

1. **Scalability:**
    a. Objective: The platform must be scalable to handle increasing loads by accommodating up to 500 concurrent users (as demonstrated in K6 testing).
    b. SLO Target: Maintain consistent performance metrics, with no more than 5% increase in average response time when scaling from 250 to 500 users.
    c. Status: Achieved for both AKS and K3S, as both platforms demonstrated scalability and high request throughput under test conditions.

2. **Performance:**
    a. Objective: To achieve high throughput for data processing tasks in services like Currency Conversion and Shipping Cost Estimates while ensuring the average response times for user interactions remain under 100 milliseconds.
    b. SLO Target: To maintain throughput of at least 100 requests per second with response times not exceeding 100 ms for 99% of requests.
    c. Status: Partially achieved, as K3S meets these targets; however, AKS requires optimization to reduce request failures and improve response times.
3. **Reliability:**
    a. Objective: To ensure system robustness and fault tolerance to achieve 99.9% uptime.
    b. SLO Target: To limit system downtime to less than 43 minutes per month, with rapid recovery from any faults.
    c. Status: To be validated through long-term monitoring, as reliability cannot be fully assessed within the scope of this thesis. Continuous operation and real-world testing are required to confirm these targets.
4. **Portability:**
    a. Objective: The system should be deployable across various environments without significant modifications.
    b. SLO Target: To achieve deployment across different Kubernetes clusters (e.g., AKS and K3S) within 30 minutes using the same configuration files.
    c. Status: Achieved, as the application was successfully deployed on both AKS and K3S with consistent configurations.
5. **Efficiency:**
    a. Objective: Specifies the goal of keeping CPU and memory usage below 75% under normal operation to prevent overuse and manage costs effectively.
    b. SLO Target: Clearly states that the target is to maintain average CPU and memory usage below 75% for both AKS and K3S during benchmark tests.
    c. Status: Achieved for K3S, as demonstrated by Sysbench results showing efficient resource use; AKS may need further tuning.
6. **Scalability of Monitoring:**
    a. Objective: To ensure monitoring systems scale with platform growth, providing real-time insights into performance metrics.

b. SLO Target: To enable monitoring for 100% of deployed services with less than 1% performance overhead on system resources.
c. Status: Achieved, with Grafana and Prometheus providing comprehensive monitoring across both AKS and K3S.

Non-functional requirements that are beyond the scope of this thesis:

1. **Security**: Implementing robust security measures, including encrypted communications, role-based access control, and regular security assessments.
2. **Maintainability**: Ensuring the system is easy to maintain, with clear documentation, modular components, and automated deployment pipelines.
3. **Compliance**: Adhering to industry standards and regulations, including data privacy laws and cybersecurity guidelines.

3.2. Experimental Setup

This section outlines how the "Online Boutique" microservices application is deployed and tested using Azure Kubernetes Service (AKS) and K3S. It includes infrastructure specifications, deployment scenarios, and testing methodology.

3.2.1. Infrastructure Specifications

The testing infrastructure for AKS is provided in the cloud environment and runs on the latest Ubuntu OS, while the infrastructure for K3S is provided on-premises and runs on the Rocky Linux distribution. From the user perspective, the frontend service in both environments is accessible via IP addresses over the HTTP port.

Details about each infrastructure used for testing are provided below.

Cloud Environment (AKS)

It utilizes virtual machines provided by Microsoft Azure for AKS. The cluster SKU for worker nodes is Standard_A2_v2 with 2 vCPUs and 4GB RAM per virtual machine (VM) as shown in Table 6. There is one node pool with a count of two nodes. The OS disk size is 50GB. AKS is deployed using Terraform [14] to automate and manage the infrastructure setup.

Table 6. Specifications for Each Worker Node in Azure Kubernetes Service

Component	Specification
Cluster SKU	Standard_A2_v2
vCPUs per VM	2
RAM per VM	4 GB
Node Pool Count	2
OS Disk Size	50 GB

Terraform is an open-source infrastructure as code (IaC) software tool created by HashiCorp [14]. It enables users to define and provision data center infrastructure using a high-level configuration language known as HashiCorp Configuration Language (HCL), or optionally JSON. Terraform manages external resources (such as public cloud infrastructure, private cloud infrastructure, network appliances, software as a service, and platform as a service) with a provider model. This IaC approach provides a consistent workflow for provisioning and managing the lifecycle of cloud infrastructure.

On-Premises Environment (K3S)

It utilizes physical or virtual machines with the following specifications: three Control Plane Nodes each with 2 CPUs, 4GB of memory, and a 50GB SSD OS disk. Two Worker Nodes each have 2 CPUs, 4GB of memory, and a 50GB SSD OS disk as shown in Table 7 and Table 8. K3S is deployed using vSphere Center [15] to provide VMs and Ansible [16] for configuration management and automation.

Table 7. Specifications for Each Control Plane Node in On-Premises K3S Environment

Component	Specification
vCPUs per VM	2
RAM per VM	4 GB
OS Disk Size	50 GB SSD each
Master Node Count	3

Table 8. Specifications for Each Worker Node in On-Premises K3S Environment

Component	Specification
vCPUs per VM	2
RAM per VM	4 GB
OS Disk Size	50 GB SSD each
Worker Node Count	2

vSphere is a virtualization platform from VMware. vSphere, which includes ESXi hypervisor and vCenter server, allows users to manage virtual machines (VMs) and other infrastructure components from a centralized location.

Ansible is an open-source tool for provisioning, configuration management, and application-deployment enabling infrastructure as code. It uses SSH to execute operations on remote machines, making it agentless and easy to manage. By using Ansible with vSphere, users can automate the deployment, configuration, and management of their virtual infrastructure.

3.2.2. Platform Infrastructure Elements

The "Online Boutique" platform leverages several critical infrastructure elements to ensure robust and scalable operations:

Ingress Controller (NGINX)

The NGINX Ingress Controller [17] monitors Kubernetes Ingress resources and configures NGINX to handle incoming traffic to the Kubernetes cluster. It performs key functions such as load balancing, reverse proxying, SSL/TLS termination, and health checks. This ensures that traffic is efficiently managed and routed within the platform.

Prometheus

Prometheus [18] is an open-source monitoring and alerting toolkit designed for reliability and scalability. It is used to collect and store metrics from various Kubernetes objects, such as nodes, pods, containers, and services. Prometheus follows a pull-based model, periodically scraping metrics from targets and storing them as time series data.

Grafana

Grafana [19] is an open-source interactive data-visualization platform that allows users to create and share dashboards. It integrates with Prometheus to visualize metrics and provide insights into system performance. Grafana dashboards are used to monitor the health and behavior of “Online Boutique's” services and infrastructure.

3.2.3. Deployment Scenarios

“For this case study, we will use only a production-ready environment for deploying the “Online Boutique” application. However, in real-world scenarios, it is good practice to have three separate environment setups: development, staging, and production.

Development environment used for initial development and testing. It allows developers to build and test new features in an isolated setting. Typically, it includes tools and configurations that enable rapid iterations and debugging.

Staging environment mimics the production environment as closely as possible. It is used for final testing before deployment to production. The staging environment helps identify

any issues that might not have been caught during development and ensures that the application behaves as expected under production-like conditions.

Production environment is the live environment where the application is accessible to end-users. It is optimized for performance, reliability, and security. Changes to the production environment are made cautiously and are usually preceded by thorough testing in the development and staging environments.” [20]

3.2.4. Testing Methodology

This section outlines the tools and methodologies used for performance and load testing, as well as observability and monitoring practices.

Performance Testing

Performance testing evaluates the application's responsiveness and stability under various conditions. The tools and methodologies used for performance testing include:

- **Tools used** for simulating user interactions and load:
  - Apache AB (Apache Benchmark) [21] is a tool for benchmarking web server performance by simulating multiple user requests.
  - K6 [22] is a modern load testing tool that provides scripting capabilities to define complex testing scenarios.
- **Metrics Collected:** Response time, throughput, CPU and memory utilization, error rates, and latency.

Load Testing

Load testing assesses the application's behavior under high demand to ensure it can scale and remain stable. The key components of load testing include:

- **Scenarios:** Simulate peak traffic conditions, such as simultaneous device data submissions and multiple user interactions.
- **Objectives:** Identify the maximum load the system can handle, detect bottlenecks, and ensure the application scales appropriately.

Observability and Monitoring

Monitoring the application's performance during testing is crucial for identifying issues and ensuring smooth operation. The observability and monitoring tools used include:

- **Tools Used:** The Grafana and Prometheus stack for metrics collection and visualization.
- **Metrics Monitored:** System health, resource usage, application performance, and user activity.

Prometheus collects and stores metrics, while Grafana visualizes these metrics in customizable dashboards. This stack provides a comprehensive view of the application's performance and helps identify potential issues during testing.

4. Implementation and Deployment

4.1. Deploying Application on Azure Kuberentes Service (AKS)

This section demonstrates how Azure Kubernetes Service (AKS) is utilized to deploy and manage the "Online Boutique" microservices application effectively. This deployment focuses on configuration, setup, the deployment process, and addressing common challenges with practical solutions. Detailed steps and configuration files used for this deployment can be accessed in the GitHub repository linked in "Appendix" at the end of this document.

4.1.1. Configuration and Setup of Infrastructure

Setting up an AKS cluster involves selecting the appropriate region, VM size, and type. For this deployment, the Standard_A2_v2 SKU with 2 vCPUs and 4GB RAM for each of the two nodes is used. The OS disk size is set to 50GB SSD. This configuration provides a balance of cost-efficiency and performance, suitable for a demonstration environment. Using Terraform, the deployment in Azure includes the following resources:

```
resource "azurerm_resource_group" "resource_group" {
  name     = "${var.prefix}-rg"
  location = var.azure_location
}

resource "azurerm_kubernetes_cluster" "aks" {
  name                     = "${var.prefix}-aks"
  location                 =
azurerm_resource_group.resource_group.location
  resource_group_name      = azurerm_resource_group.resource_group.name
  dns_prefix               = "${var.prefix}-aks"
  sku_tier                 = "Standard"


  default_node_pool {
    name                 = "a2v2"
    node_count           = 2
    vm_size              = "Standard_A2_v2"
    type                 = "VirtualMachineScaleSets"
    os_disk_size_gb      = 50
  }

  identity {
    type = "SystemAssigned"
  }

  network_profile {
    network_plugin    = "kubenet"
    network_policy    = "calico"
    load_balancer_sku = "standard"
  }
}
```

Code 1. Terraform configuration script for deploying an Azure Kubernetes Service (AKS) cluster

This Terraform script shown in Code 1 defines the infrastructure for deploying an Azure Kubernetes Service (AKS) cluster along with its associated Azure Resource Group. The script is divided into four parts:

Azure Resource Group:

- An Azure Resource Group is created using a name that combines a user-defined prefix with "-rg".
- The location of the Resource Group is specified by the user in Terraform variables file.

Azure Kubernetes Service (AKS) Cluster:

1. Configuration:
    a. **Name**: The name of the AKS cluster is constructed using a prefix specified by the user followed by "-aks".
    b. **Location**: The location is inherited from the previously created resource group.
    c. **Resource Group**: The AKS cluster is created within the specified resource group.
    d. **DNS Prefix**: A DNS prefix for the AKS cluster is set, using the prefix specified by the user.
    e. **SKU Tier**: The cluster uses the "Standard" SKU tier, which is suitable for production workloads.
2. Node Pool Configuration
    a. **Name**: The default node pool is named "a2v2".
    b. **Node Count**: The node pool consists of 2 nodes.
    c. **VM Size**: The virtual machines in the node pool use the "Standard_A2_v2" size.
    d. **Type**: The nodes are part of a Virtual Machine Scale Set.
    e. **OS Disk Size**: Each node has a 50 GB OS disk.
3. Identity Configuration

   a. **Type**: The AKS cluster uses a "SystemAssigned" managed identity for resource access.
4. Network Profile
   a. **Network Plugin**: The cluster uses the "kubenet" [23] network plugin for basic networking.
   b. **Network Policy**: The cluster uses the "calico" [24] network policy for network security and policy enforcement.
   c. **Load Balancer SKU**: The cluster uses the "standard" SKU for the load balancer, which provides better performance and features compared to the basic SKU.

To achieve this configuration, the following **prerequisites and installation s**teps are followed:

Prerequisites:

- Terraform installed using Terraform official documentation
- Kubernetes cluster (AKS) [25]
- `kubectl` installed and configured to interact with the cluster
- Helm installed and configured to interact with the cluster

Installation Steps:

1. Clone the GitHub repository:

```
git clone https://github.com/ajanach/comparison-of-orchestration-
systems-for-microservices-applications.git
```

2. Navigate to the infrastructure directory:

```
cd aks/infrastructure
```

3. Copy the example Terraform variables file:

```
cp terraform.tfvars.example terraform.tfvars
```

   The `terraform.tfvars` file should be edited with the correct credentials for the Terraform service provider on Azure. This file contains configuration parameters such as subscription ID, client ID, client secret, and tenant ID necessary for Terraform to manage resources on Azure.

4. Initialize Terraform:

```
terraform init
```

5. Plan the Terraform deployment:

```
terraform plan
```

The `terraform plan` command creates an execution plan, showing what actions Terraform will take to achieve the desired state defined in the configuration files. It verify that the configuration is correct before making any changes.

6. Apply the Terraform plan:

```
terraform apply
```

   This command applies the changes required to reach the desired state of the configuration, creating and configuring the AKS cluster.

7. Configure kubectl:

```
mkdir ~/.kube
terraform output -raw kube_config > ~/.kube/config
```

8. Verify the nodes:

```
kubectl get nodes
```

The output of above command should be similar to the following example output:

```
NAME                              STATUS   ROLES   AGE     VERSION
aks-a2v2-96764596-vmss000000   Ready    agent   3m36s   v1.28.10
aks-a2v2-96764596-vmss000001   Ready    agent   3m41s   v1.28.10
```

AKS infrastructure overview:

After completing the installation steps, it's essential to understand the key components involved in a typical AKS deployment. The following diagram in Figure 4 provides a generic overview of an Azure Kubernetes Service (AKS) deployment, illustrating the critical infrastructure elements that ensure the smooth operation of Kubernetes clusters:

1. **Azure Container Registry (ACR)** is used to store and manage Docker container images that are deployed to the AKS cluster.
2. **Azure Key Vault** securely stores and manages sensitive information such as secrets, keys, and certificates, which are necessary for accessing various resources within the AKS environment.
3. **Kubernetes API Server** is the central management entity of the Kubernetes control plane. It processes requests to and from the Kubernetes cluster and ensures the desired state is maintained.
4. **Log Analytics Workspace** collects and analyzes data generated by resources within the AKS environment, providing insights into performance, security, and operational issues.

5. **PostgreSQL Database** is used to store relational data required by the "Online Boutique" application. It serves as the backend database for various microservices.
6. **User and System Node Pools** are groups of nodes within the AKS cluster. User node pools handle application workloads, while system node pools manage system-level processes and infrastructure services.

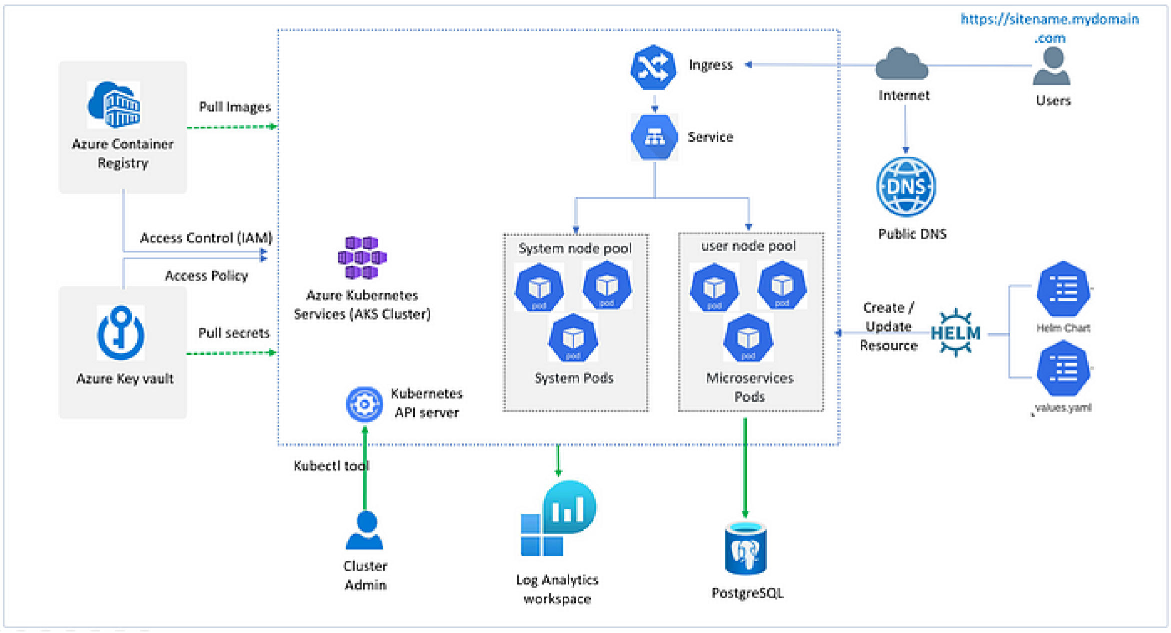


Figure 4. Create Azure Kubernetes Service (AKS) using Terraform. Source: [26].

4.1.2. Deployment Process of Application Services

A standardized deployment process is utilized across both AKS and K3S to ensure consistency and reproducibility. The application is deployed using YAML manifests to configure Kubernetes resources. This approach allows for version control and easy modifications to the deployment configuration

1. Navigate to the application services directory:

```
cd aks/application-services
```

2. Apply the Kubernetes manifests:

```
kubectl apply -f kubernetes-manifests.yaml
```

   The `kubernetes-manifests.yaml` file contains definitions for all the Kubernetes resources required to deploy the "Online Boutique" application, including deployments, services, and ingress controllers.

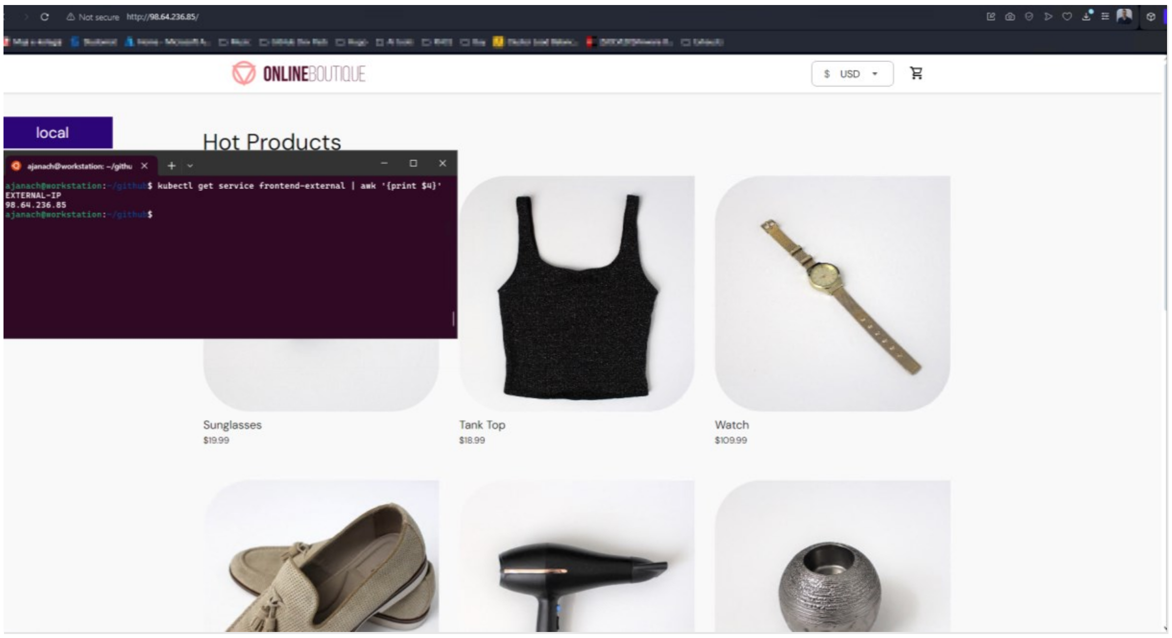
3. Retrieve the external IP address to access the web GUI:

```
kubectl get service frontend-external | awk '{print $4}'
```

4. Open the browser and navigate to:

   ```
   http://<external_IP>
   ```

   Note: As shown in Figure 5, it may take a few minutes for the platform to be online.



```
ajanach@workstation:~/github$ kubectl get service frontend-external | awk '{print $4}'
EXTERNAL-IP
98.64.236.85
ajanach@workstation:~/github$
```

Figure 5. Retrieving the external IP address and accessing the “Online Boutique” application through a browser on Azure Kubernetes Service (AKS)

4.1.3. Deployment Process of System Services

To ensure a comprehensive monitoring and management solution for the AKS environment, the kube-prometheus-stack is deployed using Helm. This setup includes Prometheus for monitoring, Grafana for visualization, and other essential components.

Installation Steps:

1. Add Prometheus Community Helm Repository:

   ```
   helm repo add prometheus-community https://prometheus-
   community.github.io/helm-charts
   helm repo update
   ```

2. Install kube-prometheus-stack:

   ```
   helm install kube-prometheus-stack prometheus-community/kube-
   prometheus-stack --version 61.3.1
   ```

   This command installs the kube-prometheus-stack Helm chart, which includes Prometheus, Grafana, and other monitoring tools.

3. Verify Installation:

```
kubectl --namespace default get pods -l "release=kube-prometheus-
stack"
```

This command checks the status of the installed pods to ensure they are running correctly.

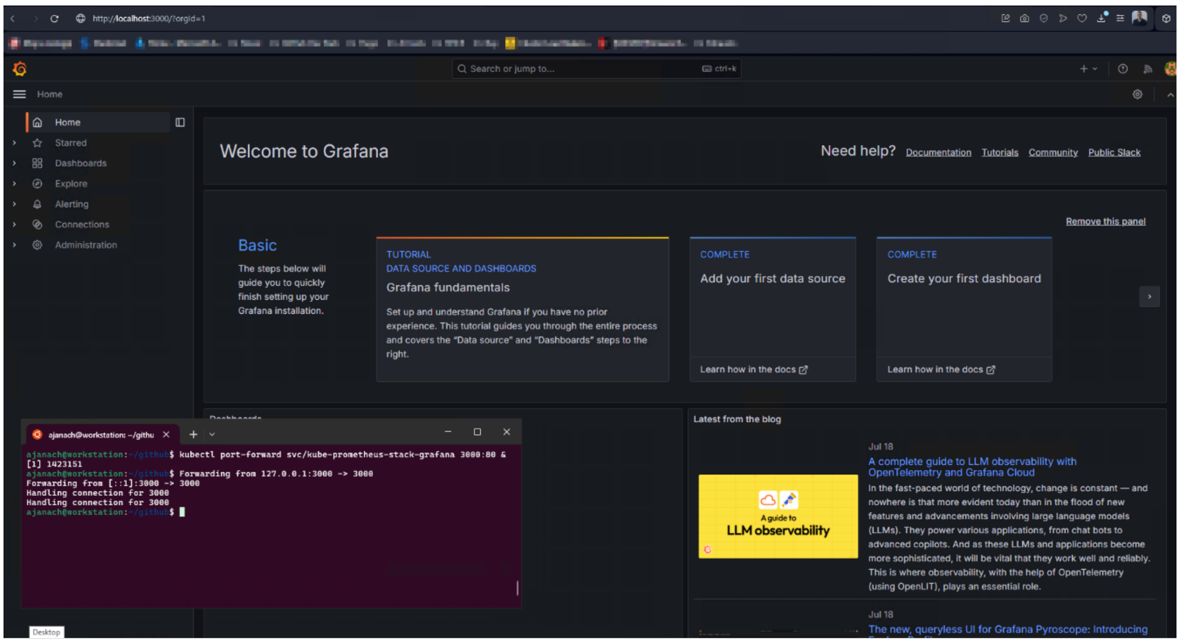
4. Set port forwarding to access Grafana web GUI:

```
kubectl port-forward svc/kube-prometheus-stack-grafana 3000:80
&
```

This sets up port forwarding to access the Grafana web GUI. After setting up port forwarding, open a browser and navigate to http://localhost:3000.

As shown in Figure 6, you will be prompted for a username and password:

- **Username**: admin
- **Password**: prom-operator



The steps below will guide you to quickly finish setting up your Grafana installation.

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

Jul 18
A complete guide to LLM observability with OpenTelemetry and Grafana Cloud
In the fast-paced world of technology, change is constant — and nowhere is that more evident today than in the flood of new features and advancements involving large language models (LLMs). They power various applications, from chat bots to advanced copilots. And as these LLMs and applications become more sophisticated, it will be vital that they work well and reliably. This is where observability, with the help of OpenTelemetry (using OpenLIT), plays an essential role.

Jul 18
The new, queryless UI for Grafana Pyroscope: Introducing

```
ajanach@workstation:~/github$ kubectl port-forward svc/kube-prometheus-stack-grafana 3000:80 &
[1] 1423151
ajanach@workstation:~/github$ Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
Handling connection for 3000
Handling connection for 3000
ajanach@workstation:~/github$
```

Figure 6. Accessing the Grafana dashboard through port forwarding on Azure Kubernetes Service (AKS)

4.1.4. Challenges and Solutions

Deploying and managing microservices applications on AKS comes with several challenges. Next, challenges are discussed along with possible solutions in real-world scenarios.

1. Secret Management

Challenge: Managing sensitive information such as API keys, database credentials, and certificates securely is a critical challenge. Ensuring that these secrets are not exposed and are managed properly across various environments is essential for maintaining the security of the application.

Solution: Kubernetes Secrets [27] can be used to securely store and manage sensitive information. These secrets are encrypted at rest and can be accessed by the applications running in the cluster, ensuring that sensitive information is kept secure. Additionally, Azure Key Vault [28] can be integrated with Kubernetes to manage secrets, keys, and certificates centrally, offering an extra layer of security and ease of management across different environments.

2. Load Balancing

Challenge: Efficient distribution of traffic across multiple service instances is essential for maintaining high performance and availability. This process involves managing traffic flow, balancing loads, and ensuring seamless failovers to prevent service disruptions.

Solution: Ingress Controllers can be implemented to manage traffic routing effectively. The NGINX Ingress Controller [17] is commonly used for this purpose, providing robust load balancing, SSL termination, and routing capabilities.

3. Autoscaling

Challenge: Automatically adjusting the number of running instances based on demand is crucial for handling varying loads and ensuring optimal resource utilization. This involves scaling out by adding more instances during high traffic and scaling in by reducing instances when demand decreases.

Solution: The Horizontal Pod Autoscaler (HPA) [29] can be utilized to automatically scale applications based on resource utilization metrics such as CPU and memory usage. This ensures that the application can handle increased load without manual intervention.

4. Resource Management

Challenge: Optimizing the allocation of resources to prevent overuse or underutilization is important for cost-efficiency and performance. Proper resource management ensures that applications have enough resources to function correctly while avoiding waste.

Solution: Resource quotas and limits can be set to ensure that applications have the necessary resources while preventing overuse. This helps maintain a balance between resource allocation and application performance.

5. Deployment Rollbacks

Challenge: Handling issues that arise during deployments and needing to roll back to a previous stable state is a common challenge. Ensuring that deployments can be rolled back quickly and without disruption is critical for maintaining service availability.

Solution: Use Kubernetes' built-in deployment strategies, such as rolling updates and rollbacks, to manage application updates. This allows for safe and efficient deployment processes, with the ability to revert to previous versions if issues are encountered.

6. Monitoring and Observability

Challenge: Implementing robust monitoring and observability to track system performance, health, and security is essential. This includes setting up metrics collection, logging, and alerting to proactively identify and resolve issues.

Solution: Utilize the Grafana and Prometheus stack to monitor the application and infrastructure. Prometheus collects and stores metrics, while Grafana visualizes these metrics in customizable dashboards. Alerting is configured within Prometheus using the Grafana dashboard GUI, enabling easy setup and management of alerts that automatically notify the relevant teams when predefined thresholds are exceeded, ensuring timely responses to potential issues.

7. Cluster Management

Challenge: Managing the Kubernetes cluster itself, including node health, scaling, and upgrades, can be complex. Ensuring that the cluster remains healthy and up to date requires continuous attention and maintenance.

Solution: Use Azure's managed AKS features, which handle many cluster management tasks automatically, such as node upgrades and patching. Additionally, leverage Azure Monitor [30] and Azure Advisor [31] to gain insights and recommendations on cluster health and performance.

8. Network Configuration

Challenge: Configuring and managing the network settings for AKS, including VNet integration, network policies, and ingress/egress controls, can be challenging. Ensuring secure and efficient network traffic flow is crucial.

Solution: Use Azure's native VNet integration for AKS to simplify network configuration. Implement Kubernetes Network Policies to control traffic flow between pods and use Azure Firewall or Network Security Groups (NSGs) to manage ingress and egress traffic securely.

9. Identity and Access Management

Challenge: Managing access to the AKS cluster and its resources securely is vital. This includes controlling who can access the cluster, what actions they can perform, and ensuring that applications running in the cluster can access necessary resources securely.

Solution: Utilize Azure Active Directory (AAD) integration with AKS for robust identity and access management. Use Role-Based Access Control (RBAC) to define fine-grained access controls and leverage Managed Identities for secure access to Azure resources from applications running in the cluster.

4.2. Deploying on K3S

This section details the process of deploying and managing the "Online Boutique" microservices application using K3S. The following subsections cover the configuration and setup of the infrastructure, the deployment process for application and system services, and the challenges encountered along with their solutions. Detailed steps and configuration files used for this deployment can be accessed in the GitHub repository linked in "Appendix" at the end of this document.

4.2.1. Configuration and Setup of Infrastructure

Virtual Machine Configuration in vSphere

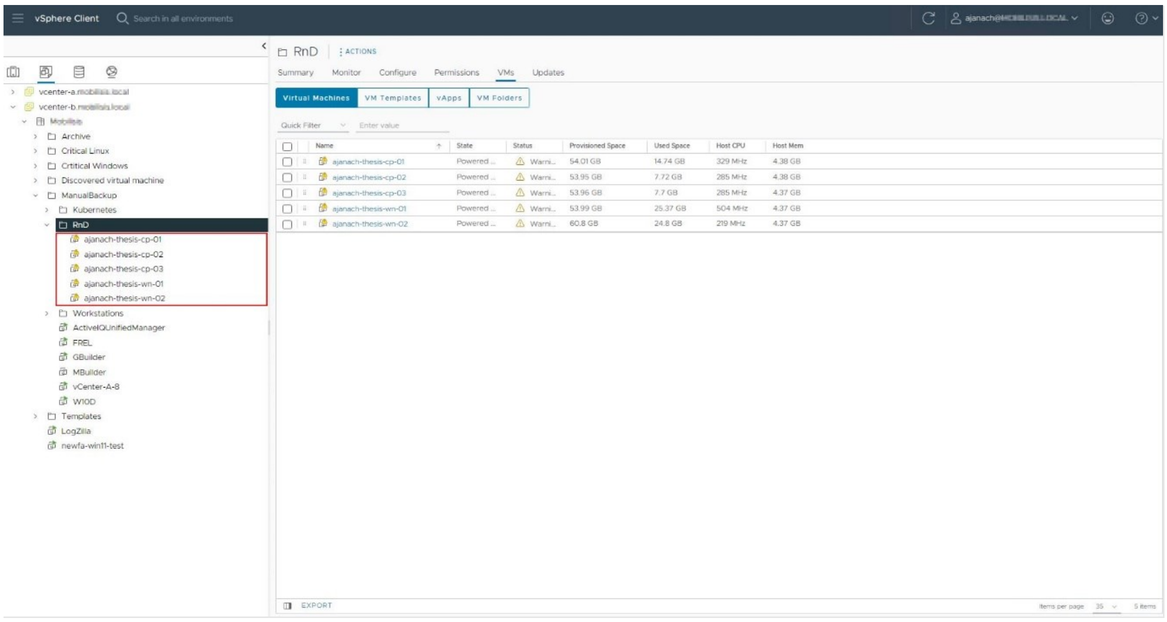
The K3S Kubernetes cluster is set up on virtual machines configured in vSphere, as shown in Table 9. The configuration for these VMs is outlined in the table below:

Table 9. Configuration of virtual machines for the K3S Kubernetes cluster in vSphere

Host	CPUs	Total RAM (GB)	OS Disk Size (GB)	IP Address
ajanach-thesis-cp-01	2	4.06	50	10.10.48.151
ajanach-thesis-cp-02	2	4.06	50	10.10.48.152

ajanach-thesis-cp-03	2	4.06	50	10.10.48.153
ajanach-thesis-wn-01	2	4.06	50	10.10.48.154
ajanach-thesis-wn-02	2	4.06	50	10.10.48.155

The vCenter web UI on Figure 7 illustrates the setup process of a K3S Kubernetes cluster on virtual machines (VMs) configured within a vSphere environment. This visual representation demonstrates the creation and configuration of the five VMs.



Name	State	Status	Provisioned Space	Used Space	Host CPU	Host Mem
ajanach-thesis-cp-01	Powered ...	Warni...	54.01 GB	14.74 GB	329 MHz	4.38 GB
ajanach-thesis-cp-02	Powered ...	Warni...	53.95 GB	7.72 GB	285 MHz	4.38 GB
ajanach-thesis-cp-03	Powered ...	Warni...	53.96 GB	7.7 GB	285 MHz	4.37 GB
ajanach-thesis-wn-01	Powered ...	Warni...	53.99 GB	25.37 GB	504 MHz	4.37 GB
ajanach-thesis-wn-02	Powered ...	Warni...	60.8 GB	24.8 GB	219 MHz	4.37 GB

Figure 7. vCenter Web UI Displaying the Creation of Five Virtual Machines for the K3S Kubernetes Cluster

Ansible Setup

Prerequisites:

- Ansible installed on workstation machine,
- K3s cluster [25],
- `kubectl` installed and configured,
- Helm installed and configured.

Ansible [16] is used to automate the configuration and deployment of the K3S cluster. The following steps outline the Ansible setup:

1. Clone the GitHub repository:

```
git clone https://github.com/ajanach/comparison-of-orchestration-
systems-for-microservices-applications.git
```

2. Navigate to the infrastructure directory:

```
cd k3s/infrastructure
```

3. Edit the `inventory` file, which specifies the IP addresses of the control plane and worker nodes in the cluster.

```
[master_1]
ajanach-thesis-cp-01 ansible_host=<IP_CONTROL_PLANE_01>

[master_2]
ajanach-thesis-cp-02 ansible_host=<IP_CONTROL_PLANE_02>
ajanach-thesis-cp-03 ansible_host=<IP_CONTROL_PLANE_03>

[worker]
ajanach-thesis-wn-01 ansible_host=<IP_WORKER_01>
ajanach-thesis-wn-02 ansible_host=<IP_WORKER_02>

[production:children]
master_1
master_2
worker
```

4. Edit `ansible.cfg`. This configuration file sets the defaults for Ansible operations, including the inventory file location and user credentials.

```
[defaults]
inventory = inventory
remote_user = <USERNAME>
ask_pass = false

[privilege_escalation]
become = true
become_user = root
become_method = sudo
become_ask_pass = false
```

5. Edit `vars.yaml`. This file includes variables used by the Ansible playbooks, such as the IP address of the K3S server.

```
k3s_server_ip: "<IP_CONTROL_PLANE_01>"
```

6. Test reachability. Use Ansible to ping all specified nodes and ensure they are reachable.

```
cd k3s/infrastructure
ansible -i inventory all -m ping
```

7. Run Ansible playbook to install K3S. This playbook installs K3S on the specified nodes in inventory.

```
ansible-playbook -i inventory k3s_install.yaml
```

8. Verify Cluster Availability: Use kubectl to check the status of the nodes and ensure they are ready.

```
kubectl get nodes
```

Example output:

```
NAME                  STATUS   ROLES                        AGE     VERSION
ajanach-thesis-cp-01  Ready    control-plane,etcd,master    3d17h   v1.29.6+k3s2
ajanach-thesis-cp-02  Ready    control-plane,etcd,master    3d17h   v1.29.6+k3s2
ajanach-thesis-cp-03  Ready    control-plane,etcd,master    3d17h   v1.29.6+k3s2
ajanach-thesis-wn-01  Ready    <none>                       3d17h   v1.29.6+k3s2
ajanach-thesis-wn-02  Ready    <none>                       3d17h   v1.29.6+k3s2
```

The figure Figure 8 illustrates the architecture of the K3s setup used in this specific case scenario, showing how the control plane and worker nodes are configured within the cluster. It reflects the deployment strategy tailored to the current requirements, providing a clear view of the infrastructure as implemented.

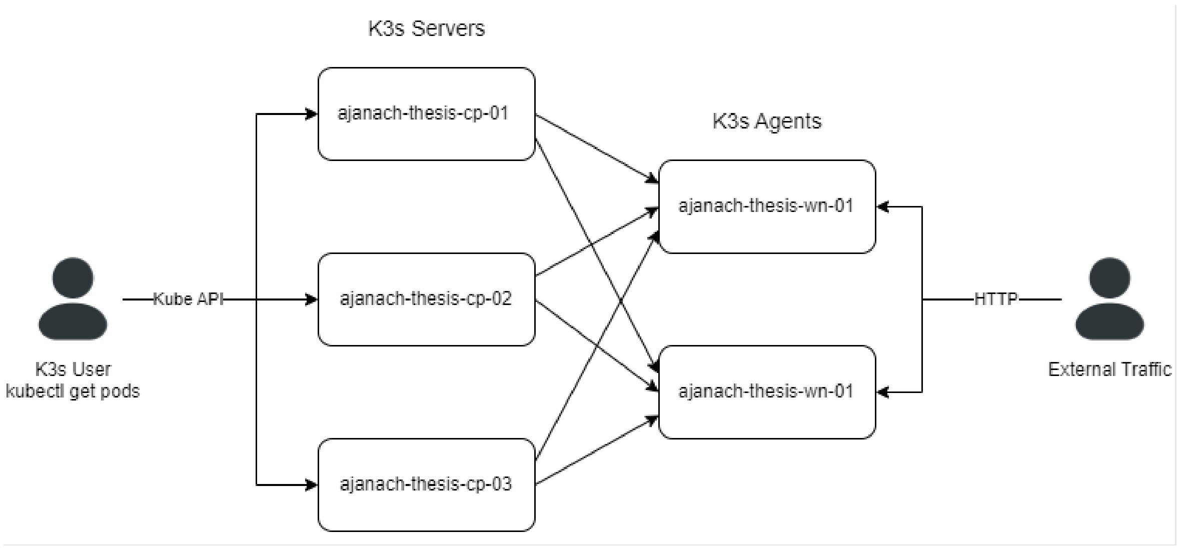


Figure 8. K3s Infrastructure for Online Boutique Deployment

In contrast to the specific setup shown in Figure 8, Figure 9 presents a recommended high-availability architecture for K3s, suitable for production environments. This setup includes multiple control planes and worker nodes to enhance resilience and scalability. While Figure 8 details the configuration used in this case, Figure 9 offers a broader, more robust approach to deploying K3s in scenarios that demand higher availability and fault tolerance.

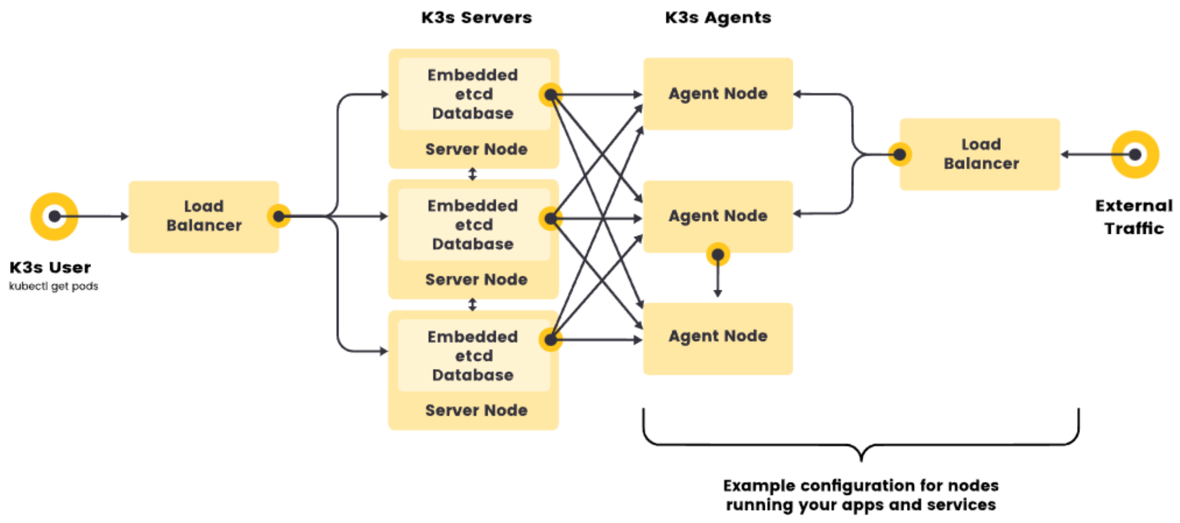


Figure 9. Recommended High-Availability K3s Setup. Source: [32].

4.2.2. Deployment Process of Application Services

Deploying the "Online Boutique" application on the K3S cluster involves applying Kubernetes manifests to configure and launch the services. It is important to note that the YAML manifests used for deploying the application are the same for both AKS and K3S, ensuring consistency across different environments. The following steps outline the deployment process:

1. Navigate to the Application Services Directory:

   ```
   cd k3s/application-services
   ```

2. Apply the Kubernetes Manifests:

   ```
   kubectl apply -f kubernetes-manifests.yaml
   ```

   This command applies the Kubernetes manifests, creating the necessary resources such as deployments, services, and configurations for the application.

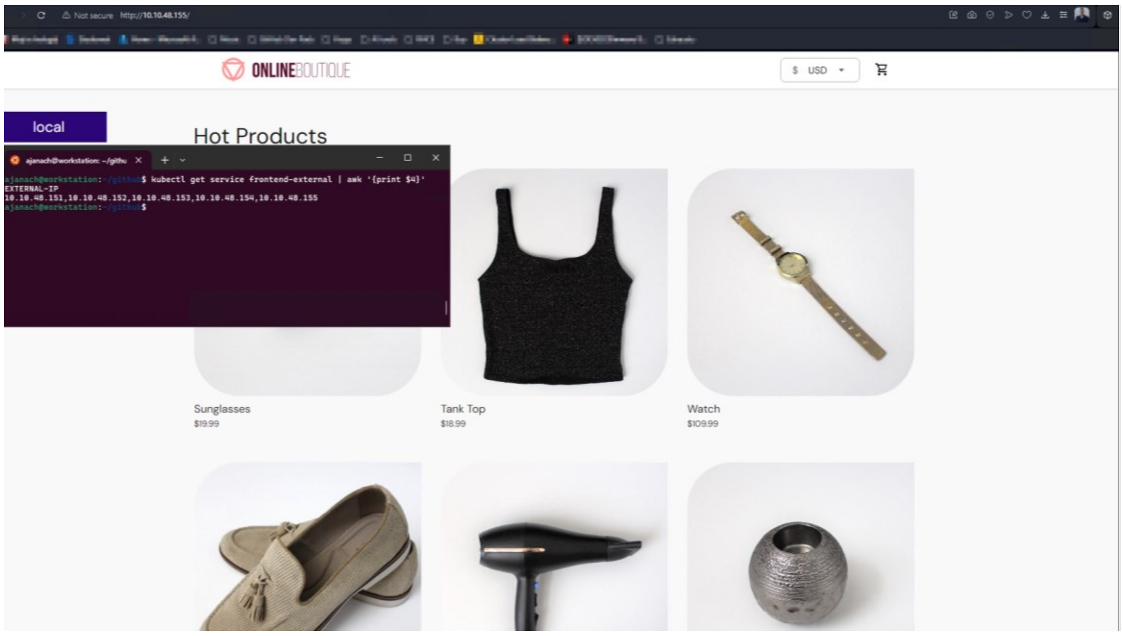
3. Retrieve the External IP Address to Access the Web GUI:

   ```
   kubectl get service frontend-external | awk '{print $4}'
   ```

   A list of IP addresses will be provided, each representing the IP address of a node. This means that the application is available through the IP address of any node.

4. Access the Application:

   Open a browser and navigate to `http://<external_IP>`. As shown in Figure 10, it may take a few minutes for the application to be online.



```
ajanach@workstation:~/github$ kubectl get service frontend-external | awk '{print $4}'
EXTERNAL-IP
10.10.48.151,10.10.48.152,10.10.48.153,10.10.48.154,10.10.48.155
ajanach@workstation:~/github$
```

Figure 10. Retrieving the external IP address and accessing the "Online Boutique" application through a browser on K3s

4.2.3. Deployment Process of System Services

The process of deploying system services on K3S is the same as described in Chapter 4.1.3 for AKS. This involves setting up the kube-prometheus-stack using Helm to ensure comprehensive monitoring and management of the K3S environment. The deployment includes Prometheus for monitoring, Grafana for visualization, and other essential components.

4.2.4. Challenges and Solutions

Deploying and managing microservices applications on K3S comes with several challenges. Here, real challenges are discussed along with solutions to overcome them, including real-world scenarios.

1. Resource Constraints

Challenge: K3S is designed for lightweight environments, but resource constraints can still pose challenges. Ensuring adequate resource allocation and monitoring is crucial.

Solution: Optimize resource usage by configuring appropriate resource limits and requests for each microservice. Use monitoring tools like Prometheus and Grafana to track resource usage and identify bottlenecks and underutilization.

2. Networking Complexity

Challenge: Managing network configurations and ensuring reliable communication between services can be complex.

Solution: Utilize the built-in networking capabilities of K3S, such as the Flannel [33] or Calico [24] network plugins. Ensure proper configuration of network policies and security groups to facilitate secure and reliable communication.

3. High Availability

Challenge: Achieving high availability in a resource-constrained environment can be difficult, especially with limited nodes.

Solution: Deploy multiple control plane nodes to ensure redundancy. Configure K3S to use an external database for storing cluster state, enhancing resilience and availability.

4. Security

Challenge: Ensuring robust security in a lightweight Kubernetes environment is essential.

Solution: Implement best practices for securing the cluster, including using role-based access control (RBAC), enabling mutual TLS (mTLS) for service communication, and regularly updating and patching the cluster components.

5. Integration with CI/CD Pipelines

Challenge: Integrating K3S with continuous integration and continuous deployment (CI/CD) pipelines can be challenging.

Solution: Use tools like Jenkins, GitLab CI, or GitHub Actions to automate the deployment process. Configure these tools to interact with the K3S cluster using kubectl and Helm, ensuring a smooth and automated deployment process.

5. Performance and Cost Analysis

5.1. Resource Utilization

This section evaluates the resource utilization of Azure Kubernetes Service (AKS) and K3S clusters. The analysis focuses on CPU, memory usage, storage, and network performance, emphasizing the different architectures and configurations that contribute to resource consumption. The measurements are taken during the idle state of the clusters, with the "Online Boutique" application and necessary system services deployed but not actively processing user interactions.

AKS, being a cloud-based Kubernetes service, runs additional operational and management containers compared to K3S. This results in different resource utilization patterns between the two. AKS utilizes Calico as its Container Network Interface (CNI), which provides robust networking capabilities suited for complex environments, whereas K3S employs Flannel, a lightweight CNI, contributing to lower system overhead [34].

5.1.1. Comparison of Pods and Containers

A critical factor influencing resource utilization and performance in Kubernetes clusters is the number and type of pods and containers running within the environment. This section provides a detailed comparison between Azure Kubernetes Service (AKS) and K3S, focusing on the distinct workloads managed by each platform. By examining the specific pods and containers deployed, we can gain insights into how each cluster is configured and the implications this has on resource consumption.

In AKS, the cluster runs a greater number of system-related pods, which include cloud-specific components such as monitoring agents, network management tools, and storage interface drivers. These additional pods are integral to the cloud-based operations and management features provided by Azure. Opposite, K3S, being a lightweight Kubernetes distribution, is designed with minimal overhead and runs fewer system pods. The absence of cloud-specific containers in K3S leads to lower baseline resource usage, making it an ideal choice for environments where efficiency and simplicity are paramount.

Example of AKS Pods vs. K3S Pods

The Table 10 compares the pods running in AKS and K3S, showcasing both the similarities and differences in the workloads they manage:

Table 10. Detailed Comparison of Pod and Container Distribution Between Azure Kubernetes Service (AKS) and K3S

Component	**AKS (Azure Kubernetes Service)**	**K3S (Lightweight Kubernetes)**
Total Number of Pods	42 Pods	30 Pods
Application Pods of "Online Boutique"	15 Pods	15 Pods
System Services	7 Pods (e.g., kube-prometheus-stack)	7 Pods (e.g., kube-prometheus-stack)
Cloud-Specific Components	Yes (e.g., cloud node managers, CSI drivers, network management tools)	No
Network Management Tools	Yes (e.g., Calico)	Yes (e.g., Flannel)
Storage Interface Drivers	Yes (e.g., AzureDisk CSI, AzureFile CSI)	No
Baseline Resource Usage	Higher (due to additional cloud-specific containers)	Lower (minimal overhead)
Kubernetes System Pods	20 Pods (e.g., kube-proxy, metrics-server, coredns)	8 Pods (e.g., kube-proxy, metrics-server, coredns)

5.1.2. CPU and Memory Usage

The analysis of CPU and memory utilization provides insight into how each platform manages workloads and optimizes resource use. The Table 11 and Table 12 summarize the key metrics for each worker node in both clusters.

Table 11. AKS Cluster Resource Utilization (Idle State with "Online Boutique" Deployed)

Node	**CPU Busy (%)**	**System Load (%)**	**RAM Used (%)**	**Root FS Used (%)**
Worker Node 1	42.30	98.50	54.10	54.50
Worker Node 2	23.30	99.50	36.00	50.90

Table 12. K3S Cluster Resource Utilization (Idle State with "Online Boutique" Deployed)

Node	**CPU Busy (%)**	**System Load (%)**	**RAM Used (%)**	**Root FS Used (%)**
Worker Node 1	12.50	17.50	62.60	33.80
Worker Node 2	3.70	5.50	42.90	18.10

AKS Nodes: The AKS nodes exhibit higher system loads, with Worker Node 2 reaching 99.5%, despite lower CPU utilization. This suggests that the additional containers and the Calico CNI contribute significantly to overhead. In AKS, there are 42 running pods across the cluster, which include operational containers such as cloud node managers, CSI (Container Storage Interface) drivers, and monitoring agents. These extra containers are essential for cloud-specific operations and management, leading to increased resource consumption

K3S Nodes: K3S nodes demonstrate lower CPU, and system loads due to Flannel's lightweight nature and the absence of additional cloud-based operational containers. The K3S cluster has 30 running pods, which is fewer than AKS, reflecting the absence of those additional cloud-specific components. Despite this, K3S nodes show higher RAM usage, which suggests a different memory allocation strategy that potentially enhances real-time application performance by allocating more resources to active processes.

5.1.3. Storage and Network Performance

Storage and network performance are critical factors in understanding how efficiently data is handled and transferred across the cluster. The metrics are analyzed to identify key differences in performance between AKS and K3S, as shown in Table 13.

Table 13. Storage and Network Performance (Idle State with “Online Boutique” Deployed)

Cluster	**Node**	**Root FS Used (%)**	**Network Traffic Insights**
AKS	Worker Node 1	54.50	Moderate activity with occasional peaks
AKS	Worker Node 2	50.90	Consistently moderate to high traffic
K3S	Worker Node 1	33.80	Low variability, minimal traffic
K3S	Worker Node 2	18.10	Stable and minimal traffic

Storage Utilization: AKS nodes have higher filesystem usage due to the additional components and services required for cloud operations, such as persistent volume management and network overlays. In contrast, K3S nodes maintain lower filesystem usage, benefiting from the absence of these additional layers.

Network Traffic: The AKS nodes experience more network traffic variability, attributed to the extensive inter-service communications facilitated by Calico. This variability could impact latency and performance in data-intensive applications. K3S's network traffic is more stable and lower due to Flannel's simplified network operations and fewer services running concurrently.

The Figure 11 and Figure 12 demonstrate detailed metrics from Grafana dashboards to provide a visual representation of the discussed metrics.

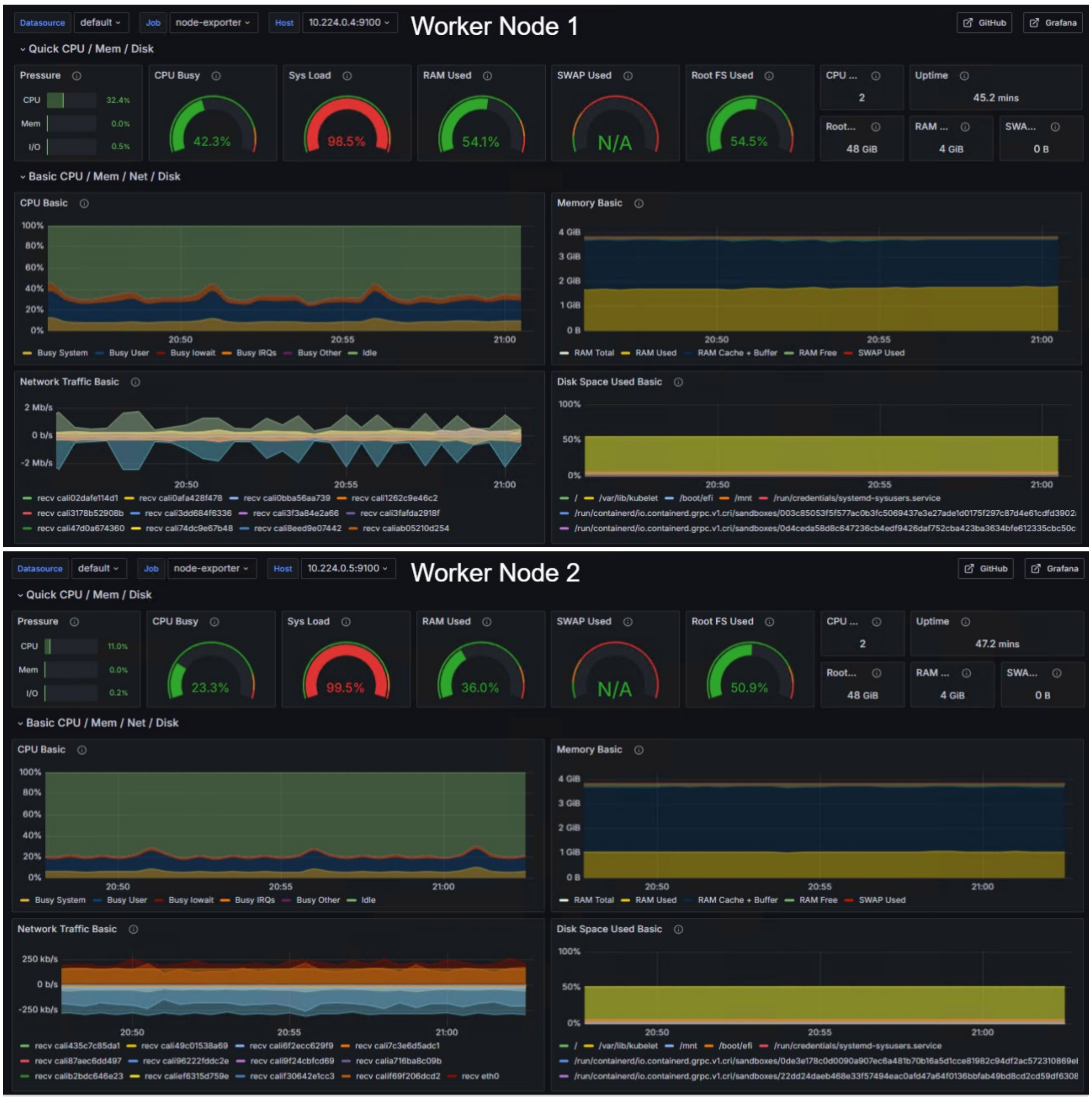


Figure 11. Grafana AKS Cluster Resource Utilization

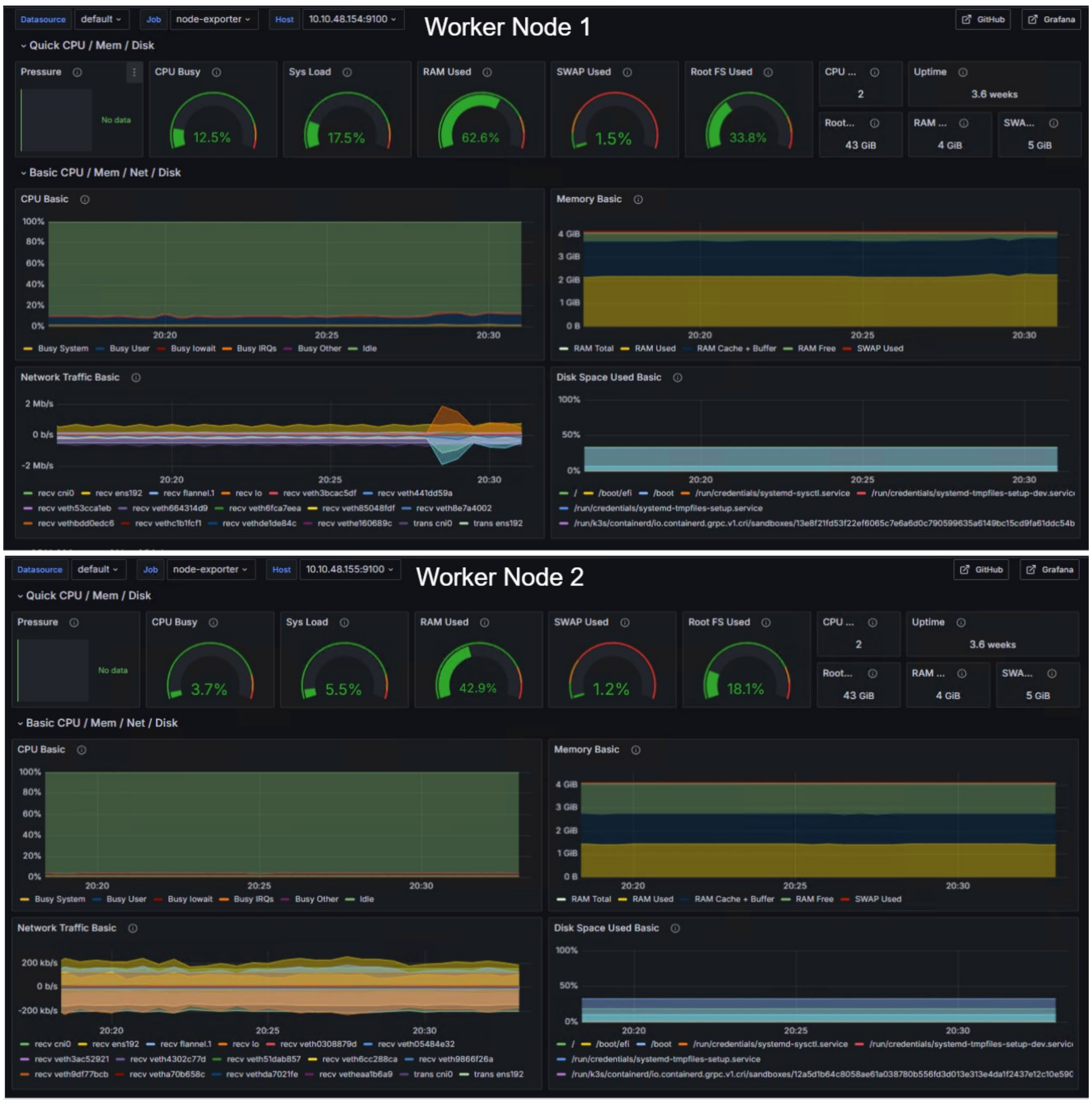


Figure 12. Grafana K3S Cluster Resource Utilization

5.2. Benchmark Results

This section evaluates the operational performance of AKS and K3S using three primary tools: Apache AB, K6, and Sysbench. These tools measure various metrics, including requests per second, response times, and system benchmarks. The analysis identifies key differences in performance and scalability between the two platforms. The configuration includes:

- **Apache Bench (AB):** Simulates high concurrency levels to evaluate the maximum request-handling capabilities of each platform.
- **K6:** Evaluates the platforms under various virtual user (VU) loads, simulating real-world traffic patterns.

- **Sysbench:** Measures CPU and memory performance to assess the raw computational capacity available to each cluster.

Test Setup:

The performance tests are conducted in a controlled environment to ensure comparability between AKS and K3S. Both platforms are configured similarly, with two worker nodes each running the "Online Boutique" microservices application. For more realistic load testing, one container for the front-end service is deployed on each worker node, ensuring that both nodes experience the load during the tests.

5.2.1. Apache AB Results

The AB tests are conducted with 1,000 requests at a concurrency level of 100. This setup is designed to evaluate the request-handling capabilities of both AKS and K3S under a moderate load, ensuring that the application can effectively manage concurrent requests without being overwhelmed. The following metrics are recorded:

- **Average Requests per Second:** This metric indicates the number of requests successfully processed by the server each second.
- **Average Time per Request:** This metric measures the average time taken to manage a single request, reflecting the platform's responsiveness.

A detailed guide on setting up and running the AB command is available on GitHub. The command used for testing is:

```
ab -k -n 1000 -c 100 -l -H "Accept-Encoding: gzip, deflate"
http://57.153.130.215/
```

This command performs a performance test on the index page of the "Online Boutique" application, accessible via the URL http://57.153.130.215/. Replace http://57.153.130.215/ with the actual URL of your service. Below is a description of what each parameter does:

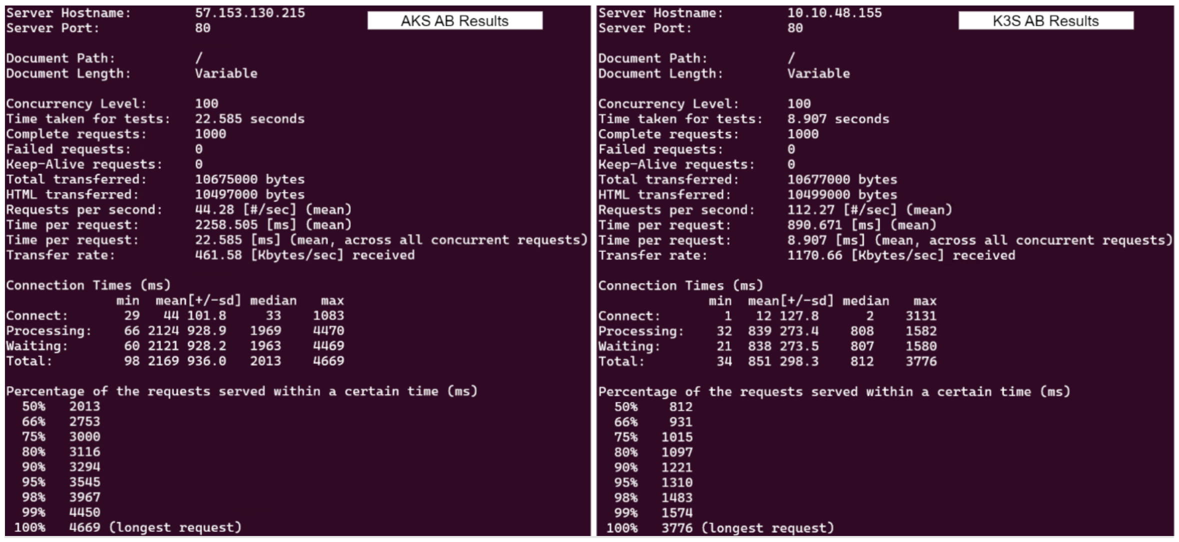
- **-k:** Sends the Keep-Alive header to maintain open connections.
- **-n 1000:** Number of requests to make.
- **-c 100**: Number of concurrent requests to simulate.
- **-l:** Accepts response sizes larger than the internal memory buffer.
- **-H Accept-Encoding: gzip, deflate**: Used to compress the output for more efficient data transfer. The use of mod-deflate compresses the text/html output, which significantly impacts the overall performance of the web server.

The results of the Apache AB benchmark are presented in Table 14:

Table 14. Apache AB Benchmark Results for AKS and K3S

Metric	**AKS**	**K3S**
Average requests per second	44.28	112.27
Average time per request (ms)	22.58	8.90

As shown in Figure 13, these results highlight the superior request-handling efficiency of K3S compared to AKS. K3S processed more than twice the number of requests per second on average and had a significantly lower average time per request.



```
Server Hostname:        57.153.130.215
Server Port:            80

Document Path:          /
Document Length:        Variable

Concurrency Level:      100
Time taken for tests:   22.585 seconds
Complete requests:      1000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      10675000 bytes
HTML transferred:       10497000 bytes
Requests per second:    44.28 [#/sec] (mean)
Time per request:       2258.505 [ms] (mean)
Time per request:       22.585 [ms] (mean, across all concurrent requests)
Transfer rate:          461.58 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:       29   44 101.8     33    1083
Processing:    66 2124 928.9   1969    4470
Waiting:       60 2121 928.2   1963    4469
Total:         98 2169 936.0   2013    4669

Percentage of the requests served within a certain time (ms)
  50%   2013
  66%   2753
  75%   3000
  80%   3116
  90%   3294
  95%   3545
  98%   3967
  99%   4450
 100%   4669 (longest request)
```

```
Server Hostname:        10.10.48.155
Server Port:            80

Document Path:          /
Document Length:        Variable

Concurrency Level:      100
Time taken for tests:   8.907 seconds
Complete requests:      1000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      10677000 bytes
HTML transferred:       10499000 bytes
Requests per second:    112.27 [#/sec] (mean)
Time per request:       890.671 [ms] (mean)
Time per request:       8.907 [ms] (mean, across all concurrent requests)
Transfer rate:          1170.66 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        1   12 127.8      2    3131
Processing:    32  839 273.4    808    1582
Waiting:       21  838 273.5    807    1580
Total:         34  851 298.3    812    3776

Percentage of the requests served within a certain time (ms)
  50%    812
  66%    931
  75%   1015
  80%   1097
  90%   1221
  95%   1310
  98%   1483
  99%   1574
 100%   3776 (longest request)
```

Figure 13. Apache AB Benchmark Results for AKS and K3S

The diagram on Figure 14 visually represents the Apache AB benchmark results, illustrating the differences in performance between the two platforms.

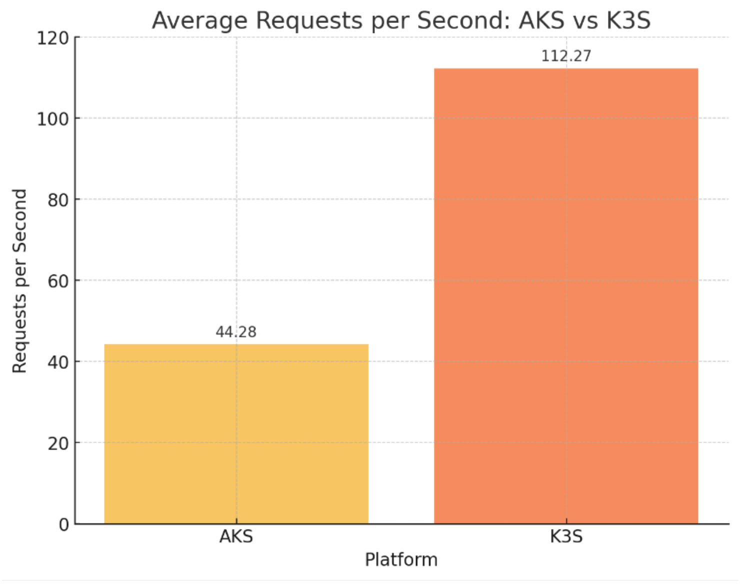


Figure 14. Comparison of Average Requests per Second Between AKS and K3S

System load analysis of Apache AB benchmark:

To further understand the impact of the AB testing on system resources, Grafana dashboards were used to monitor the load on both worker nodes in each cluster. Grafana, utilizing Prometheus and Node Exporter, provided real-time insights into CPU, memory, and network usage during the tests.

The results show that there was no significant impact on CPU, RAM, or disk usage in both AKS and K3S clusters during the tests. However, as shown in Figure 15 and Figure 16 both clusters experienced a considerable increase in network traffic.

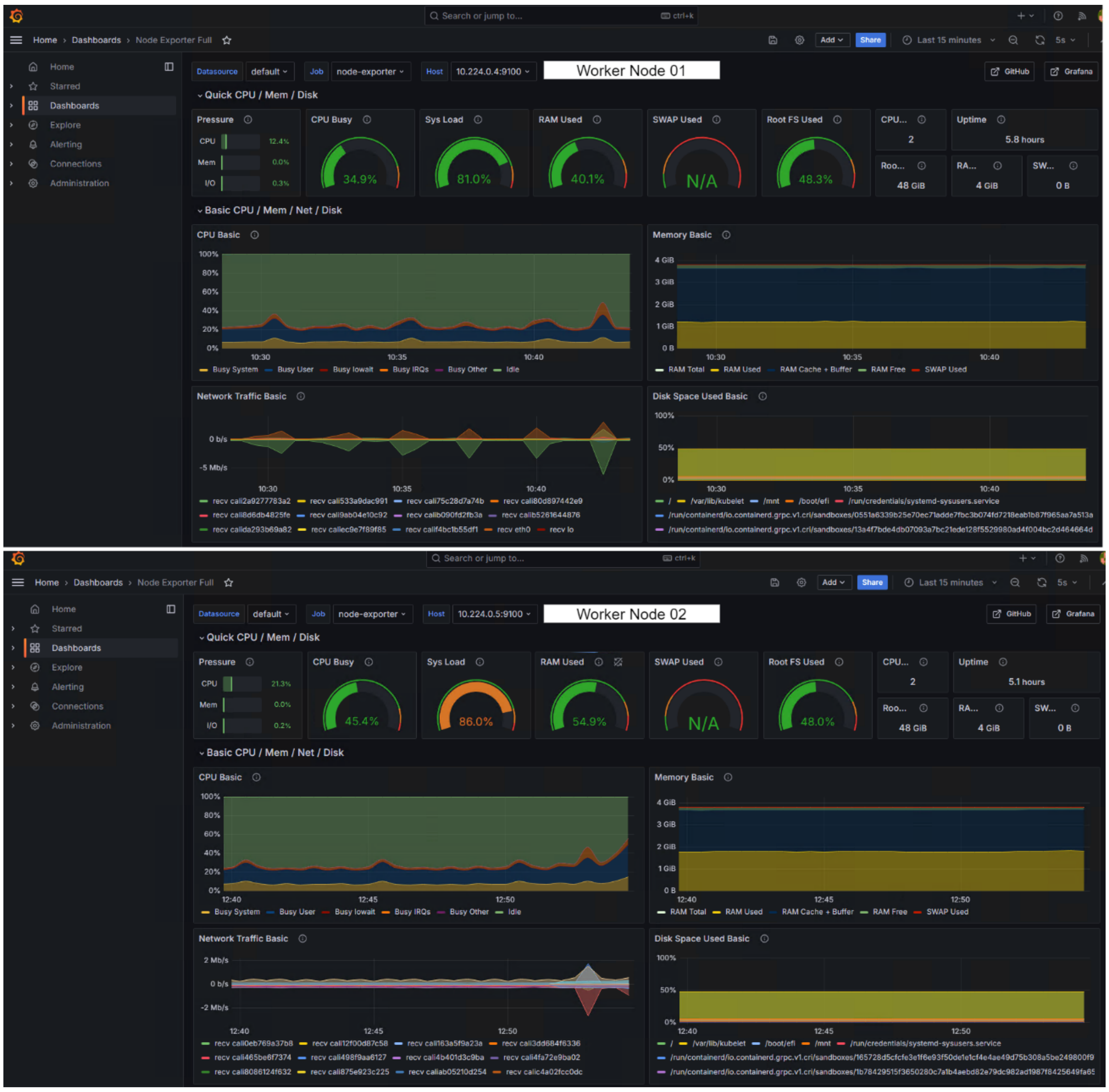


Figure 15. System Load on AKS Worker Nodes During Apache AB Testing

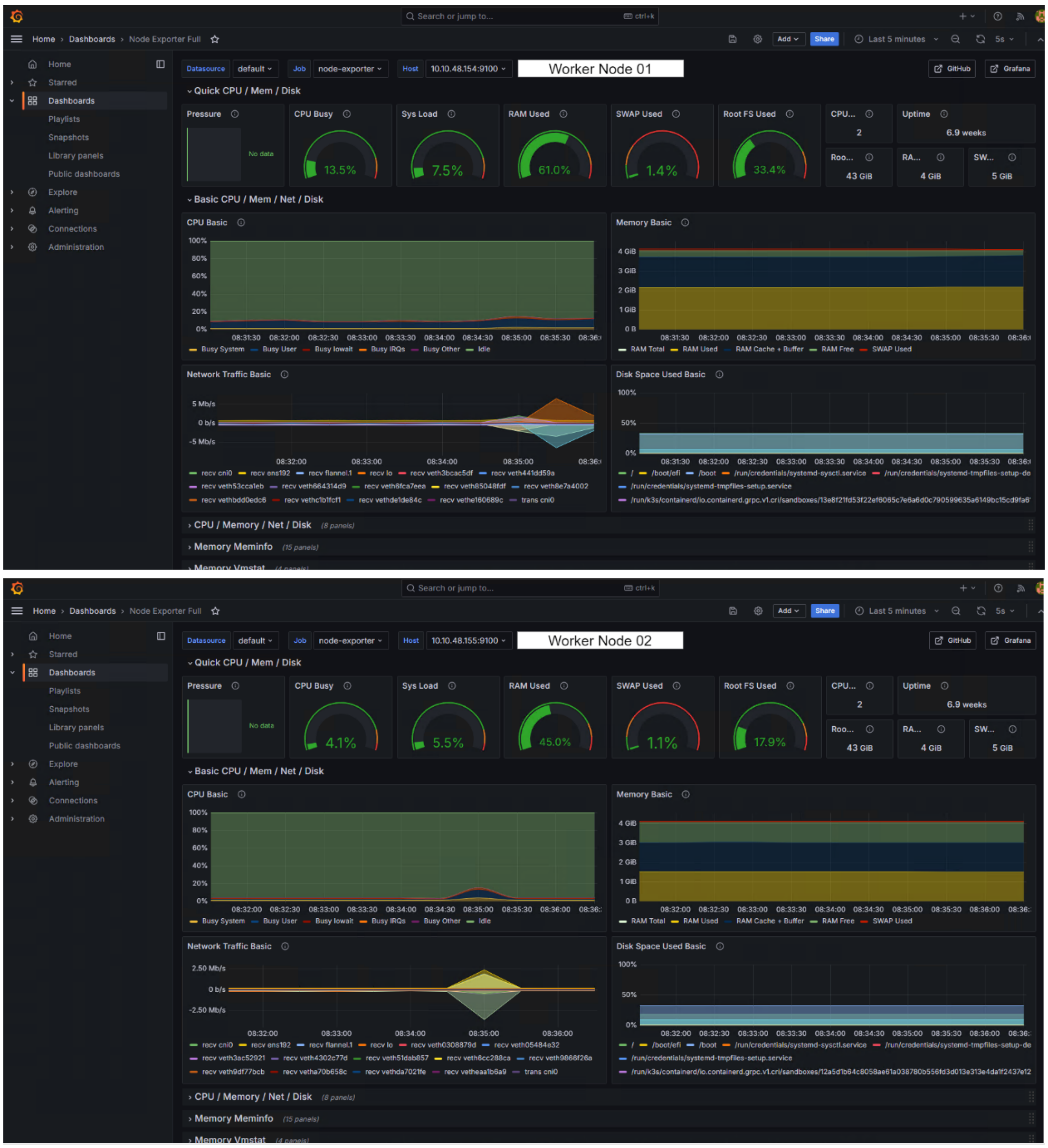


Figure 16. System Load on K3S Worker Nodes During Apache AB Testing

5.2.2. K6 Results

The goal of this test is to evaluate how well each orchestration platform can handle realistic workloads that mirror user behavior, which is critical for ensuring a robust user experience. Unlike synthetic benchmarks such as Apache AB, which primarily focus on request-handling capabilities under stress.

The K6 test script is designed to simulate a typical user journey through the "Online Boutique" application. This journey includes visiting the homepage, browsing products, adding items to the cart, proceeding to checkout, and returning to the homepage. These steps represent the core interactions a user would have on the “Online Boutique” site, making the test highly relevant for evaluating the performance of AKS and K3S in a production-like environment.

The test script is structured into three stages:

- **Ramp-up to 100 VUs over 30 seconds**: This stage evaluates the platform's ability to manage increasing loads quickly.
- **Ramp-up to 200 VUs for 30 seconds**: This stage evaluates the system's stability under a high, consistent load.
- **Ramp-down to 0 VUs over 30 seconds**: This stage examines how the platform manages decreasing traffic.

The script is structured as follows:

```
import http from 'k6/http';
import { check, group, sleep } from 'k6';
import { randomIntBetween } from 'https://jslib.k6.io/k6-
utils/1.2.0/index.js';

export let options = {
  stages: [
    { duration: '30s', target: 100 },
    { duration: '30s', target: 200 },
    { duration: '30s', target: 0 },
  ],
};

const BASE_URL = 'http://10.10.48.155/'; // replace with your Online
Boutique URL

export default function () {
  group('Visit Homepage', function () {
    let res = http.get(`${BASE_URL}/`);
    check(res, {
      'Homepage loaded successfully': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));
  });

  group('Browse Products', function () {
    let res = http.get(`${BASE_URL}/product/0PUK6V6EV0`);
    check(res, {
      'Product page loaded successfully': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));
    res = http.get(`${BASE_URL}/product/9SIQT8TOJO`);
    check(res, {
      'Another product page loaded successfully': (r) => r.status ===
200,
    });
    sleep(randomIntBetween(1, 3));
  });

  group('Add to Cart', function () {
    let res = http.post(`${BASE_URL}/cart`, JSON.stringify({
product_id: '0PUK6V6EV0', quantity: 1 }), {
      headers: { 'Content-Type': 'application/json' },
    });
    check(res, {
      'Product added to cart': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));

    res = http.post(`${BASE_URL}/cart`, JSON.stringify({ product_id:
'9SIQT8TOJO', quantity: 1 }), {
      headers: { 'Content-Type': 'application/json' },
    });
    check(res, {
      'Another product added to cart': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));
  });

  group('Proceed to Checkout', function () {
    let res = http.get(`${BASE_URL}/cart`);
    check(res, {
      'Cart page loaded successfully': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));

    res = http.post(`${BASE_URL}/cart`, JSON.stringify({
      email: 'user@example.com',
      street_address: '123 Main St',
      zip_code: '12345',
      city: 'Somewhere',
      state: 'CA',
      country: 'US',
      credit_card_number: '1234 5678 9012 3456',
      credit_card_expiration_month: '12',
      credit_card_expiration_year: '2030',
      credit_card_cvv: '123',
    }), {
      headers: { 'Content-Type': 'application/json' },
    });
    check(res, {
      'Checkout completed': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));
  });

  group('Return to Homepage', function () {
    let res = http.get(`${BASE_URL}/`);
    check(res, {
      'Homepage loaded after checkout': (r) => r.status === 200,
    });
    sleep(randomIntBetween(1, 3));
  });
}
```

Code 2. K6 test script for simulating virtual user load on AKS and K3S

To execute the test script provided in Code 2, use the following command:

```
k6 run load_test.js
```

A detailed guide on setting up and running the K6 tests is available on GitHub.

The K6 tests generated several key metrics that offer insights into the performance of AKS and K3S. These metrics include:

- **Group duration (avg):** The average time taken to complete the entire user journey, from visiting the homepage to checking out and returning to the homepage.
- **HTTP request duration (avg):** The average time taken to process individual HTTP requests during the user journey.
- **Total iterations:** The number of complete user journeys processed during the test.
- **Total HTTP requests (http_reqs):** The total number of HTTP requests made during the test.
- **Data received and data sent:** The total amount of data exchanged between the client and server during the test, which includes data retrieved from the server and data sent by the client, such as form submissions.
- **Iteration duration (avg):** The average time taken for one complete iteration of the user journey.

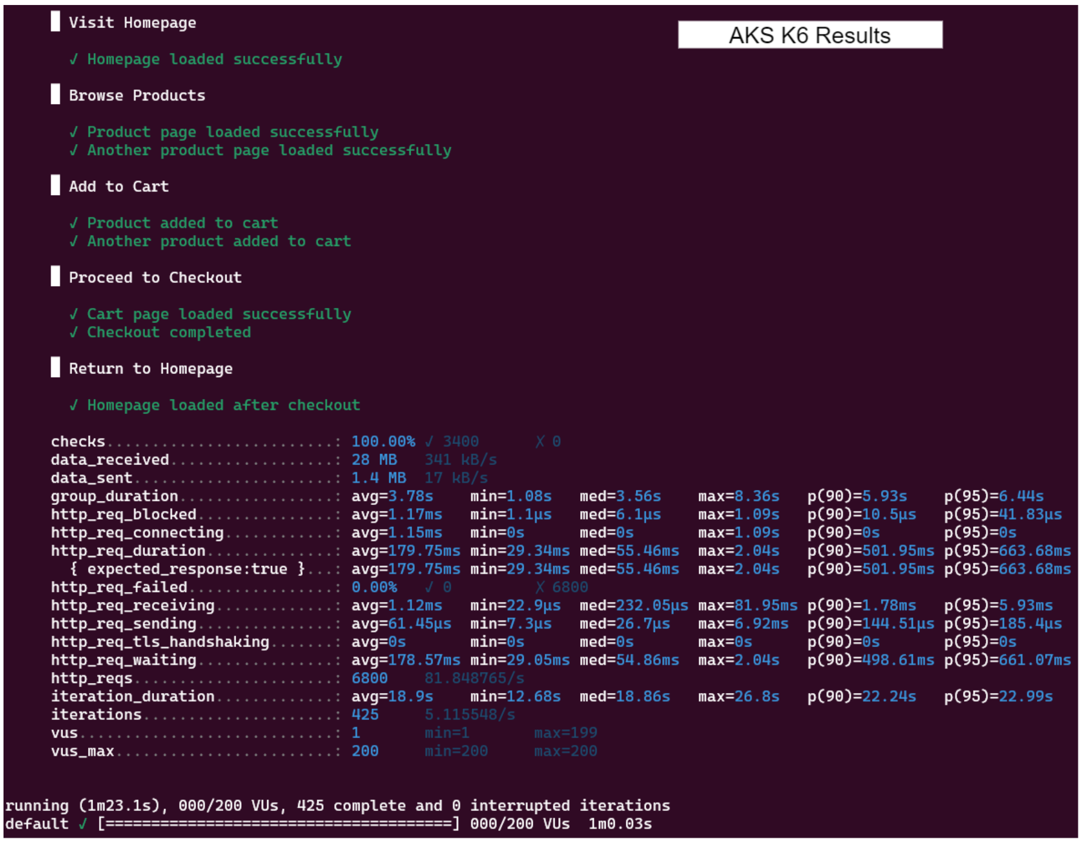
Table 15 summarizes the key performance metrics obtained from the K6 testing on AKS and K3S.

Table 15. K6 Benchmark Results for AKS and K3S

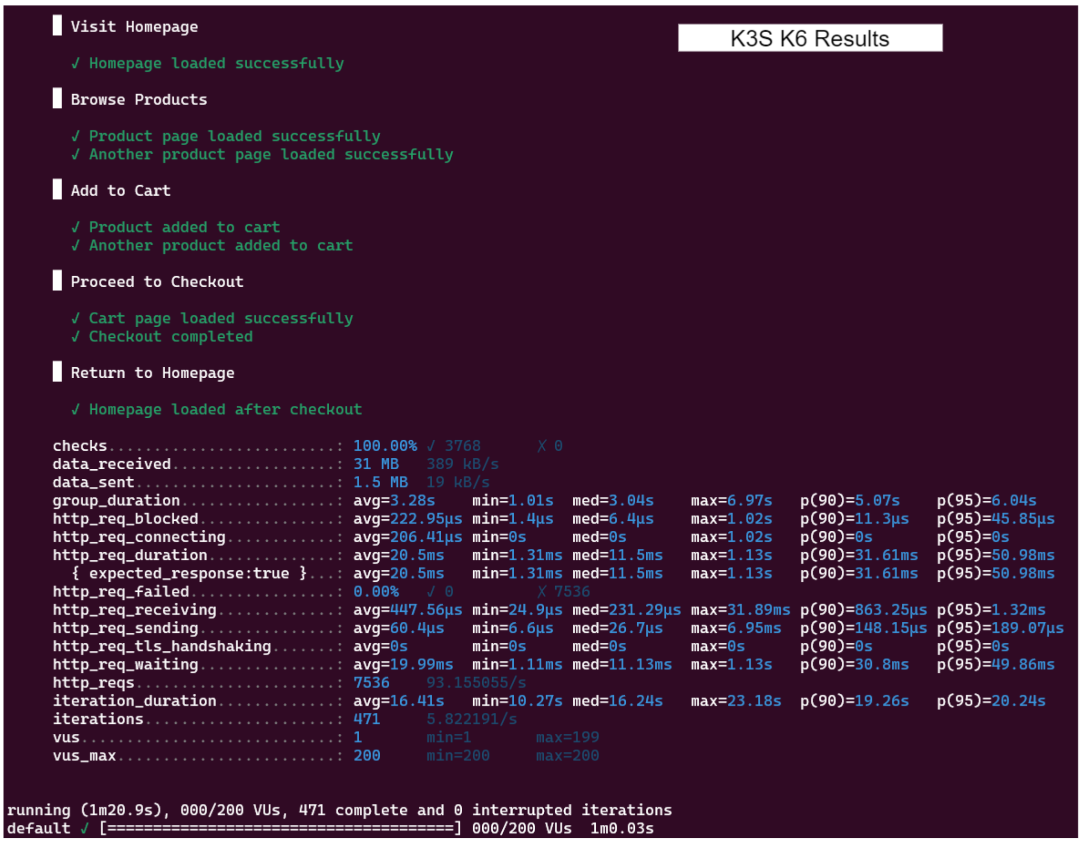
Metric	**AKS**	**K3S**
Group Duration (avg)	3.78 s	3.28 s
HTTP Request Duration (avg)	179.75 ms	20.5 ms
Total Iterations	425	471
Total Requests (http_reqs)	6800	7536
Data Received	28 MB	31 MB
Data Sent	1.4 MB	1.5 MB
Iteration Duration (avg)	18.9 s	16.41 s

The K6 performance tests reveal that K3S achieves faster response times and handles more requests compared to AKS when running the "Online Boutique" application. Specifically, K3S demonstrates a significantly lower average HTTP request duration (20.5 ms vs. 179.75 ms) and completes a higher number of iterations and total requests.

However, it is important to interpret these results within the broader context of the underlying infrastructure and operational environments, which is discussed in detail in Chapter 5.2.3, “System Benchmarking Results”. As shown in Figure 17, AKS operates within the Azure cloud ecosystem, which involves additional layers of management and cloud services that may impact raw performance metrics. Factors such as the hypervisor overhead, the specific CPU performance, and the cloud services running within the AKS cluster can contribute to the observed differences in performance.



```
█ Visit Homepage

  ✓ Homepage loaded successfully

█ Browse Products

  ✓ Product page loaded successfully
  ✓ Another product page loaded successfully

█ Add to Cart

  ✓ Product added to cart
  ✓ Another product added to cart

█ Proceed to Checkout

  ✓ Cart page loaded successfully
  ✓ Checkout completed

█ Return to Homepage

  ✓ Homepage loaded after checkout

checks.........................: 100.00% ✓ 3400      ✗ 0
data_received..................: 28 MB   341 kB/s
data_sent......................: 1.4 MB  17 kB/s
group_duration.................: avg=3.78s    min=1.08s    med=3.56s     max=8.36s    p(90)=5.93s    p(95)=6.44s
http_req_blocked...............: avg=1.17ms   min=1.1µs    med=6.1µs     max=1.09s    p(90)=10.5µs   p(95)=41.83µs
http_req_connecting............: avg=1.15ms   min=0s       med=0s        max=1.09s    p(90)=0s       p(95)=0s
http_req_duration..............: avg=179.75ms min=29.34ms  med=55.46ms   max=2.04s    p(90)=501.95ms p(95)=663.68ms
  { expected_response:true }...: avg=179.75ms min=29.34ms  med=55.46ms   max=2.04s    p(90)=501.95ms p(95)=663.68ms
http_req_failed................: 0.00%   ✓ 0         ✗ 6800
http_req_receiving.............: avg=1.12ms   min=22.9µs   med=232.05µs  max=81.95ms  p(90)=1.78ms   p(95)=5.93ms
http_req_sending...............: avg=61.45µs  min=7.3µs    med=26.7µs    max=6.92ms   p(90)=144.51µs p(95)=185.4µs
http_req_tls_handshaking.......: avg=0s       min=0s       med=0s        max=0s       p(90)=0s       p(95)=0s
http_req_waiting...............: avg=178.57ms min=29.05ms  med=54.86ms   max=2.04s    p(90)=498.61ms p(95)=661.07ms
http_reqs......................: 6800    81.848765/s
iteration_duration.............: avg=18.9s    min=12.68s   med=18.86s    max=26.8s    p(90)=22.24s   p(95)=22.99s
iterations.....................: 425     5.115548/s
vus............................: 1       min=1       max=199
vus_max........................: 200     min=200     max=200


running (1m23.1s), 000/200 VUs, 425 complete and 0 interrupted iterations
default ✓ [======================================] 000/200 VUs  1m0.03s
```



```
█ Visit Homepage

  ✓ Homepage loaded successfully

█ Browse Products

  ✓ Product page loaded successfully
  ✓ Another product page loaded successfully

█ Add to Cart

  ✓ Product added to cart
  ✓ Another product added to cart

█ Proceed to Checkout

  ✓ Cart page loaded successfully
  ✓ Checkout completed

█ Return to Homepage

  ✓ Homepage loaded after checkout

checks.........................: 100.00% ✓ 3768      ✗ 0
data_received..................: 31 MB   389 kB/s
data_sent......................: 1.5 MB  19 kB/s
group_duration.................: avg=3.28s    min=1.01s   med=3.04s    max=6.97s   p(90)=5.07s    p(95)=6.04s
http_req_blocked...............: avg=222.95µs min=1.4µs   med=6.4µs    max=1.02s   p(90)=11.3µs   p(95)=45.85µs
http_req_connecting............: avg=206.41µs min=0s      med=0s       max=1.02s   p(90)=0s       p(95)=0s
http_req_duration..............: avg=20.5ms   min=1.31ms  med=11.5ms   max=1.13s   p(90)=31.61ms  p(95)=50.98ms
  { expected_response:true }...: avg=20.5ms   min=1.31ms  med=11.5ms   max=1.13s   p(90)=31.61ms  p(95)=50.98ms
http_req_failed................: 0.00%   ✓ 0         ✗ 7536
http_req_receiving.............: avg=447.56µs min=24.9µs  med=231.29µs max=31.89ms p(90)=863.25µs p(95)=1.32ms
http_req_sending...............: avg=60.4µs   min=6.6µs   med=26.7µs   max=6.95ms  p(90)=148.15µs p(95)=189.07µs
http_req_tls_handshaking.......: avg=0s       min=0s      med=0s       max=0s      p(90)=0s       p(95)=0s
http_req_waiting...............: avg=19.99ms  min=1.11ms  med=11.13ms  max=1.13s   p(90)=30.8ms   p(95)=49.86ms
http_reqs......................: 7536    93.155055/s
iteration_duration.............: avg=16.41s   min=10.27s  med=16.24s   max=23.18s  p(90)=19.26s   p(95)=20.24s
iterations.....................: 471     5.822191/s
vus............................: 1       min=1       max=199
vus_max........................: 200     min=200     max=200


running (1m20.9s), 000/200 VUs, 471 complete and 0 interrupted iterations
default ✓ [======================================] 000/200 VUs  1m0.03s
```

Figure 17. K6 Benchmark Results for AKS and K3S

The diagram on Figure 18 visually represents the K6 benchmark results, illustrating the differences in performance between the two platforms.

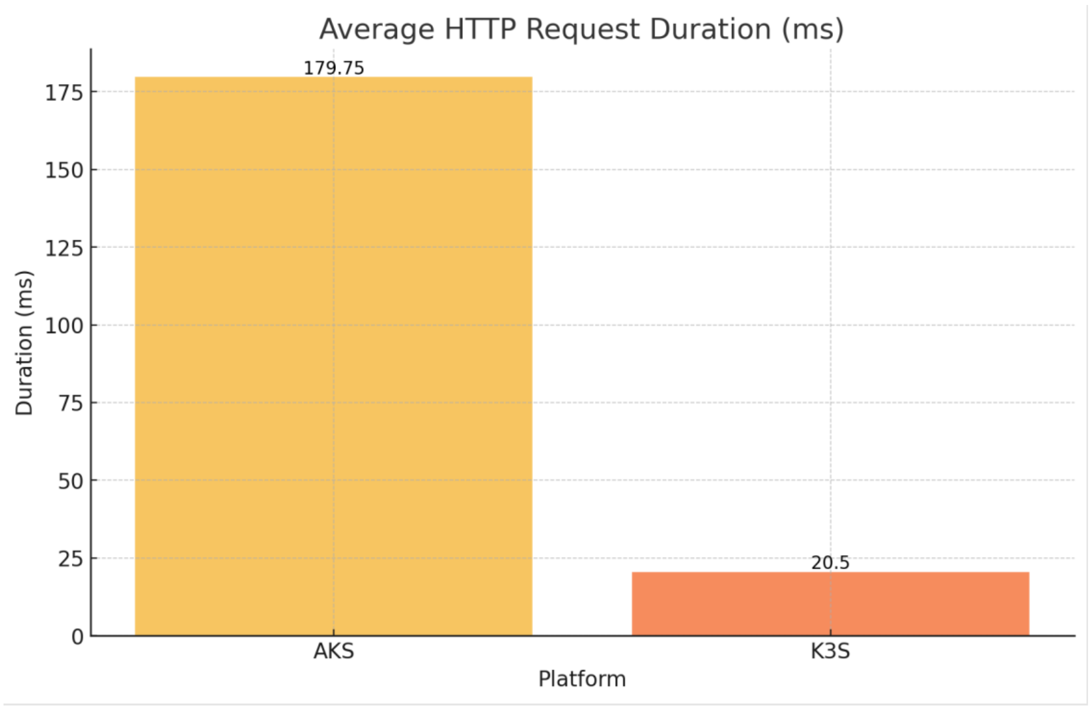


Figure 18. Diagram of K6 Benchmark Results for AKS and K3S

System load analysis of K6 benchmark:

To further understand the impact of the K6 testing on system resources, Grafana dashboards were used to monitor the load on both worker nodes in each cluster. Grafana, utilizing Prometheus and Node Exporter, provided real-time insights into CPU, memory, and network usage during the tests. The results, as shown in Figure 19 and Figure 20, indicate a significant impact on CPU usage in both AKS and K3S clusters during the tests. However, there was no significant impact on RAM or disk usage. Both clusters experienced a considerable increase in network traffic, similar to the results observed in the AB testing.

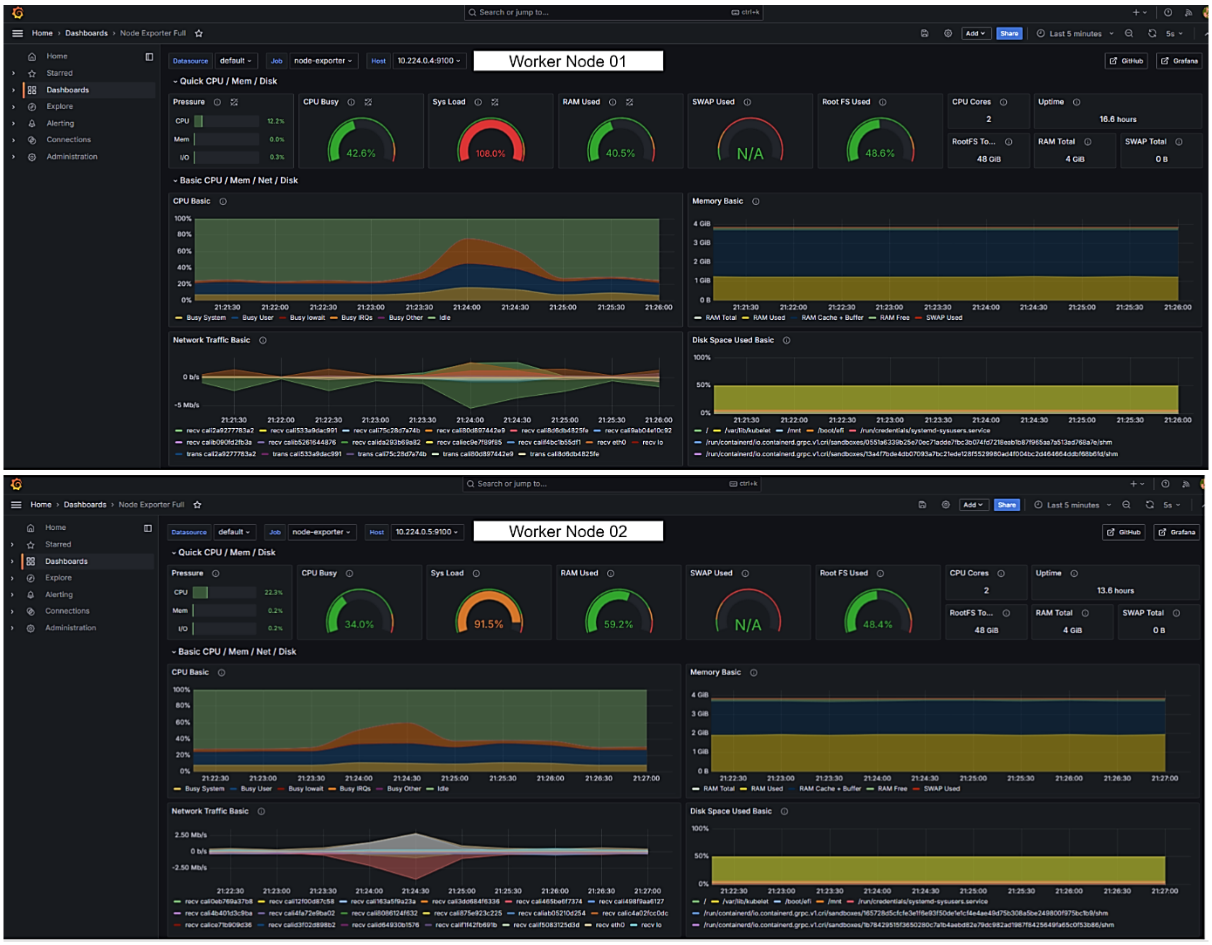


Figure 19. System Load on AKS Worker Nodes During K6 Testing

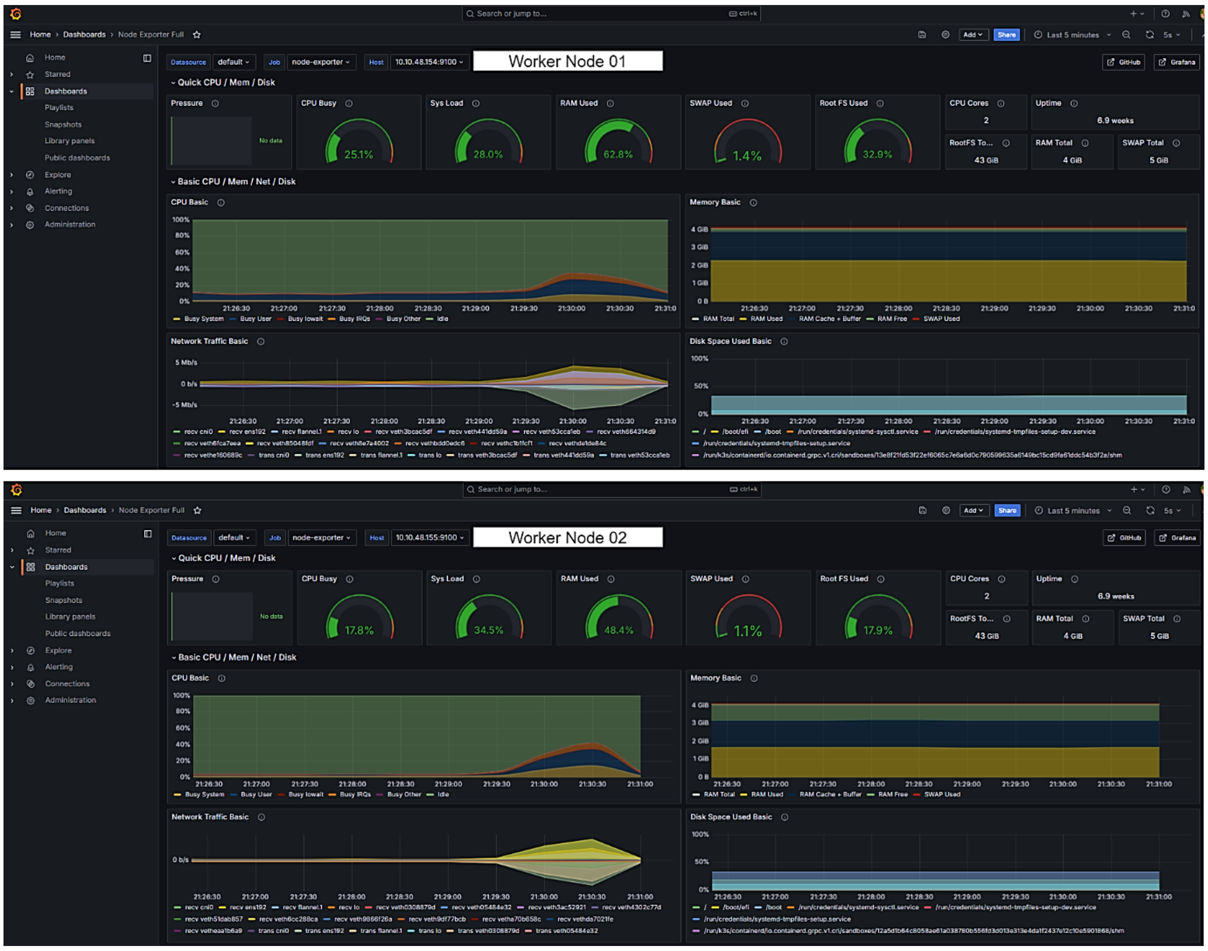


Figure 20. System Load on K3S Worker Nodes During K6 Testing

5.2.3. System Benchmarking Results

The benchmarking is conducted using Sysbench, a multi-threaded benchmark tool that evaluates system parameters such as CPU speed, memory transfer rate, and file I/O throughput. For consistency, the tests are performed on one worker node from each cluster, as both nodes in each cluster are configured identically.

Evaluate parameters and metrics:

1. **CPU Test**: Measures the number of prime numbers generated per second, providing an indication of CPU efficiency.
2. **Memory Test**: Evaluates the memory transfer speed by performing read operations with a specified block size.
3. **File I/O Test**: Assesses the read/write throughput and file synchronization (fsync) rates, which are crucial for applications with intensive disk operations.

Table 16 presents the results of the system benchmarking for AKS and K3S:

Table 16. System Benchmark Results Comparison for AKS and K3S

Metric	**AKS**	**K3S**
CPU Speed (events/s)	8050.65	20219.14
Memory Transfer Rate (MiB/s)	451.7	1339.27
Read Throughput (MiB/s)	6.78	36.19
Write Throughput (MiB/s)	4.52	24.13
Reads/s	433.73	2316.35
Writes/s	289.16	1544.23
Fsyncs/s	925.69	4942.25

The results indicate that K3S significantly outperforms AKS in all measured metrics. The CPU speed of K3S is more than twice that of AKS, highlighting its superior computational efficiency. Similarly, K3S demonstrates higher memory transfer rates and better I/O throughput, suggesting that it can handle more demanding workloads with greater resource efficiency.

The diagram on Figure 21 is visually represents the system benchmark results for AKS and K3S, highlighting the differences in performance across key metrics

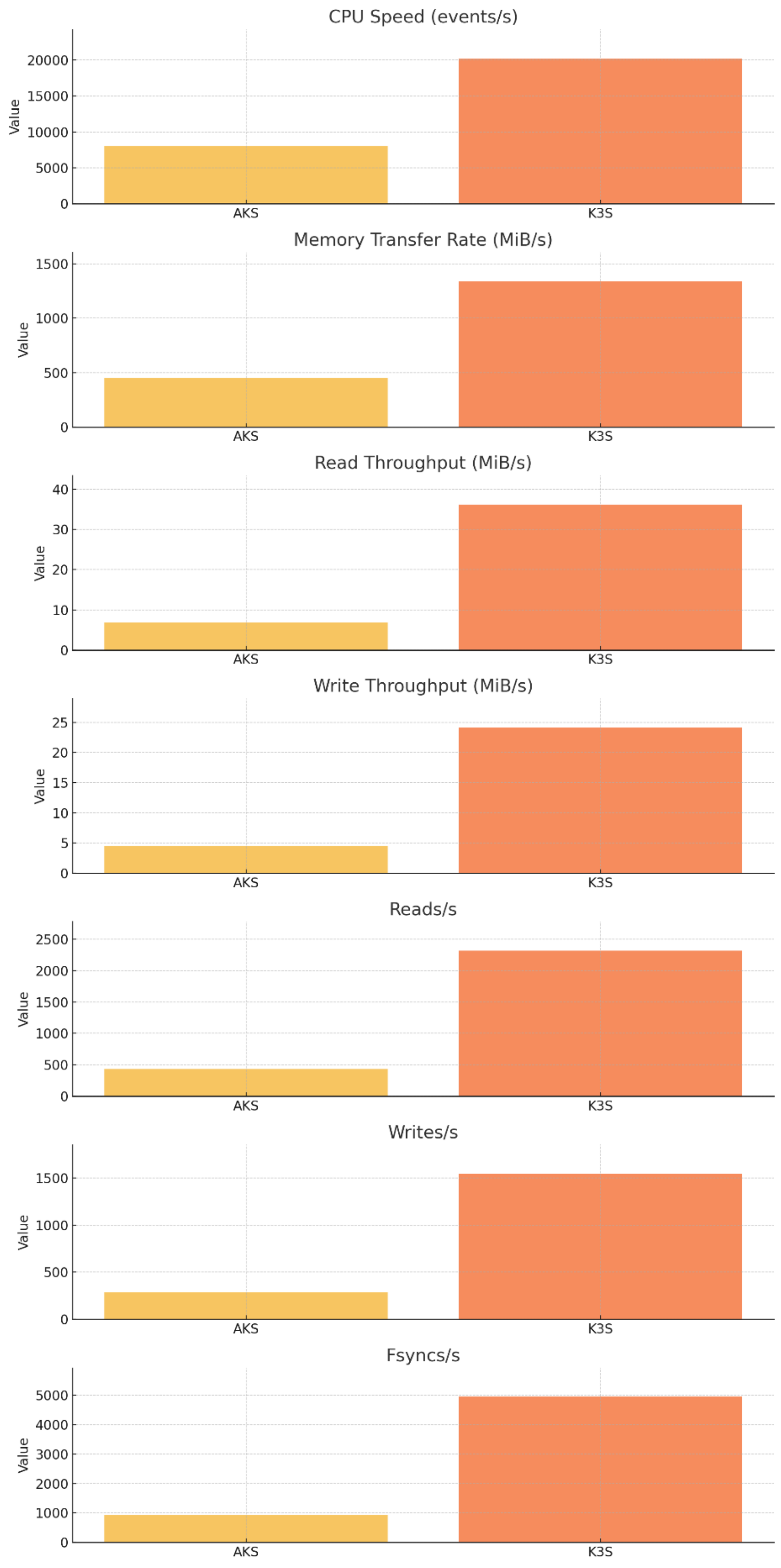


Figure 21. System Benchmark Results for AKS and K3S

AKS worker node sysbench output summary:

The performance results for the AKS and K3S worker nodes during the Sysbench tests are summarized in Table 17 and Table 18.

Table 17. AKS worker node sysbench output summary

CPU test	**Memory test**	**File I/O test**
CPU speed: 8,050.65 events per second **Latency (ms):** avg: 0.12, max: 29.49, 95th percentile: 0.08	**Total operations:** 3,072 (451.70 per second) **Transfer rate:** 451.70 MiB/sec **Latency (ms):** avg: 2.21, max: 37.02, 95th percentile: 6.67	**File operations: reads/s:** 433.73, writes/s: 289.16, fsyncs/s: 925.69 **Throughput: read**, MiB/s: 6.78, written, MiB/s: 4.52 **Latency (ms):** avg: 0.60, max: 180.51, 95th percentile: 2.66

Table 18. K3S worker node sysbench summary

CPU test	**Memory test**	**File I/O test**
CPU Speed: 20,219.14 events per second **Latency (ms):** avg: 0.05, max: 2.03, 95th percentile: 0.06	**Total Operations:** 3,072 (1,339.27 per second) **Transfer Rate:** 1,339.27 MiB/sec **Latency (ms):** avg: 0.74, max: 1.69, 95th percentile: 1.04	**File Operations:** reads/s: 2,316.35, writes/s: 1,544.23, fsyncs/s: 4,942.25 **Throughput:** read: 36.19 MiB/s, written: 24.13 MiB/s **Latency (ms):** avg: 0.11, max: 7.32, 95th percentile: 0.65

Analysis of lscpu output:

The *lscpu* command provides details about the CPU architecture and configuration for worker nodes on AKS and K3S. Below is a comparison and analysis based on the *lscpu* outputs, as shown in Table 19.

Table 19. Highlights of *lscpu* output

AKS worker node *lscpu* output	**K3S worker node *lscpu* output**
`Architecture:             x86_64`	`Architecture:             x86_64`
`CPU(s):                   2`	`CPU(s):                   2`
`Model name:                Intel(R) Xeon(R) CPU E5-2673 v4 @ 2.30GHz`	`Model name:                Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz`
`Thread(s) per core:       1`	`Thread(s) per core:       1`
`Core(s) per socket:       2`	`Core(s) per socket:       1`
`Socket(s):                1`	`Socket(s):                2`
`BogoMIPS:                 4589.37`	`BogoMIPS:                 4389.68`
`Hypervisor vendor:       Microsoft`	`Hypervisor vendor:        VMware`
`L1d cache:                32 KiB`	`L1d cache:                32 KiB`
`L1i cache:                32 KiB`	`L1i cache:                32 KiB`
`L2 cache:                 256 KiB`	`L2 cache:                 1 MiB`
`L3 cache:                 50 MiB`	`L3 cache:                 27.5 MiB`

Explanation of Results

The K3S configuration shows better performance in system benchmarks compared to AKS, despite both having a similar number of cores and threads. Several factors contribute to this performance difference:

1. **Cache Hierarchy**: K3S has a larger L2 cache per instance (1 MiB) compared to AKS (256 KiB), which can significantly enhance processing speed for memory-intensive tasks.
2. **CPU Architecture**: Although both systems use Xeon processors, the architecture and generation differences might lead to variations in handling workloads. The Xeon Silver 4114 on K3S, with a different cache architecture and newer optimizations, shows improved throughput and efficiency.

3. **Hypervisor Overhead**: The hypervisor technology used (Microsoft for AKS and VMware for K3S) might also affect how efficiently CPU resources are allocated and managed, impacting the overall performance.
4. **Virtualization and Execution Environment**: The difference in virtualization platforms (Microsoft vs. VMware) may lead to variations in CPU performance due to different management of resources and overheads.
5. **Memory Transfer Rates**: The higher memory transfer rate observed in K3S indicates better memory handling, due to the configuration and optimizations in the virtualization stack or hardware capabilities.

Clarification on experimental constraints:

It is important to clarify that due to limitations in accessing identical hardware configurations in the Azure cloud, it was not possible to set up an AKS environment with the same or equivalent CPU and memory specifications as the on-premises K3S setup. As a result, while the hardware configurations chosen were the closest possible matches, they were not identical. This discrepancy introduces a variable that complicates direct comparison between the two platforms. Consequently, the superior performance observed in K3S is likely influenced by these hardware differences rather than a fundamental advantage of K3S as an orchestration tool.

Therefore, while the results from the benchmarking tests provide valuable insights into the performance of AKS and K3S under specific conditions, they should not be interpreted as definitive evidence of the superiority of one orchestration tool over the other.

5.3. Cost Analysis

This section provides a detailed cost analysis of deploying the "Online Boutique" microservices application on Azure Kubernetes Service (AKS) versus K3S. The analysis is grounded in the methodology provided by the Azure Total Cost of Ownership (TCO) Calculator. This tool allows for a comprehensive comparison of costs between on-premises infrastructure and cloud deployments on Azure, using industry-standard pricing assumptions. The goal is to outline the methodology used to estimate the total cost of ownership for both platforms and to present a comparative cost analysis based on these estimates.

5.3.1. Cost Calculation Methodology

This thesis utilizes the Azure Total Cost of Ownership (TCO) Calculator, an official online tool provided by Microsoft Azure [35]. This tool is designed to help businesses estimate the cost savings that can be achieved by migrating workloads to Azure. The TCO Calculator compares the costs associated with running on-premises infrastructure against deploying in Azure, considering a wide range of factors such as hardware, software, operational, and maintenance expenses.

U.S. average pricing assumptions:

The analysis uses U.S. average pricing assumptions provided by the TCO Calculator, which are based on industry-standard values recognized by Nucleus Research [36]. These assumptions are crucial for creating a reliable and consistent cost comparison between AKS and K3S. Below are the specific assumptions used.

- **Software Assurance Coverage (Azure Hybrid Benefit)**: This analysis includes the Azure Hybrid Benefit, which allows organizations to apply their existing Windows and SQL Server licenses with Software Assurance to Azure virtual machines. This benefit can lead to significant cost savings by reducing the need to purchase new cloud licenses.
- **Geo-Redundant Storage (GRS)**: This assumption covers the cost of replicating data to a secondary region, enhancing disaster recovery capabilities but increasing storage expenses. This feature enhances data availability and disaster recovery capabilities, though it adds to storage costs.
- **Virtual Machine Costs**: The calculator allows the exclusion of cost recommendations for lower-cost, less performing virtual machines, ensuring that the analysis focuses on more relevant and widely used VM types.
- **Electricity Costs**: Electricity costs are estimated based on the average U.S. price per kilowatt-hour, reflecting the power consumption necessary for running servers and maintaining data center operations.
- **Storage Costs**: Storage costs are detailed for various types of storage, including SAN, NAS, and Blob storage. Additionally, an annual enterprise storage software support cost is included, reflecting ongoing maintenance and support expenses.
- **IT Labor Costs**: The model assumes that one IT administrator can manage a specific number of physical servers or virtual machines, with an average hourly rate reflecting the cost of IT staff required to maintain and manage infrastructure.

- **Hardware Costs**: Detailed hardware costs are included, covering various configurations of processors, cores, and RAM. These costs reflect the price of acquiring, maintaining, and replacing physical hardware over time.
- **Networking Costs**: Network-related costs are calculated as a percentage of the overall hardware and software costs, including expenses for network hardware, maintenance, and data transfer.
- **Data Center Costs**: The analysis includes the costs associated with constructing, maintaining, and operating data centers. This encompasses initial setup costs, ongoing operational expenses, and infrastructure maintenance.
- **Virtualization Costs**: The cost of virtualization is considered, including the per virtual machine, per month, infrastructure cost for load balancing, backup, and patching.
- **Database Costs**: The TCO model also accounts for database-related expenses, including licensing for SQL Server and other database management systems.
- **Data Warehouse Costs**: The costs associated with data warehouses are also considered, including the price per 2-core pack for Windows Server Standard, the number of devices per server (CAL), and related licensing and infrastructure costs. These costs reflect the financial implications of deploying and maintaining data warehouse solutions.

Calculation approach:

To ensure an accurate comparison, the TCO Calculator estimates the total cost of ownership for deploying the 'Online Boutique' application, both on-premises and in Azure. The process involves the following steps:

1. **Define Workload:** The current on-premises infrastructure for running the "Online Boutique" is detailed, including the number of servers, storage requirements, and network components.
2. **Customize Cost Inputs:** Cost details for the on-premises setup, such as hardware purchase, maintenance, power, and cooling, are entered into the calculator.
3. **Review Azure Costs:** The TCO Calculator estimates the equivalent costs on Azure, considering virtual machines, storage, networking, and additional services like Azure Kubernetes Service (AKS).

4. **Analyse Results:** The tool provides a detailed comparison, highlighting potential cost savings or increases over a specified period. This analysis helps in understanding the financial implications of migrating to Azure or maintaining an on-premises K3S setup.

Objective of the methodology:

The primary objective of using the TCO Calculator in this analysis is to establish a standardized methodology that can be applied to any market, application, or orchestration tool. While the specific cost figures are based on U.S. averages, the methodology itself is universal, allowing for its application in various contexts with appropriate adjustments.

5.3.2. Comparative Cost Analysis

This section outlines the comparative cost analysis between Azure Kubernetes Service (AKS) and K3S using the Total Cost of Ownership (TCO) Calculator provided by Microsoft Azure. The analysis is based on the inputs and assumptions defined in the TCO Calculator.

Defining workloads

The first step in the TCO Calculator is to define the workloads that will be used in the analysis. For this comparative study, the workloads represent the on-premises infrastructure that would be required to run the K3S cluster:

1. **Servers:**
    a. **Workload Type**: Windows/Linux Server
    b. **Environment**: Virtual Machines
    c. **Operating System**: Linux
    d. **VMs**: 5 (three control plane nodes and two worker nodes for the K3S cluster)
    e. **Virtualization**: VMware
    f. **Core(s)**: 2 per VM
    g. **RAM (GB)**: 4 per VM, optimized by CPU
2. **Storage:**
    a. The on-premises infrastructure uses a total of 250 GB SSD storage, divided across the five VMs.
    b. **Backup**: 250 GB for backup storage to ensure data integrity and recovery.
    c. **Archive**: 0 GB allocated for archiving, as this scenario does not include long-term storage.
    d. **IOPS**: Input/Output Operations Per Second set to 0, reflecting a standard configuration without high-demand storage performance requirements.

3. **Networking:**
    a. Outbound bandwidth is set to 50 GB with the destination region specified as West Europe. This bandwidth allocation reflects the typical network traffic generated by the K3S cluster during normal operations.

Adjusting assumptions

After defining the workloads, the TCO Calculator allows for adjustments in various assumptions to better reflect the real-world conditions under which these platforms operate.

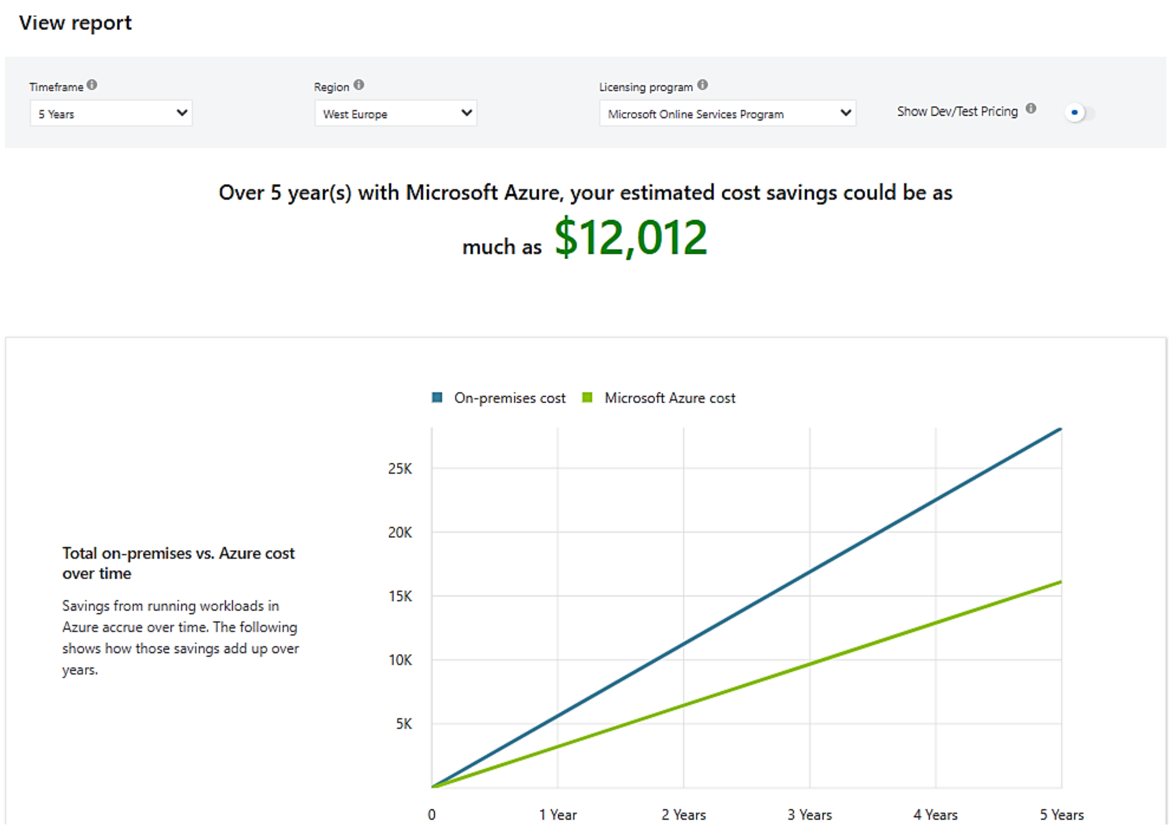
1. **Software Assurance Coverage:**
    a. **Disabled**: Neither Windows Server nor SQL Server Software Assurance is enabled in this scenario, meaning Azure Hybrid Benefit (AHB) is not applied. This choice results in a more generalized cost estimate without the potential savings provided by AHB.
2. **Geo-Redundant Storage (GRS):**
    a. **Disabled**: Data replication to a secondary region (GRS) is not enabled, which lowers storage costs but also reduces data redundancy.
3. **Virtual Machine Costs:**
    a. The option to recommend cost-optimized virtual machines, such as the Bs-series, is **disabled**. This setting ensures that the cost estimates are not influenced by specific VM recommendations, allowing for a broader and more applicable comparison.
4. **Electricity Costs:**
    a. Set at \$0.1334 per kWh, reflecting the average electricity cost in the U.S. This is an important factor in the total cost of ownership for on-premises infrastructure, as energy consumption directly impacts operational expenses.
5. **Storage Costs:**
    a. **Procurement Costs:**
    b. SAN-SSD: \$0.4 per GB
    c. SAN-HDD: \$0.2 per GB
    d. NAS: \$0.2 per GB
    e. Blob Storage: \$0.2 per GB
    f. **Annual Enterprise Storage Software Support Cost:** 10% of the storage procurement costs.

    g. **Cost per Tape Drive:** $160, though this is not directly applied in the current scenario.
6. **IT Labor Costs:**
    a. **Physical Servers per Administrator:** 100
    b. **Virtual Machines per Administrator:** 120
    c. **Hourly Rate for IT Administrator:** $23, reflecting the average cost of IT labor in the U.S.
7. **Other Assumptions:**
    a. **Default Values:** For other assumptions in the TCO model, including hardware costs, software costs, virtualization costs, data center costs, networking costs, database costs, and data warehouse costs, the default values provided by the TCO Calculator are used. These default values are industry-standard and provide a consistent basis for comparison.

Report and results analysis

After defining workloads and adjusting assumptions, the TCO Calculator generates a detailed report comparing the five-year total cost of ownership for on-premises K3S infrastructure versus on Azure.

The report indicates that over five years, migrating from an on-premises K3S setup to Azure could result in savings of approximately $**12,012.00**. This is primarily due to Azure's efficient infrastructure management, reduced data center costs, and lower IT labor requirements, as illustrated in Figure 22.


View report

Timeframe: 5 Years
Region: West Europe
Licensing program: Microsoft Online Services Program
Show Dev/Test Pricing

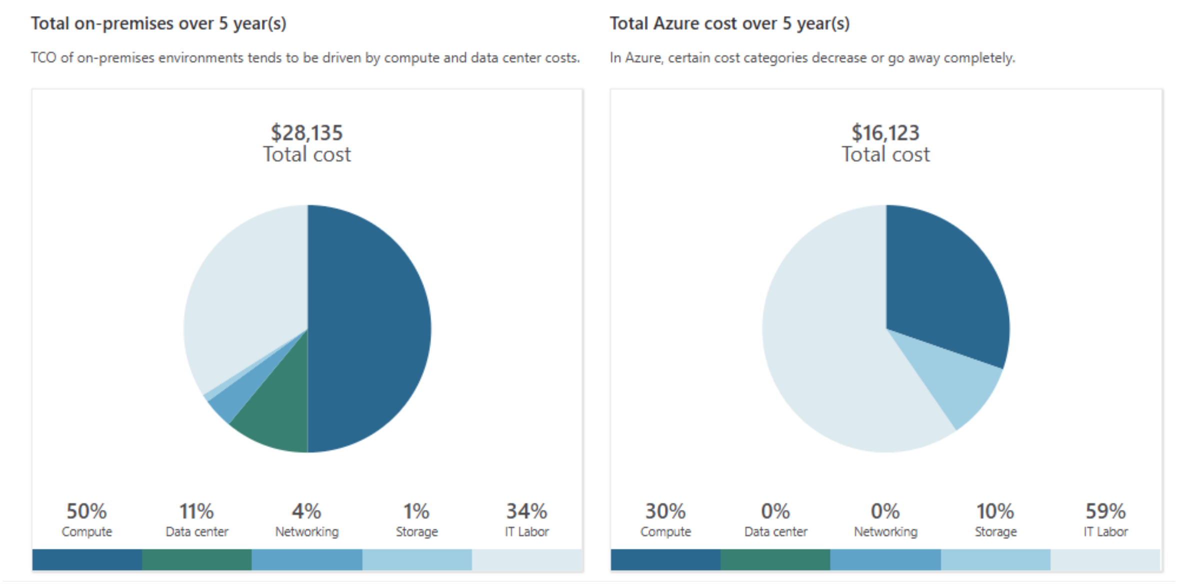
Over 5 year(s) with Microsoft Azure, your estimated cost savings could be as much as $12,012

Total on-premises vs. Azure cost over time

Savings from running workloads in Azure accrue over time. The following shows how those savings add up over years.

Figure 22. Total Cost of Ownership Over 5 Years: On-Premises vs. Azure.

A breakdown of costs shows that on-premises expenses are primarily driven by compute (50%), data center (11%), and IT labor (34%). In contrast, Azure's cost distribution shifts significantly, with compute at 30% and IT labor at 59%, while data center and networking costs are effectively eliminated due to Azure's managed services. Total costs are shown in Figure 23.


Total on-premises over 5 year(s)

TCO of on-premises environments tends to be driven by compute and data center costs.

$28,135 Total cost

50%	11%	4%	1%	34%
Compute	Data center	Networking	Storage	IT Labor

Total Azure cost over 5 year(s)

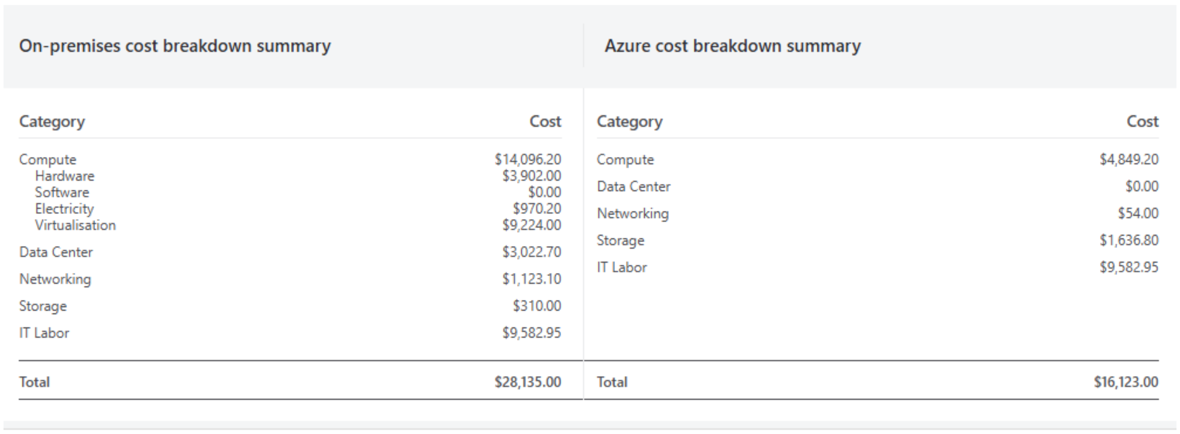
In Azure, certain cost categories decrease or go away completely.

$16,123 Total cost

30%	0%	0%	10%	59%
Compute	Data center	Networking	Storage	IT Labor

Figure 23. Cost Distribution of On-Premises and Azure Infrastructure Over 5 Years.

A detailed breakdown highlights that the on-premises setup incurs higher expenses due to the need for physical infrastructure and significant IT labor costs are reported in Figure 24 Azure's cost efficiency stems from economies of scale and the reduction of direct management responsibilities through managed services.



On-premises cost breakdown summary

Category	Cost
Compute	$14,096.20
Hardware	$3,902.00
Software	$0.00
Electricity	$970.20
Virtualisation	$9,224.00
Data Center	$3,022.70
Networking	$1,123.10
Storage	$310.00
IT Labor	$9,582.95
Total	$28,135.00

Azure cost breakdown summary

Category	Cost
Compute	$4,849.20
Data Center	$0.00
Networking	$54.00
Storage	$1,636.80
IT Labor	$9,582.95
Total	$16,123.00

Figure 24. Detailed Cost Breakdown for On-Premises and Azure Over 5 Years

This analysis looks at the direct migration of five virtual machines (VMs) from an on-premises environment to Azure, where each VM, including its compute, storage, and networking resources, is replicated in the cloud to mirror the original setup.

However, using Azure's managed Kubernetes service (AKS) would significantly reduce costs. In AKS, the control plane is fully managed by Azure at no extra charge, so the main expenses come from the compute and storage needs of the worker nodes. This is different from the direct VM migration approach, where all VMs, including those for the control plane, need to be fully provisioned and billed.

Leveraging AKS not only reduces management overhead but also takes advantage of Azure’s resource optimization for Kubernetes workloads. This approach is especially beneficial for long-term operational costs and scalability, where AKS offers greater efficiency compared to traditional VM-based deployments.

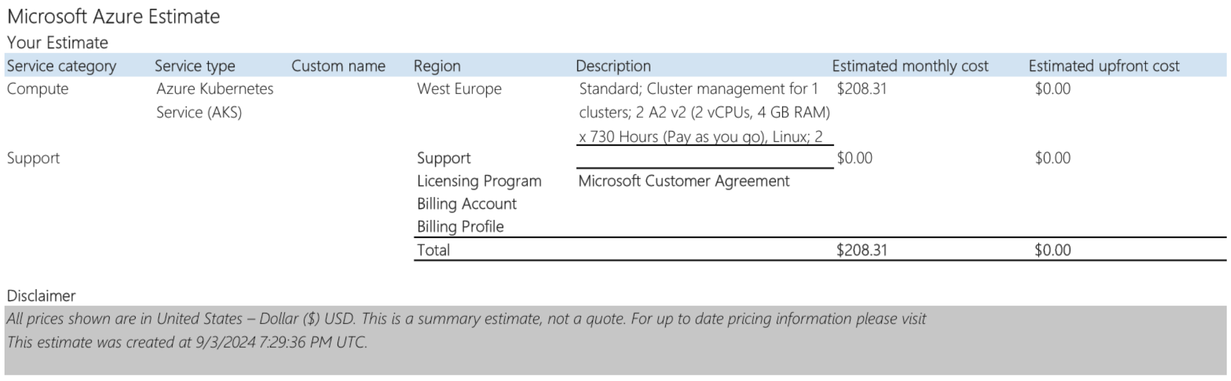
Azure AKS Pricing Breakdown

To further illustrate the cost-effectiveness of Azure, an additional pricing analysis was performed using the Azure Pricing Calculator for an AKS setup [37]. The AKS setup detailed below is the exact configuration used throughout this thesis to conduct performance testing, cost analysis, and other comparative evaluations between Azure-based Kubernetes services and on-premises deployments.

The AKS cluster was configured with the following parameters:

- Region: West Europe
- Cluster Management: Standard cluster management for 1 cluster, which is fully managed by Azure.
- Nodes:
    - Operating System: Linux
    - Instance Series: Standard A2 v2 (2 vCPUs, 4GB RAM)
    - VMs: 2 nodes, each priced at $0.086/hour for a total of 730 hours/month.
    - OS Disks: Standard SSD (64GB) per node, priced at $4.80/month per disk.

The estimated monthly cost for this AKS setup is **$208.31**, which translates to **$12,498.60** over five years, as shown in Figure 25.



Microsoft Azure Estimate

Your Estimate

Service category	Service type	Custom name	Region	Description	Estimated monthly cost	Estimated upfront cost
Compute	Azure Kubernetes Service (AKS)		West Europe	Standard; Cluster management for 1 clusters; 2 A2 v2 (2 vCPUs, 4 GB RAM) x 730 Hours (Pay as you go), Linux; 2	$208.31	$0.00
Support			Support		$0.00	$0.00
			Licensing Program	Microsoft Customer Agreement		
			Billing Account			
			Billing Profile			
			Total		$208.31	$0.00

Disclaimer

All prices shown are in United States – Dollar ($) USD. This is a summary estimate, not a quote. For up to date pricing information please visit

This estimate was created at 9/3/2024 7:29:36 PM UTC.

Figure 25. Estimated Monthly Cost for Azure AKS Setup.

In comparison, the cost of running the same five VMs directly in Azure for five years is 29% higher, totaling **$16,123** as per the TCO report generated for the same workloads, as shown in Figure 26.

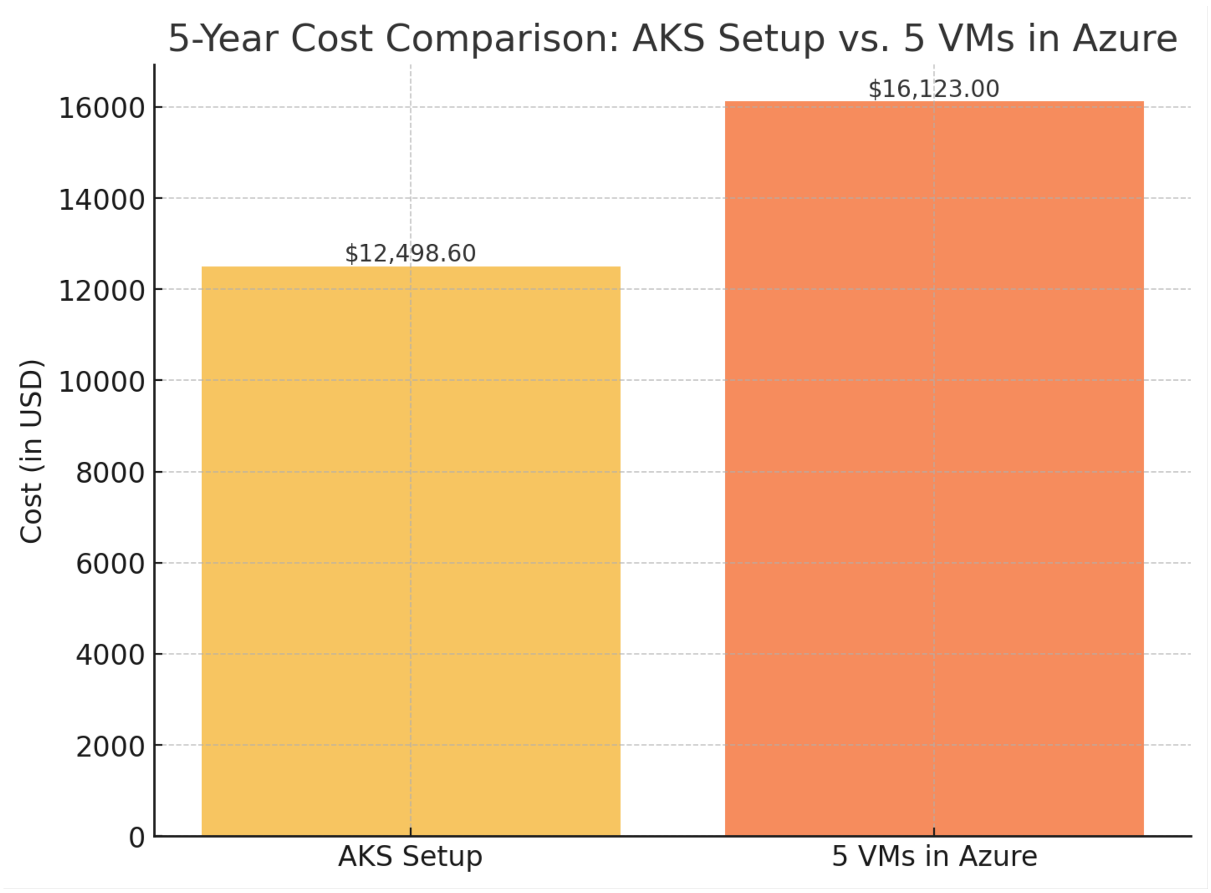


Figure 26. 5-Year Cost Comparison: AKS Setup vs. 5 VMs in Azure

Cost savings strategies in Azure:

Azure provides several options for reducing costs, making it an even more cost-effective platform for running Kubernetes workloads.

- **Pay-As-You-Go:** The standard pay-as-you-go pricing for 2 VMs in an AKS cluster is **$125.71** per month. This pricing model offers flexibility with no upfront commitment, making it suitable for workloads with unpredictable usage patterns.
- **Savings Plans**: For example, a three-year savings plan for 2 VMs in an AKS cluster can reduce the monthly cost to **$67.88**, representing a **46%** discount. The estimated monthly cost of this package, including cluster management and Managed OS Disks, is $**149.92**. Over a five-year period, this results in a total cost of **$8,995.20.**
- **Reservations**: For example, a three-year reservation for 2 VMs in an AKS cluster can reduce the monthly cost to **$47.77**, representing a **62%** discount. The estimated monthly cost of this package, including cluster management and Managed OS Disks, is **$130.37**. Over five years, this translates to a total cost of **$7,822.20**.

Cost Savings Strategies for K3S

When deploying infrastructure on-premises, several strategies can be employed to optimize costs and ensure efficient use of resources. These strategies are crucial for managing expenses while maintaining the necessary performance and reliability.

- **Virtualization with Proxmox**: Implementing virtualization platforms like Proxmox [38] can enhance resource utilization and reduce hardware costs. Proxmox is an open-source platform that allows you to create and manage virtual machines efficiently, providing high availability, disaster recovery, and backup functionalities without the licensing fees associated with commercial hypervisors. This enables better consolidation of workloads onto fewer physical machines, thereby lowering power, cooling, and maintenance costs.
- **Energy-efficient hardware**: Investing in energy-efficient servers and networking equipment can lead to substantial savings on power and cooling costs. Modern hardware often comes with features that optimize energy consumption, such as dynamic power scaling and efficient cooling systems. By selecting hardware with a favourable power-to-performance ratio, one can achieve significant long-term operational savings.
- **Open-source management tools**: Utilizing open-source tools for system management, monitoring, and automation is another effective cost-saving strategy. Solutions like Ansible for configuration management, Prometheus for monitoring, and Grafana for visualization provide robust functionality without the licensing costs associated with commercial alternatives. These tools help maintain visibility and control over your infrastructure while keeping software costs to a minimum.

5.3.3. Insights on On-Premises vs Cloud-Based Solutions

A critical insight from this thesis is the potential benefits of using K3S in on-premises environments, particularly when an organization already has a significant investment in physical infrastructure. While the cost analysis in this study was based on a setup with only five VMs, it is important to recognize that as the scale of on-premises infrastructure increases, the relative cost-effectiveness of K3S improves significantly. If an organization has high server utilization, running K3S on-premises could lead to substantial savings compared to cloud-based solutions like AKS. This is because the incremental cost of adding more VMs or utilizing more of the existing infrastructure is generally lower than the equivalent cost in a cloud environment, where pricing is based on usage.

For organizations with extensive on-premises resources, K3S does not only provide superior performance but also offers a way to maximize the value of existing hardware investments. This could make K3S a more economically viable option for large-scale or high-utilization scenarios, where the fixed costs of maintaining an on-premises data center are spread across a larger number of workloads.

6. Qualitative Analysis

This chapter presents a qualitative analysis of Azure Kubernetes Service (AKS) and K3S, focusing on their implementation, configuration, deployment ease, management, security features and tools for monitoring. Additionally, the chapter will explore community support and the quality of documentation available for each platform. Each section will critically examine the differences between these platforms, offering insights into their suitability for different environments and use cases.

6.1. Implementation and Configuration Differences

The implementation and configuration of AKS and K3S reflect their distinct design philosophies and target environments.

Initial Setup

The setup process for AKS is designed to be user-friendly and streamlined, especially given its managed nature within the Azure ecosystem. One of the significant advantages of AKS is the ability to deploy it through Azure GUI, which makes the process even simpler and more intuitive for users. This graphical interface reduces the need for deep technical knowledge, allowing for rapid deployment with minimal manual intervention. Additionally, the use of Terraform scripts, which have been written and used in this master's thesis, automates much of the configuration, including resource provisioning, networking setup, and cluster identity management. This automation further simplifies the initial setup, making AKS a compelling choice for enterprises looking to deploy a robust Kubernetes cluster quickly and efficiently.

In contrast, setting up K3S requires a more hands-on and technically involved approach. The process includes manually configuring virtual machines, setting up networking, and deploying the K3S binaries using tools like Ansible, with playbooks also developed and used in this thesis. Unlike AKS, where deployment can be done via a GUI, K3S setup is typically performed through the Linux terminal, adding a layer of complexity. While this approach offers greater flexibility, particularly in resource-constrained environments such as edge computing, it necessitates a deeper understanding of Kubernetes and the underlying infrastructure. K3S's lightweight nature allows it to be deployed in environments where AKS might be too resource-intensive, but this comes with the requirement for more technical expertise and manual configuration.

Customization

AKS offers a range of built-in features that are pre-configured, such as integrated identity management, network policies enforced through Calico, and seamless integration with Azure services like Azure Active Directory and Azure Monitor. These pre-configured services enhance security, simplify management, and provide a consistent operational experience. However, they can also limit the extent of customization available to users. For instance, modifying default network configurations, integrating non-Azure services, or deploying specialized software may require additional configuration steps or might be constrained by Azure's proprietary ecosystem. Additionally, while AKS provides a streamlined experience, it also ties users to the Azure platform, making cross-cloud or hybrid configurations more challenging without significant workarounds.

In contrast, K3S is designed with flexibility at its core, offering a high degree of customization. The absence of pre-configured services allows users to tailor the environment according to their specific requirements. For example, users can select from various Container Network Interface (CNI) plugins, such as Flannel or Calico, depending on their networking needs. Moreover, K3S's lightweight architecture supports the deployment of custom monitoring solutions and the integration of third-party services without the restrictions imposed by a managed service like AKS. This flexibility is particularly advantageous in scenarios where bespoke configurations are necessary, such as specialized workloads, non-standard environments, or when integrating with a mix of on-premises and cloud-based systems. However, this customization comes with the trade-off of requiring more effort, expertise, and hands-on management.

6.2. Ease of Deployment and Integrations

The ease of deploying applications and integrating with other tools and services is generally similar between AKS and K3S, particularly in the context of deploying the 'Online Boutique' application.

Deployment Process

Deploying the "Online Boutique" application on both AKS and K3S follows a similar process, utilizing YAML manifests to configure and launch the microservices. These tools provide a consistent deployment experience across both platforms, enabling users to define and manage their applications declaratively.

In addition to YAML manifests for deploying the application, Helm is used in both environments to deploy essential system services, such as the kube-prometheus-stack, which includes Prometheus for monitoring and Grafana for visualization. This consistent approach across AKS and K3S ensures that the monitoring and management infrastructure is deployed in a standardized manner.

However, there are distinctions in how each platform handles networking and service exposure. In AKS, the Azure Load Balancer is used as the service load balancer, providing a fully managed, scalable solution for exposing applications to external traffic. This integration with Azure's cloud infrastructure simplifies the process of making services available over the internet, with built-in support for features like automatic scaling and SSL termination.

In contrast, K3S uses a more lightweight approach with its built-in Klipper load balancer. Klipper allows K3S to expose applications on port 80, making it possible to serve applications directly without requiring an external load balancer. This feature is particularly advantageous in resource-constrained environments or edge computing scenarios, where a full-fledged cloud-based load balancer might be unnecessary or impractical. Despite its simplicity, Klipper effectively handles the essential task of load balancing within a K3S cluster, providing a streamlined solution for service exposure.

Integrations

AKS is inherently designed for deployment in cloud environments, specifically within the Azure ecosystem. Its managed nature, along with deep integration with Azure services, makes it an excellent choice for organizations that are fully invested in Azure. AKS's seamless integration with other Azure tools like Azure DevOps, Azure Security Center, and Azure Policy further strengthens its position as the go-to solution for cloud-native applications within this ecosystem. However, AKS's close ties to Azure mean that it is less adaptable for on-premises or hybrid cloud environments unless these environments are heavily reliant on Azure's infrastructure. While Azure Arc enables some hybrid cloud capabilities, deploying AKS outside of Azure requires significant additional configuration and may not offer the same level of support or performance.

On the other hand, K3S stands out for its versatility across various environments. It is equally at home in cloud, on-premises, or hybrid deployments, making it a strong contender for organizations that need flexibility in where and how their Kubernetes clusters are deployed. K3S's lightweight footprint and reduced system requirements make it particularly well-suited for edge computing, IoT applications, and environments with limited resources. Furthermore, K3S can be integrated with various cloud services, including those from multiple providers, making it an ideal choice for multi-cloud or hybrid strategies. Although this requires more manual setup compared to AKS's integrated approach, it provides organizations with the freedom to optimize their infrastructure across different platforms, choosing the best tools and services from each environment to meet their specific needs.

Learning Curve

The learning curve for AKS is generally lower, particularly for users already familiar with Azure services. The managed nature of the platform, along with its extensive documentation and community support, makes it easier to get started with Kubernetes deployments.

K3S, while also supported by good documentation and a growing community, presents a steeper learning curve. The need for manual setup and configuration, combined with the flexibility it offers, means that users must have a deeper understanding of Kubernetes and the infrastructure it runs on.

6.3. Management, Monitoring and Security Features

The management and security features of AKS and K3S reflect their design priorities and intended use cases.

Management

Azure Kubernetes Service (AKS) leverages Azure's extensive ecosystem to provide robust management capabilities. Through Azure's native tools like Azure Monitor, Azure Policy, and Log Analytics, AKS offers a highly integrated management experience. These tools are designed to work seamlessly with AKS, allowing users to automate monitoring, scaling, and compliance tasks with ease. For organizations already embedded in the Azure ecosystem, this integration simplifies the management of Kubernetes clusters, enabling centralized control and streamlined operations.

K3S, while equally capable of integrating with tools like Rancher, Grafana, and Prometheus, often requires a more hands-on approach to cluster management. Users need to set up and configure these tools manually, giving them the flexibility to tailor their management stack to the specific needs of their environment. This manual setup can be more time-consuming but offers greater customization.

Despite the availability of Rancher and Grafana on both platforms, AKS benefits from the managed nature of the Azure platform, where much of the heavy lifting such as cluster upgrades, scaling, and patching is handled automatically by Azure. In contrast, K3S places more responsibility on the user, requiring more direct management and configuration efforts to maintain the cluster.

Monitoring

Monitoring in AKS leverages the capabilities of the kube-prometheus-stack, which includes Prometheus and Grafana, to provide robust monitoring across the cluster. This setup is consistent across both AKS and K3S, allowing for comprehensive metric collection and real-time visualization through customizable Grafana dashboards. In AKS, the integration with Azure's native tools, such as Azure Monitor and Log Analytics, further enhances the monitoring experience by offering advanced analytics and pre-configured alerts directly through the Azure portal.

One of the key strengths of AKS lies in its alerting capabilities. Azure Monitor's built-in alerting features enable the creation of complex, rule-based alerts that can notify DevOps teams of potential issues before they escalate. These alerts can be configured to trigger on various conditions, such as high CPU usage, memory thresholds, or application errors, ensuring that potential problems are addressed promptly.

In K3S, monitoring is also handled through the kube-prometheus-stack, providing the same foundational tools as in AKS Prometheus for data collection and Grafana for visualization. Although the setup in K3S requires more manual configuration, it offers flexibility in customizing the monitoring stack to fit specific needs, such as integrating additional exporters or customizing alert rules.

Grafana's alerting capabilities are fully available in K3S as well, allowing for the creation and management of alerts based on the metrics collected by Prometheus. This means that you can set up alerting in K3S just as effectively as in AKS, using Grafana's interface to define alert conditions and receive notifications. However, since K3S does not have the same level of integrated cloud services as AKS, the configuration and management of these alerts are more manual, but this also allows for greater customization.

Security Features

Security in AKS is deeply integrated with Azure’s security framework, providing a comprehensive, managed security posture. This includes built-in support for Azure Active

Directory (AAD) for authentication, automated security updates, and native network policies enforced through Calico. Azure Security Center further enhances security by offering advanced threat protection and continuous security assessments, ensuring that AKS clusters adhere to best practices and regulatory requirements with minimal user intervention.

One of the key advantages of AKS is its ease of updates. Azure handles Kubernetes version upgrades and patches automatically, minimizing the risk of human error and ensuring that clusters remain secure and up to date with the latest features and security fixes. This approach to updates is a significant benefit for organizations that prioritize operational efficiency and security.

In contrast, K3S requires users to manually manage updates, which can add complexity and operational overhead, particularly in environments with multiple nodes. Each node must be individually updated to the new version of Kubernetes, which can be time-consuming and requires careful planning to avoid downtime or inconsistencies across the cluster. While this manual approach allows for greater control over the timing and process of updates, it also increases the complexity of maintaining a secure and up-to-date environment.

Operational Overhead

The operational complexity associated with AKS is generally lower, thanks to its managed nature within the Azure ecosystem. Azure handles critical operational tasks such as cluster scaling, security patching and network management.

One of the standout features of AKS is its node pool scaling capability. AKS allows users to easily scale node pools up or down based on demand, with just a few clicks or through automated scaling rules. This ability to quickly adjust the size of the node pool without manual intervention is a significant advantage in dynamic environments where workload demands fluctuate. The automated management of node pools not only saves time but also ensures that resources are used efficiently, without the need for manual provisioning and configuration.

K3S, in contrast, does not offer built-in node pool scaling. When additional capacity is needed in a K3S cluster, users must manually create new virtual machines, configure them with the necessary software, and join them to the cluster. This process, while offering granular control over the infrastructure, is considerably more labor-intensive and time-consuming compared to the automated scaling available in AKS. This increases the operational overhead in K3S environments, particularly in scenarios where scaling is required frequently or rapidly.

6.4. Documentation and Community Support

The quality of documentation and community support plays a significant role in the usability and adoption of any platform, particularly in open-source technologies like Kubernetes. This section compares Azure Kubernetes Service (AKS) and K3S in terms of the support provided by their respective communities and the quality of available documentation.

Documentation

AKS benefits from comprehensive and professionally maintained documentation provided by Microsoft. The documentation is extensive, covering everything from initial setup to advanced configurations, integrations with other Azure services, and best practices for security and scalability. Microsoft's documentation is regularly updated in line with Azure's continuous delivery of new features and updates, ensuring that users have access to the latest information. Additionally, the integration of AKS with Azure's managed services means that the documentation also covers related services like Azure Monitor, Azure Active Directory, and Azure Policy, providing a holistic guide to managing Kubernetes within the Azure ecosystem.

K3S documentation, while not as extensive as AKS's, is well-suited to its target audience of developers and operators working in specialized or resource-constrained environments. The official K3S documentation is concise and focused on getting users up and running quickly, reflecting the lightweight and simplified nature of K3S itself. However, for more complex scenarios or detailed explanations, users may need to rely on broader Kubernetes documentation or community-contributed guides and tutorials. This can make it more challenging for less experienced users to find the specific information they need, particularly for advanced configurations or troubleshooting.

Community Support

AKS benefits from being part of the broader Azure ecosystem, which includes a large and active community of cloud professionals, developers, and DevOps engineers. Microsoft actively supports this community through forums like Microsoft Q&A, GitHub discussions, and Azure-specific community events. The size and engagement of this community ensure that users can quickly find solutions to common problems, share best practices, and stay updated on the latest features and updates. Additionally, Microsoft's official support channels, including Azure support plans, provide a reliable fallback for critical issues that require direct assistance.

K3S, has a growing and vibrant community, particularly among those working in edge computing, IoT, and resource-constrained environments. The community, primarily centred around the open-source ecosystem, is very active on platforms like GitHub, Rancher forums, and Kubernetes-specific Slack channels. While the community may not be as large as that of AKS, its members are typically very knowledgeable about Kubernetes and open-source technologies, which can lead to high-quality support and collaboration. However, K3S users may have to rely more on community support than formal channels, especially since K3S is often deployed in non-standard environments where specific issues may not have as much coverage in broader Kubernetes discussions.

7. Discussion

This chapter presents a thorough analysis of the findings from the comparative study of Azure Kubernetes Service (AKS) and K3S. It is structured to assess the strengths and weaknesses of each platform across key criteria, offer recommendations for selecting the most suitable orchestration tool, and explore the implications for future work. To accommodate different reader preferences, the results are intentionally presented in various formats. Whether readers are looking for a high-level summary or a detailed analysis, this structure provides flexibility in how the information is accessed. Readers are encouraged to focus on the sections most relevant to their interests and can skip any subsections that offer alternative presentations of the same data.

7.1. Summary of Findings

This thesis has conducted an in-depth evaluation of AKS and K3S, focusing on key factors such as cost, performance, resource utilization, scalability, management complexity, and security. The findings highlight significant differences between these platforms, which are crucial for organizations when selecting a Kubernetes orchestration tool.

Key Findings:

1. **Resource Utilization:**
    a. **AKS**: Exhibits higher CPU and system load, largely due to additional operational and management containers required for cloud operations. However, it benefits from Azure's resource optimization and scalability features.
    b. **K3S**: Demonstrates lower CPU and system load, making it more efficient in resource-constrained environments, such as edge computing or on-premises deployments.
2. **Performance analysis:**
    a. **AKS**: While AKS provides adequate performance under various load conditions, it falls behind K3S in raw performance metrics. The additional overhead of managed services and cloud-specific components contributes to this gap.

    b. **K3S**: K3S consistently outperforms AKS in all benchmark tests, including Apache AB, K6, and Sysbench, demonstrating superior CPU efficiency, lower latency, and higher throughput.

3. **Scalability and flexibility:**
    a. **AKS**: Offers superior scalability with built-in autoscaling and seamless integration with Azure services. This makes it the better option for applications requiring rapid scaling and dynamic resource management.
    b. **K3S**: While highly flexible and customizable, K3S requires manual intervention for scaling, which can increase operational complexity but offers greater control over the environment.
4. **Cost Analysis**:
    a. **AKS**: The total cost of deploying and running an application on AKS over a five-year period is significantly lower, at **$12,498.60**, compared to **$28,135** for K3S. This cost advantage is primarily due to Azure's managed services, which reduce operational overhead and optimize resource allocation.
    b. **K3S**: The higher cost associated with K3S is due to the need for on-premises infrastructure, manual management, and higher resource utilization. Despite this, K3S offers superior performance, which may justify the higher expenditure in performance-critical scenarios.
5. **Ease of deployment and integration:**
    a. **AKS**: Simplifies deployment through Azure's GUI and Terraform scripts, making it accessible even to teams with limited Kubernetes expertise. It also integrates seamlessly with Azure's cloud services.
    b. **K3S**: Requires a more manual setup process, which can be complex but offers greater flexibility in customization. This makes it ideal for environments where specific configurations are necessary, such as on-premises or edge deployments.
6. **Security and management complexity:**
    a. **AKS**: Integrated with Azure's security framework, including Azure Active Directory and Azure Security Center, AKS provides a managed, comprehensive security posture. It also benefits from lower management complexity due to automated updates and scaling.
    b. **K3S**: Requires more hands-on management for security configurations, increasing the operational overhead. However, its open-source nature allows for greater customization and integration with third-party security tools.

7.2. Comparative strengths and Weaknesses of Each Tool

This section evaluates the strengths and weaknesses of AKS and K3S based on the comprehensive data gathered during this study.

Azure Kubernetes Service (AKS)

Strengths:

- **Cost-effective in cloud environments**: AKS provides significant cost savings through its integration with Azure's managed services, particularly when leveraging Azure's pricing models like Reserved Instances and Spot VMs. This makes AKS an attractive option for organizations looking to optimize their cloud spending while maintaining robust Kubernetes operations.
- **Ease of management**: One of the standout features of AKS is its ease of management, particularly through automated processes such as scaling, security patching, and updates. This reduces the need for extensive in-house Kubernetes expertise and allows organizations to focus more on application development and less on infrastructure management.
- **Scalability**: AKS's built-in autoscaling capabilities are invaluable for applications that experience varying workloads, providing seamless scalability that adjusts dynamically to meet demand without requiring manual intervention.
- **Security**: AKS benefits from deep integration with Azure's security tools, including Azure Active Directory, Azure Security Center, and built-in compliance checks. This comprehensive security framework ensures that AKS deployments are secure by default, reducing the risk of vulnerabilities and improving overall compliance.

Weaknesses:

- **Performance overhead**: AKS introduces additional overhead due to its managed services, which can result in lower performance compared to K3S in certain scenarios. For applications where maximum throughput and minimal latency are critical, this can be a disadvantage.
- **Complexity in cross-cloud and hybrid deployments**: AKS can be challenging to integrate into multi-cloud or hybrid cloud environments where resources span across

different cloud providers. This limitation could complicate deployments for organizations pursuing a multi-cloud strategy.

K3S

Strengths:

- **Superior performance**: K3S consistently outperforms AKS in all benchmark tests, making it a better choice for performance-critical applications. Its lightweight architecture and efficient resource utilization enable it to handle higher workloads with lower latency.
- **Resource efficiency**: K3S's low resource consumption makes it ideal for environments with limited capacity. Its ability to operate efficiently on less powerful hardware is a significant advantage in resource-constrained settings.
- **Flexibility and customization**: K3S offers a high degree of flexibility, allowing users to tailor the environment to their specific needs. This is particularly beneficial in specialized environments such as edge computing, IoT, or on-premises deployments.

Weaknesses:

- **Higher cost in on-premises deployments**: The cost of running K3S in an on-premises environment is significantly higher than AKS, primarily due to the need for physical infrastructure, manual management, and higher resource utilization.
- **Complex management**: K3S requires more hands-on management, including manual updates and scaling. This increases the operational complexity and may necessitate a higher level of expertise from the IT staff.
- **Limited built-in security**: Unlike AKS, which benefits from Azure's managed security features, K3S requires manual configuration of security protocols. This adds to the management overhead and increases the risk of misconfigurations.

Table 20. Detailed Comparative Analysis of Azure Kubernetes Service (AKS) and K3S Based on Criteria

Criteria	**AKS**	**K3S**	**Winner**
Resource Utilization (Idle State)	Higher CPU and System Load (42.3% CPU, 99.5% Load)	Lower CPU and System Load (12.5% CPU, 17.5% Load)	K3S
Performance (Apache AB Test)	34 requests/s, 29.413 ms/request	109.44 requests/s, 9.137 ms/request	K3S
Performance (K6 Test)	22.24 requests/sec, 13.22 s/request	109.92 requests/sec, 1.57 s/request	K3S
Performance (Sysbench CPU Speed)	8050.65 events/s	20219.14 events/s	K3S
Cost (5-Year Total)	$12,498.60	$28,135.00	AKS
Implementation and Configuration	Easy with GUI and Terraform scripts, integrated with Azure	Hands-on, manual setup with more flexibility	AKS (Ease); K3S (Flexibility)
Ease of Deployment and Integrations	Seamless Azure integration, GUI-based deployment	Flexible, supports diverse environments, manual setup	AKS (Ease); K3S (Versatility)
Management Complexity	Lower - automated updates and scaling	Higher - manual updates and scaling	AKS
Scalability	High - built-in autoscaling	Medium - requires manual intervention	AKS
Flexibility and Customization	High - Azure-centric with extensive options	High - open-source and flexible	Tie
Security	Integrated with Azure Security, automated patches	Manual security configurations, higher overhead	AKS
Documentation and Community Support	Extensive, backed by Microsoft, large community	Growing, focused community, strong in edge computing	AKS

Overall Winner	AKS (6/13 criteria)	K3S (4/13 criteria)	AKS Wins

While AKS stands out as the overall winner, the choice between AKS and K3S ultimately depends on the specific use case and the expertise of the DevOps team. AKS offers greater ease of deployment, integrated management, and strong Azure service integration, making it suitable for teams working within the Azure ecosystem. On the other hand, K3S excels in performance, resource efficiency, and flexibility, particularly in smaller, resource-constrained environments or edge computing scenarios. The right platform should be selected based on the project's needs and operational expertise, as outlined in the detailed comparative analysis shown in Table 20.

7.3. Recommendations for Selecting an Orchestration Platform

When deciding between AKS and K3S, organizations should carefully consider their specific needs, strategic goals, existing infrastructure and team expertise. Below are recommendations tailored to different scenarios:

When to choose AKS:

- **Cloud-native applications**: For organizations deeply integrated with Azure, AKS is the logical choice due to its seamless integration with Azure services, which simplifies deployment and management. This is particularly advantageous for teams already utilizing other Azure services, as it creates a cohesive ecosystem with centralized management and support.
- **Ease of management**: AKS is ideal for organizations looking to reduce operational overhead. The platform’s managed services automate many routine tasks, such as scaling and security patching, freeing up IT resources to focus on more strategic initiatives.
- **Cost efficiency**: AKS is more cost-effective for cloud deployments, especially when considering Azure’s flexible pricing models, such as Reserved Instances and Spot VMs. For organizations looking to minimize long-term costs while benefiting from robust infrastructure, AKS presents a compelling option.

- **Scalability needs:** If an organization's workload is expected to fluctuate significantly, AKS's autoscaling capabilities ensure that resources are allocated dynamically and efficiently, avoiding both underutilization and overprovisioning.

When to choose K3S:

- **Performance-critical applications**: K3S excels in environments where performance is paramount. Its superior benchmarks in throughput and latency make it a better choice for applications requiring high performance, such as real-time data processing, IoT, and edge computing.
- **Customization and flexibility**: K3S is ideal for organizations that need a high degree of customization in their Kubernetes environment. Whether for specialized workloads or unique operational requirements, K3S's open-source flexibility allows for tailored configurations that might not be possible with a more managed solution like AKS.
- **On-premises and edge deployments**: For deployments outside the cloud, particularly in on-premises data centers or at the edge, K3S's lightweight architecture and efficient resource usage make it a strong candidate. It is well-suited for environments where cloud resources are not feasible or where local processing power is critical.
- **Existing infrastructure**: If an organization already has significant on-premises infrastructure, the higher initial and operational costs of K3S might be offset by its performance benefits and the ability to integrate with existing systems. K3S's performance gains can justify the investment in these scenarios, particularly when high throughput and low latency are required.

7.4. Gartner Magic Quadrant for Container Management

As part of this analysis, it's important to consider the broader industry context in which AKS operates. Microsoft's Azure Kubernetes Service has been recognized as a Leader in the 2023 Gartner Magic Quadrant for Container Management, as shown in Figure 27 [39].

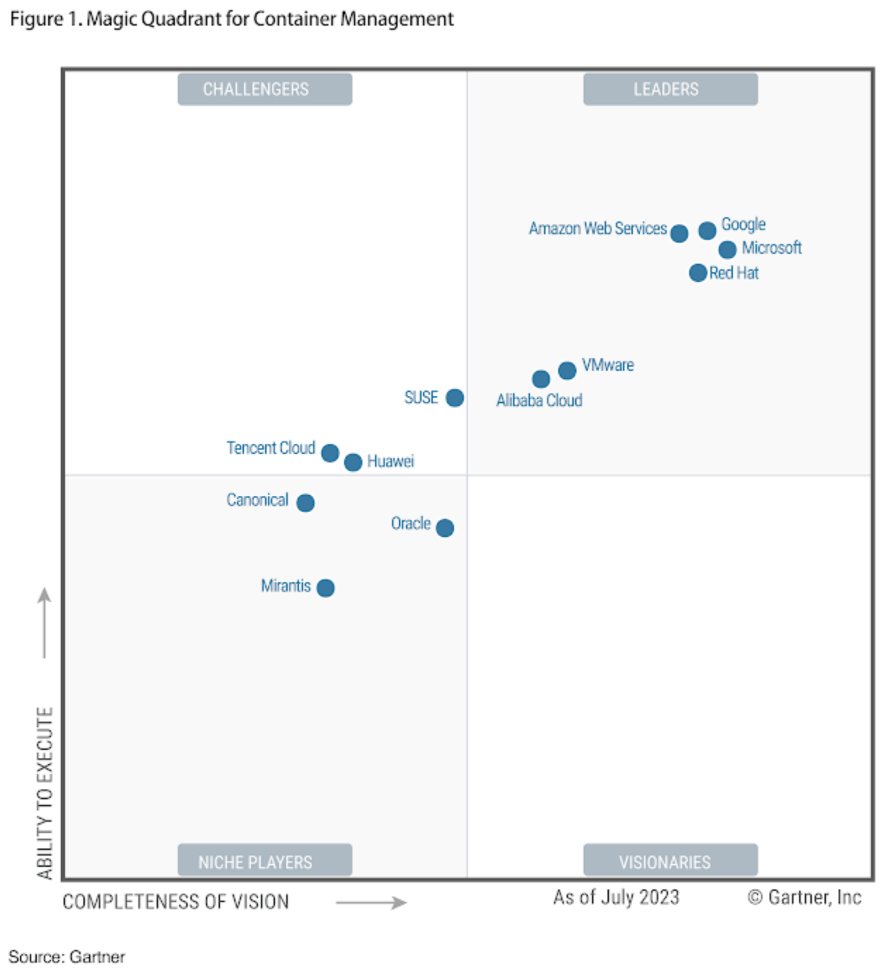


Figure 27. Magic Quadrant for Container Management. Source: [39].

The Gartner Magic Quadrant for Container Management is a significant industry benchmark that evaluates vendors based on their ability to execute and the completeness of their vision. Microsoft's position as a Leader, alongside other industry giants like Google, Amazon Web Services and Red Hat underscores its strong market presence and the robust capabilities of AKS.

In contrast, K3S, while highly capable, is positioned differently in the market as a product developed by Rancher Labs (part of the SUSE ecosystem) which Gartner places in the "Challengers" quadrant. This positioning reflects SUSE's solid ability to execute but a less complete vision compared to leaders like Microsoft. However, for specific use cases such as on-premises or edge computing, K3S remains a powerful and flexible tool, even if it does not have the same broad market penetration or recognition as AKS.

In addition to Azure Kubernetes Service (AKS) and K3S, it is important to consider other popular container orchestration tools that excel in different deployment environments. Among these, **Amazon Web Services (AWS)** and **OpenShift** are two key players in the Kubernetes space. AWS EKS is widely adopted for its extensive cloud-native capabilities, while OpenShift is recognized for its hybrid and on-premises solutions. By comparing these tools with AKS and K3S, we can better understand how each platform fits into various cloud and on-premises deployment scenarios.

Amazon Web Services (AWS)

AWS, through its Elastic Kubernetes Service (EKS), is also a Leader in the container management space. Like AKS, AWS EKS offers seamless cloud integration and managed services, making it an ideal choice for enterprises fully committed to cloud-based infrastructure. In comparison to AKS, AWS EKS provides similar capabilities but stands out for its vast ecosystem and flexibility in deploying services across AWS's extensive global infrastructure.

OpenShift (Red Hat Kubernetes Platform)

OpenShift, recognized as a Leader for its hybrid and multi-cloud Kubernetes capabilities, offers a powerful on-premises solution. Compared to AKS, which is tightly integrated into the Azure ecosystem, OpenShift offers broader flexibility in managing Kubernetes clusters across multiple environments. For enterprises looking for a robust, on-premises solution with strong developer tools, **OpenShift** stands out. Compared to **K3S**, OpenShift is more enterprise-focused, offering additional features like integrated CI/CD pipelines and developer workflows, but it comes with a higher resource and operational overhead, making it more suitable for larger deployments.

8. Conclusion

This thesis conducted an extensive comparison between Azure Kubernetes Service (AKS) and K3S to determine their suitability for orchestrating a medium complexity microservices application, exemplified by the "Online Boutique" application. The aim was to provide a thorough understanding of how each platform performs in both cloud and on-premises environments, particularly in terms of performance, cost, management complexity, and scalability.

The study's quantitative analysis demonstrated distinct differences between the two platforms. AKS, deeply integrated with the Azure ecosystem, showed significant cost advantages over a five-year period when compared to K3S, particularly in a scenario involving only five virtual machines (VMs). This cost-efficiency stems from Azure's managed services, which reduce the operational overhead and optimize resource allocation. The total cost of ownership for AKS was markedly lower at \$12,498.60, compared to \$28,135 for K3S.

However, K3S outperformed AKS in all key performance benchmarks, including CPU speed, memory transfer rate, and request-handling capabilities. K3S's lightweight architecture allowed it to deliver superior throughput and lower latency, making it the better choice for performance-critical applications. The analysis also showed that K3S has a lower CPU and system load in idle states, which is advantageous for resource-constrained environments. This performance difference is also influenced by the hardware and virtualization factors, such as K3S’s larger L2 cache, optimized CPU architecture, and lower hypervisor overhead compared to AKS.

The qualitative comparison further illuminated the strengths and weaknesses of each platform. AKS was noted for its ease of management, particularly in cloud environments, where its automated updates, built-in security features, and seamless integration with Azure services simplified operations. This makes AKS an attractive option for organizations looking to minimize their operational overhead and focus on application development.

On the other hand, K3S excelled in flexibility and customization. Its open-source nature allows for a high degree of tailoring, making it ideal for on-premises deployments or edge computing scenarios where specific configurations are necessary. K3S's ability to run efficiently on less powerful hardware also makes it suitable for environments with limited resources.

8.1. Comprehensive Comparison and Final Recommendations

The findings of this thesis suggest that the choice between AKS and K3S should be guided by an organization's specific needs and existing infrastructure:

- **For cloud-native applications or organizations heavily invested in the Azure ecosystem**, AKS is the preferred choice. It offers significant cost savings, ease of management, and seamless integration with other Azure services. Its built-in scalability and security features make it a robust option for enterprises looking to leverage the full potential of cloud infrastructure without the need for extensive in-house Kubernetes expertise.
- **For performance-critical applications or organizations with significant on-premises infrastructure**, K3S is the superior choice. Its lightweight architecture and efficient resource utilization allow it to deliver better performance, especially in scenarios requiring high throughput and low latency. Moreover, for organizations with high server utilization, K3S could be more cost-effective, as it maximizes the use of existing physical infrastructure.

In conclusion, this thesis has provided a detailed comparison of AKS and K3S, offering valuable insights into their strengths and weaknesses. The decision between these two platforms should be made based on a thorough understanding of an organization's current infrastructure, strategic goals, and specific application requirements. Whether optimizing for cost, performance, or operational simplicity, this analysis serves as a comprehensive guide to selecting the optimal Kubernetes orchestration tool for diverse deployment environments.

Additionally, this thesis makes several significant contributions:

- **Documented Deployment Process and Code**: A fully documented process and code for deploying a microservices-based application using both AKS and K3S, available via GitHub repository and accompanied by step-by-step instructions.
- **Benchmark Tests and Results Interpretation**: Detailed benchmark tests were conducted, including Apache AB, K6, and Sysbench, with comprehensive analysis and interpretation of the results.
- **Qualitative Comparison**: An in-depth comparison of key orchestration tool properties, such as resource utilization, performance, cost, and scalability.

- **Cost Analysis Methodology**: A clear methodology for conducting cost analysis, comparing the total cost of ownership (TCO) between AKS and K3S, highlighting cost-saving strategies for both cloud-based and on-premise solutions.
- **Recommendations for Selection**: Clear recommendations for choosing between AKS and K3S based on specific application needs, infrastructure, and goals.

8.2. Implications for Future Work

The comparative analysis of AKS and K3S reveals several areas where further research and development could provide additional value to organizations, particularly as the landscape of Kubernetes and cloud computing continues to evolve.

Exploration of cost optimization:

Additional studies should focus on further optimizing costs within AKS, such as exploring the use of Azure's Spot VMs, Reserved Instances, or hybrid benefit programs. Understanding how to maximize cost savings while maintaining performance could help organizations make more informed decisions about their cloud strategies.

For K3S, cost optimization efforts could examine how organizations can leverage synergies with their existing infrastructure, including the use of existing on-premise hardware, energy-efficient configurations, and virtualization solutions like Proxmox. Exploring opportunities to reduce operational overhead by automating management tasks and optimizing resource allocation in K3S clusters would further enhance the cost-effectiveness of on-premise deployments.

Long-term performance studies:

Conducting longitudinal studies that evaluate the performance of AKS and K3S over extended periods and under varied workloads would provide valuable insights into their long-term viability. Such research would be particularly beneficial for understanding how each platform handles scaling, resource allocation, and overall cost-efficiency in real-world scenarios.

Hybrid and multi-cloud strategies:

Future research should explore how AKS and K3S can be integrated within hybrid or multi-cloud architectures. Leveraging AKS for cloud-native workloads while deploying K3S in on-premises or edge environments could offer a balanced approach, combining the strengths of both platforms. Such strategies would allow organizations to maintain flexibility, optimize costs, and ensure that applications are deployed in the most suitable environment.

Security and compliance enhancements:

As security remains a top priority for organizations, future work should focus on enhancing the security features of K3S, possibly through automation or integration with enterprise-grade security tools. This would make K3S more attractive to organizations with stringent security and compliance requirements, particularly in regulated industries.

References

[1] Linux Foundation, "Cloud Native Computing Foundation," 2024. [Online]. Available: https://www.cncf.io/.

[2] S. Newman, Monolith to Microservices, O'Reilly Media, Inc., 2019.

[3] B. Burns, J. Beda and K. Hightower, Kubernetes: Up and Running: Dive into the Future of Infrastructure, O'Reilly Media, 2019.

[4] Kubernetes Authors, "Kubernetes Components," 2024. [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/.

[5] Microsoft Corporation, Inc., "Azure Kubernetes Services (AKS) core concepts - Azure Kubernetes Service," 1 August 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/core-aks-concepts.

[6] K3s Project Authors, "K3s," 2024. [Online]. Available: https://k3s.io/.

[7] S. Newman, Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, 2015.

[8] S. Newman and M. Dorian, Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, Ascent Audio, 2021.

[9] H. v. Merode, Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines, Apress, 2023.

[10] M. Krief, Learning DevOps, Packt Publishing, 2019.

[11] F. Harris, Cloud Native Architecture: Efficiently moving legacy applications and monoliths to microservices and Kubernetes, bpb, 2024.

[12] "Managed Kubernetes Service (AKS) | Microsoft Azure," Microsoft, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/kubernetes-service.

[13] Google LLC, "Online Boutique: Microservices demo.," 2020-2024. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo/tree/main.

[14] HashiCorp, "Terraform by HashiCorp," 2024. [Online]. Available: https://www.terraform.io/.

[15] Broadcom Inc., "VMware vSphere documentation," 2024. [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/index.html.

[16] Red Hat, "Homepage | Ansible Collaborative," 2024. [Online]. Available: https://www.ansible.com/.

[17] Kubernetes Contributors, "Installation Guide - Ingress-Nginx Controller," 2024. [Online]. Available: https://kubernetes.github.io/ingress-nginx/deploy/.

[18] Prometheus Authors, "Prometheus - Monitoring system & time series database," 2024. [Online]. Available: https://prometheus.io/.

[19] Grafana Labs, "Grafana: The open observability platform | Grafana Labs," 2024. [Online]. Available: https://grafana.com/.

[20] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Addison-Wesley Signature Series (Fowler)), Addison-Wesley Professional, 2010.

[21] The Apache Software Foundation, "ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4," 2024. [Online]. Available: https://httpd.apache.org/docs/current/programs/ab.html.

[22] Grafana Labs, "Load testing for engineering teams | Grafana k6," 2024. [Online]. Available: https://k6.io/.

[23] Kubenet Team, "kubenet," 2024. [Online]. Available: https://learn.kubenet.dev/.

[24] Tigera, Inc., "Calico Documentation," 2024. [Online]. Available: https://docs.tigera.io/.

[25] A. Janach, "ajanach/comparison-of-orchestration-systems-for-microservices-applications," 2024. [Online]. Available: https://github.com/ajanach/comparison-of-orchestration-systems-for-microservices-applications.

[26] A. Keesari, "Create Azure Kubernetes Service (AKS) using terraform | by Anji Keesari | Medium," 4 September 2023. [Online]. Available:

https://medium.com/@anjkeesari/create-azure-kubernetes-service-aks-using-terraform-4a847464501c.

[27] D. Odazie, "Kubernetes Secrets – How to Create, Use, and Manage," 25 October 2022. [Online]. Available: https://spacelift.io/blog/kubernetes-secrets.

[28] A. learning, "Key Vault Integration with AKS — Azure | by Always learning | Medium," 23 May 2024. [Online]. Available: https://ibrahims.medium.com/key-vault-azure-a2b5729fdfe8.

[29] Kubernetes Authors, Horizontal Pod Autoscaling, 2024.

[30] Microsoft, "Azure Monitor features for Kubernetes monitoring," 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-overview.

[31] S. Cuff, "14. On AKS Automatic, Azure Advisor Cost Optimization workbook, and Navigating your career in IT.," 2024. [Online]. Available: https://www.linkedin.com/pulse/14-aks-automatic-azure-advisor-cost-optimization-workbook-sonia-cuff-etdqc/.

[32] K3s Project Authors, "Architecture | K3s," 2024. [Online]. Available: https://docs.k3s.io/architecture.

[33] CoreOS, "flannel-io/flannel: flannel is a network fabric for containers, designed for Kubernetes," 2024. [Online]. Available: https://github.com/flannel-io/flannel.

[34] S. James and V. Lancey, Networking and Kubernetes: A Layered Approach, O'Reilly Media, 2021.

[35] Microsoft Corporation, Inc., "Total Cost of Ownership (TCO) Calculator | Microsoft Azure," 2024. [Online]. Available: https://azure.microsoft.com/en-us/pricing/tco/calculator/.

[36] O. Brightrose, "What is Azure TCO Calculator," 2020. [Online]. Available: https://www.cloudysave.com/azure/what-is-azure-tco-calculator/.

[37] Microsoft Corporation, Inc., "Pricing Calculator | Microsoft Azure," 2024. [Online]. Available: https://azure.microsoft.com/en-us/pricing/calculator/.

[38] Proxmox Server Solutions GmbH, "Proxmox - Powerful open-source server solutions," 2024. [Online]. Available: https://www.proxmox.com/en/.

[39] Gartner, Inc, "Gartner Magic Quadrant for Container Management," 20 September 2023. [Online]. Available: https://www.gartner.com/en/documents/4763231.

[40] S. Bostandoust, "ssbostan/kubernetes-complete-reference: Kubernetes reference, awesome, cheatsheet, concepts, tools, examples," 2024. [Online]. Available: https://github.com/ssbostan/kubernetes-complete-reference/tree/master.

List of Tables

Table 1. Summary of Benefits and Challenges in Microservices Architecture. Source [2]. ... 10

Table 2. Summary of Key Features and Functions in Orchestration Tools. Source [3]. 11

Table 3. Use Cases for Kubernetes 12

Table 4. Overview of Key Orchestration Tools for Microservices 13

Table 5. Deployment Methods for Kubernetes 18

Table 6. Specifications for Each Worker Node in Azure Kubernetes Service 27

Table 7. Specifications for Each Control Plane Node in On-Premises K3S Environment...... 27

Table 8. Specifications for Each Worker Node in On-Premises K3S Environment 28

Table 9. Configuration of virtual machines for the K3S Kubernetes cluster in vSphere 40

Table 10. Detailed Comparison of Pod and Container Distribution Between Azure Kubernetes Service (AKS) and K3S 48

Table 11. AKS Cluster Resource Utilization (Idle State with "Online Boutique" Deployed) 49

Table 12. K3S Cluster Resource Utilization (Idle State with "Online Boutique" Deployed) . 49

Table 13. Storage and Network Performance (Idle State with "Online Boutique" Deployed) 50

Table 14. Apache AB Benchmark Results for AKS and K3S 54

Table 15. K6 Benchmark Results for AKS and K3S 61

Table 16. System Benchmark Results Comparison for AKS and K3S 66

Table 17. AKS worker node sysbench output summary 68

Table 18. K3S worker node sysbench summary 68

Table 19. Highlights of *lscpu* output 69

Table 20. Detailed Comparative Analysis of Azure Kubernetes Service (AKS) and K3S Based on Criteria 94

List of Figures

Figure 1. The Components of a Kubernetes Cluster. Source: [4]. 16

Figure 2. Kubernetes Cluster Architecture. Source: [4]. 17

Figure 3. The "microservices-demo" interconnection of microservices. Source: [13]. 23

Figure 4. Create Azure Kubernetes Service (AKS) using Terraform. Source: [26]. 35

Figure 5. Retrieving the external IP address and accessing the "Online Boutique" application through a browser on Azure Kubernetes Service (AKS) 36

Figure 6. Accessing the Grafana dashboard through port forwarding on Azure Kubernetes Service (AKS) 37

Figure 7. vCenter Web UI Displaying the Creation of Five Virtual Machines for the K3S Kubernetes Cluster 41

Figure 8. K3s Infrastructure for Online Boutique Deployment 43

Figure 9. Recommended High-Availability K3s Setup. Source: [32]. 44

Figure 10. Retrieving the external IP address and accessing the "Online Boutique" application through a browser on K3s 45

Figure 11. Grafana AKS Cluster Resource Utilization 51

Figure 12. Grafana K3S Cluster Resource Utilization 52

Figure 13. Apache AB Benchmark Results for AKS and K3S 54

Figure 14. Comparison of Average Requests per Second Between AKS and K3S 55

Figure 15. System Load on AKS Worker Nodes During Apache AB Testing 56

Figure 16. System Load on K3S Worker Nodes During Apache AB Testing 57

Figure 17. K6 Benchmark Results for AKS and K3S 62

Figure 18. Diagram of K6 Benchmark Results for AKS and K3S 63

Figure 19. System Load on AKS Worker Nodes During K6 Testing 64

Figure 20. System Load on K3S Worker Nodes During K6 Testing 65

Figure 21. System Benchmark Results for AKS and K3S 67

Figure 22. Total Cost of Ownership Over 5 Years: On-Premises vs. Azure. 76

Figure 23. Cost Distribution of On-Premises and Azure Infrastructure Over 5 Years. 76

Figure 24. Detailed Cost Breakdown for On-Premises and Azure Over 5 Years 77

Figure 25. Estimated Monthly Cost for Azure AKS Setup. 78

Figure 26. 5-Year Cost Comparison: AKS Setup vs. 5 VMs in Azure 79

Figure 27. Magic Quadrant for Container Management. Source: [39]. 97

List of Codes

Code 1. Terraform configuration script for deploying an Azure Kubernetes Service (AKS) cluster....................32

Code 2. K6 test script for simulating virtual user load on AKS and K3S....................60

Appendix

The source code, configuration files, and automation scripts used in this thesis are available in the following GitHub repository: https://github.com/ajanach/comparison-of-orchestration-systems-for-microservices-applications