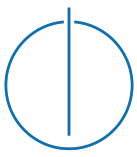
SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Next-Generation Orchestration Frameworks for Multicluster Cloud-Edge Integration

Jakob Kempter



INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

Next-Generation Orchestration Frameworks for Multicluster Cloud-Edge Integration

Orchestrierungs-Frameworks der nächsten Generation für Multicluster Cloud-Edge-Integration

Author:	Jakob Kempter
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan, Giovanni Bartolomeo
Submission Date:	14.06.2024

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 14.06.2024 Jakob Kempter

Acknowledgments

I am deeply grateful to my friends and colleagues who have provided advice and support over the years. I consider myself fortunate to have had you by my side.

My heartfelt thanks go to my parents for their unwavering support and encouragement throughout my studies; this achievement would not have been possible without them. A special thank you to my girlfriend Julia for her constant encouragement and belief in me.

I would also like to thank my university, TUM, and all the professors and assistants who guided me along this path. Special thanks to Giovanni Bartolomeo, my supervisor during the final thesis at TUM, for his invaluable guidance and support, as well as to Nitinder Mohan.

Many thanks to my supervisor at inovex, Maximilian Bischoff, for his invaluable technical expertise and generous investment of time.

Thank you all!

Munich, June 2024

Abstract

Managing and coordinating multiple clusters across cloud and edge environments poses numerous challenges, including the need for scalability, seamless communication, and infrastructure abstraction. Services in such environments can be deployed either in the cloud or at the edge. While edge networks are prone to failures due to the unpredictable behavior of edge nodes, cloud environments typically offer more stable networks. Deploying applications at the edge brings them closer to users, whereas resource-intensive applications are better suited for cloud deployment. This thesis presents a practical implementation of an integration tool that seamlessly connects Oakestra, an orchestration framework for edge computing, and Kubernetes, the defacto standard for cloud orchestration. By leveraging Kubernetes extensions such as Custom Resource Definitions (CRDs) and Controllers, Kubernetes Clusters can be used as Oakestra child clusters to enable cloud-to-edge multi-cluster operations. Additionally, a new Container Network Interface (CNI) is developed, incorporating Oakestra's networking technology. Although the architecture limits direct comparison with current alternatives, the integration's performance in terms of overhead and deployment time is comparable to competitive solutions designed for Kubernetes-native and cloud environments. Furthermore, it is demonstrated that the integration of Kubernetes clusters does not adversely impact the Oakestra root.

Kurzfassung

Die Verwaltung und Koordinierung mehrerer Cluster in Cloud- und Edge-Umgebungen ist mit zahlreichen Herausforderungen verbunden, darunter die Notwendigkeit von Skalierbarkeit, nahtloser Kommunikation und Abstraktion der Infrastruktur. Dienste in solchen Umgebungen können entweder in der Cloud oder am Edge bereitgestellt werden. Während Edge-Netzwerke aufgrund des unvorhersehbaren Verhaltens von Edge-Knoten anfällig für Ausfälle sind, bieten Cloud-Umgebungen in der Regel stabilere Netzwerke. Die Bereitstellung von Anwendungen am Edge bringt sie näher an die Nutzer heran, während ressourcenintensive Anwendungen besser für die Bereitstellung in der Cloud geeignet sind. In dieser Arbeit wird eine praktische Implementierung eines Integrations vorgestellt, das Oakestra, ein Orchestrierungs-Framework für Edge-Computing, und Kubernetes, den Standard für die Cloud-Orchestrierung, nahtlos miteinander verbindet. Durch die Nutzung von Kubernetes-Erweiterungen wie Custom Resource Definitions (CRDs) und Controllern können Kubernetes-Cluster als Oakestra-Untercluster verwendet werden, um Cloud-to-Edge-Multicluster-Operationen zu ermöglichen. Darüber hinaus wird ein neues Container Network Interface (CNI) entwickelt, das die Netzwerktechnologie von Oakestra einbezieht. Obwohl die Architektur den direkten Vergleich mit aktuellen Alternativen einschränkt, ist die Leistung der Integration in Bezug auf Overhead und Bereitstellungszeit vergleichbar mit konkurrierenden Lösungen, die für Kubernetes-native und Cloud-Umgebungen entwickelt wurden. Außerdem wird gezeigt, dass die Integration von Kubernetes-Clustern keine negativen Auswirkungen auf den Oakestra-Stamm Komponente hat.

Contents

Acknowledgments **iii**

Abstract **iv**

Kurzfassung **v**

1 Introduction **1**
1.1 Problem Statement 1
1.2 Motivation 2
1.3 Contribution 3
1.4 Thesis Structure 3

2 Background **5**
2.1 Multi-Cluster Orchestration 5
2.1.1 Architectural Approaches 8
2.1.2 Versatility in Cluster Management 12
2.1.3 Practical Applications and Use Cases 13
2.2 Multi-Cluster Communication 15
2.2.1 Container Networking 15
2.2.2 Inter-Cluster Networking Solutions 16
2.3 Edge Cloud Continuum 17
2.3.1 Reference Architectures 19
2.3.2 Paradigms 21
2.3.3 Edge Cloud Communication 21
2.3.4 Edge Cloud Orchestration 24
2.4 Existing Orchestration Tools, Frameworks, and libraries 24
2.4.1 Multi-Cluster Orchestration 25
2.4.2 Cloud-To-Edge Orchestration 29

3 Oakestra/Kubernetes Integration **33**
3.1 Requirements 33
3.2 System Design 36
3.2.1 Integration Approaches 36

3.2.2 System Components . . . 38
3.3 Orchestration . . . 41
3.3.1 Oakestra Controller Manager . . . 41
3.3.2 Orchestration Flow . . . 44
3.3.3 State Handling . . . 45
3.3.4 Scheduling . . . 46
3.4 Communication . . . 46
3.4.1 Container Communication . . . 46
3.4.2 Communication between Components . . . 48
3.5 Set Up Kubernetes as Oakestra Cluster . . . 48
3.5.1 Prerequisites . . . 49
3.5.2 Set Up Oakestra . . . 49
3.5.3 Set Up Kubernetes . . . 49
3.5.4 Install Kubernetes Plugin . . . 49

4 Evaluation **52**
4.1 Experimental Set Up . . . 52
4.2 Evaluation Results . . . 53
4.2.1 Overhead Plugin . . . 54
4.2.2 Overhead Management . . . 58
4.2.3 Deployment Time . . . 64
4.2.4 Impact Oakestra Root . . . 66

5 Conclusion **68**
5.1 Limitations and Future Work . . . 69

List of Figures **71**

List of Tables **72**

Bibliography **73**

Appendix **80**

1 Introduction

1.1 Problem Statement

In recent years, cloud orchestration has gained increasing attention, facilitated by a diverse array of tools enabling the seamless management and deployment of cloud resources and applications [20]. However, cloud orchestration is not the optimal solution in some situations, particularly when data processing necessitates proximity to the user or device. Edge computing, which involves placing computing resources closer to the end user, has emerged as a significant trend. Nonetheless, this technology demands specialized orchestration within these edge environments. This approach effectively addresses crucial operational requirements such as reduced latency and enhanced data trust preservation [70]. Similarly, many orchestration tools tailored specifically for the edge facilitate application deployment within this environment. Nonetheless, what remains lacking is the integration of these two paradigms: an orchestrator that amalgamates the benefits of both cloud and edge, facilitating deployment across both environments seamlessly.
However, this new amalgamation also presents several challenges that are currently being explored by research. The paradigms of cloud and edge differ significantly in fundamental architecture and communication. In the cloud environment, virtualization of cloud resources can create a homogeneous depiction of instances, simplifying service provisioning. Conversely, such homogeneity is absent in the edge environment, as each device may possess varying specifications and architectures. Furthermore, communication poses a challenge; within a data center, constituting a cloud environment, all instances are interconnected via cables, facilitating swift and straightforward communication between instances. Edge devices, however, are not part of this network and exhibit diverse forms of communication.
The existing few endeavors towards Cloud to Edge orchestrators predominantly rely on cloud-based technologies and often fail to devise a novel approach tailored specifically for the edge domain. Additionally, there are currently scant industrial solutions for Cloud to Edge orchestration that adequately encompass all the requisites and potentials of cloud-based solutions.
Applications deployed in the cloud or within the edge environment must communicate seamlessly, whether cloud-to-cloud, cloud-to-edge, or edge-to-edge. Services should be

deployed in the cloud or edge environments based on the application's requirements. The cloud remains suitable if extensive resources are required, but specific latency is not critical. However, where local data processing is crucial, edge instances become preferable. Yet, as requirements change, the ability to migrate seamlessly to the edge environment is essential. Scalability across both environments, alongside robust failure-handling mechanisms, is also crucial. While generally reliable, centralized cloud regions may fail, posing significant challenges that decentralized edge regions can mitigate, demonstrating the interdependence between cloud and edge for resilient infrastructure. Conversely, edge region failures, though typically less impactful globally, underscore the necessity of cloud backup systems [70].

1.2 Motivation

The evolution of computing landscapes is increasingly characterized by the convergence of edge and cloud technologies, responding to the growing demand for high computational power and low-latency data processing. In such contexts, multi-cluster architectures emerge as critical, allowing seamless data and workload management across disparate environments. The rationale for integrating multi-cluster strategies lies in their ability to provide robust, scalable solutions that leverage the strengths of both edge and cloud computing. Edge computing addresses the need for real-time data processing close to data sources, while cloud computing offers substantial resources and power for more intensive computations.

However, current orchestration solutions often face challenges. On one side, traditional cloud orchestrators like Kubernetes are well-suited for handling high-resource demands but typically struggle with the decentralized nature required for effective edge computing. They generally lack native support for multi-cluster environments, relying instead on external projects to bridge this gap. On the other side, edge-focused orchestration platforms excel in managing distributed devices in resource-constrained environments but do not seamlessly extend to cloud environments.

Oakestra, an orchestration platform initially developed for the edge, exemplifies this divide. It has proven effectiveness in managing communication among edge devices, organizing them into manageable clusters, optimizing the distribution of management tasks, and reducing overhead. This thesis aims to extend Oakestra's capabilities to include cloud orchestration. By doing so, it aims to initiate services that demand higher resources, utilizing both cloud data centers and local edge devices. The overarching goal is to enable comprehensive orchestration and management across both environments through a unified endpoint.

This thesis seeks to bridge the gap between the isolated domains of edge and cloud

computing by integrating cloud orchestration capabilities into Oakestra's existing multi-cluster management framework. The envisioned extension will facilitate seamless integration with existing cloud orchestrators, enhancing Oakestra's utility and enabling it to support a broader range of computational scenarios. This endeavor not only leverages Oakestra's proven strengths in edge orchestration but also introduces new opportunities for managing distributed computing resources more effectively, thereby advancing the capabilities of both edge and cloud computing frameworks.

1.3 Contribution

The core contribution of this thesis is the extension of Oakestra to integrate seamlessly with Kubernetes, establishing it as a complementary platform to the de facto standard for container orchestration. This integration enhances Oakestra's capabilities, allowing it to support multi-cluster orchestration from cloud to edge. Furthermore, the thesis presents a blueprint for incorporating other orchestration platforms, broadening the scope and utility of the Oakestra framework. Lastly, it defines the essential capabilities and functionalities required for a multi-cluster cloud-to-edge orchestrator, providing a detailed framework and guidelines for future developments in this area. This contribution bridges the gap between edge-specific applications and broader cloud capabilities, fostering more robust and versatile computing environments. A detailed listing of previous and upcoming technologies and tools is provided to support these innovations, enriching the historical context and foundational knowledge required for advancing this field.

1.4 Thesis Structure

This thesis is structured to systematically address and elaborate on the integration of multi-cluster orchestration across cloud and edge environments, beginning with a foundational discussion in Chapter 1, which introduces the problem statement, outlines the motivation for this research, delineates the contributions, and provides an overview of the thesis structure. Chapter 2 delves into the essential background required for understanding the current technologies in multi-cluster orchestration, starting with an analysis of architectural approaches and versatility in cluster management. This chapter also explores the edge-cloud continuum, including network resilience and the paradigms shaping this field, and concludes with a review of existing orchestration tools, frameworks, and libraries that are pivotal in the industry today. In Chapter 3, the focus shifts to the practical application of these concepts through integrating Oakestra with Kubernetes, detailing the system design, requirements, orchestration

mechanisms, and communication protocols involved. Chapter 4 thoroughly evaluates the implemented system, assessing factors such as overhead, latencies, and bandwidth, and discusses the benefits and challenges observed. The thesis culminates in Chapter 5, which critically reflects the limitations encountered during the study and outlines prospective future work aimed at refining and expanding the orchestration capabilities between edge and cloud environments. This structure is designed to guide the reader through a logical progression from foundational concepts to practical application and critical evaluation, ultimately leading to a conclusive discussion on the future directions of this research.

2 Background

2.1 Multi-Cluster Orchestration

Multi-Cluster Orchestration refers to coordinating and managing applications and services' deployment, scaling, and operation across multiple clusters in a distributed computing environment. This involves efficiently allocating resources, load balancing, and ensuring high availability and fault tolerance across the interconnected clusters. Multi-cluster orchestration enables the seamless integration and coordination of diverse computing resources, facilitating the execution of complex tasks and workflows across a network of connected clusters. This approach optimizes resource utilization, improves scalability, and centralizes distributed application management, ultimately improving overall system performance, reliability, and availability [40].

Traditionally, organizations have developed custom-built multi-cluster orchestration tools to cater to specific needs and infrastructures, effectively managing applications across various clusters. These custom solutions were essential due to the absence of a universally accepted standard in multi-cluster orchestration, making bespoke systems necessary for handling complex deployment and resource optimization scenarios within unique operational environments. Popular tools include Helm, Operators, GitOps frameworks such as Argo and Flux, Kustomize, and the Operator pattern, which have become central to multi-cluster management strategies [14].

Recent trends, however, show a shift towards standardized multi-cluster orchestration solutions, which have started to permeate the market more broadly. Modern solutions bring enhanced features, scalability, and the advantage of established best practices supported by a larger community. This shift is motivated by the need for more robust and scalable orchestration tools to meet the intricate demands of managing distributed applications in today's complex IT landscapes [30].

Centralized Management

One of the key challenges in multi-cluster orchestration is achieving centralized management across diverse clusters. As organizations operate applications across various clusters, ensuring consistent configuration, monitoring, and management becomes increasingly complex. This involves controlling and coordinating tasks, policies, and

resources across distributed clusters from a single control point. By implementing a centralized management approach, maintainers gain better visibility and control over their distributed infrastructure, resulting in more efficient resource allocation, simplified policy enforcement, and improved operational consistency. In addition, centralized management enables the implementation of global policies and enforcement compliance standards across all clusters, reducing operational overheads and improving overall governance [23, 52].

Multi-Cloud vs Multi-Cluster

In distributed computing, it's crucial to distinguish between multi-cluster and multi-cloud strategies, although both aim to optimize how resources are orchestrated and utilized. A multi-cluster setup involves managing multiple clusters under a unified interface, where each cluster functions independently with its own specific configurations and resources dedicated to distinct workloads or applications. This arrangement is designed to enhance resource allocation and manage workloads efficiently across diverse environments [52].

On the other hand, a multi-cloud strategy utilizes multiple cloud computing services from various providers such as AWS, Azure, or Google Cloud. Organizations adopt this approach to mitigate risks such as vendor lock-in [50] to capitalize on cost efficiencies and specialized services offered by different cloud vendors. While multi-cluster and multi-cloud architectures share similar goals—like improving resilience and scalability—they cater to different strategic needs and are implemented differently.

Integrating multi-cluster orchestration within a multi-cloud framework provides substantial benefits. It allows organizations to manage their applications across different cloud platforms without being confined to a single provider, thus enhancing flexibility and resource utilization. This strategy enables the seamless orchestration of resources across various cloud environments, increasing the portability of applications and data. Organizations can shift or replicate workloads between cloud platforms as per their changing requirements, leveraging specific advantages of each cloud provider while maintaining robust and flexible infrastructure management.

Network Latency and Performance

Network latency and performance are critical in multi-cluster orchestration, especially when synchronizing data between geographically distributed clusters. High network latency may significantly impact the performance of distributed applications, resulting in delays and reduced responsiveness. To mitigate these issues, edge computing can process data closer to the source to reduce latency or launch multiple copies of clusters

in geographically dispersed data centers to be closer to the user. In addition, clusters could be launched with different computing resources, with very powerful resources to ensure performance, but also less powerful ones, depending on the requirements of the services [1].

Data Consistency

Maintaining data consistency across multiple clusters presents a significant challenge, especially under conditions such as network partitioning and when managing large volumes of data. The complexity arises from ensuring that all nodes, possibly spread across various locations, have the same data at any given time. Depending on the application's specific needs, distributed databases and data stores can be employed to address this, which support different consistency models—such as eventual consistency or strong consistency. Innovative technologies like Conflict-free Replicated Data Types (CRDTs) and distributed ledgers offer solutions for maintaining data consistency without traditional locking mechanisms. These technologies allow for concurrent updates that can be merged without conflicts, enhancing the system's ability to handle data across disparate clusters efficiently and reliably [5, 64].

Security and Compliance

Securing a multi-cluster environment encompasses several critical aspects, including protecting data in transit and at rest, managing access across different clusters, and adhering to various regulatory requirements. These challenges necessitate robust solutions to ensure comprehensive security and compliance. Implementing centralized security policies through service meshes can significantly enhance security at the communication level between services. Data encryption at rest and in transit is crucial for protecting sensitive information from unauthorized access. Adopting identity and access management (IAM) solutions that integrate seamlessly across various clusters and cloud environments is essential for effective authentication and authorization management. These IAM solutions help maintain strict access controls, ensuring that only authorized users and services can access specific resources, thus upholding the integrity and confidentiality of the data within a multi-cluster architecture [64].

Multi-Cluster Management vs. Multi-Cluster Orchestration

Multi-cluster orchestration is a relatively recent development, although multi-cluster management has been established for some time. This involves managing multiple clusters across various production environments or distinct operational stages such as development, staging, and production. Each environment employs a consistent cluster

configuration but remains separate, enabling centralized governance of these clusters. This setup primarily addresses the creation and administration of clusters rather than the orchestration of services and applications, delineating a fundamental component of multi-cluster orchestrator capabilities [29].

Workload isolation

Workload isolation is a significant advantage of multi-cluster architectures, providing complete independence between clusters. In contrast to single-cluster setups, where namespaces offer only limited isolation susceptible to inherent security constraints, multi-cluster environments ensure that problems such as cluster failures or configuration changes remain isolated to the specific cluster affected [7].

2.1.1 Architectural Approaches

In the context of multi-cluster orchestration, the architecture plays a pivotal role in defining the structure and organization of the distributed computing environment. The architecture encompasses the design, layout, and interconnections of the clusters, the allocation of resources, and the orchestration of workloads across these clusters. One fundamental aspect of multi-cluster architecture is the delineation of levels, ranging from two to three or even more, each representing a distinct layer of the orchestration framework. This section will explore the architectural models prevalent in multi-cluster orchestration, including depicting two-level, three-level, and potentially higher-level architectures. Additionally, it will delve into the implications and considerations associated with each architectural configuration, providing insights into the diverse approaches for orchestrating and managing applications across multiple clusters.

Traditional cluster architectures typically consist of two levels: multiple nodes where one node serves as the control plane and the remaining nodes act as workers. This setup facilitates the management and execution of tasks within a single cluster environment. However, as organizations scale and adopt distributed systems across various environments, the need arises for orchestrating multiple clusters efficiently.

The architecture of Kubernetes fits well with the two-tier cluster model described above, which comprises a control plane and worker nodes. In Kubernetes, the control plane is responsible for managing the state and configuration of the cluster, including scheduling and orchestrating tasks. This control plane usually includes components such as the API server, the controller manager, and the scheduler. The worker nodes, on the other hand, execute the workloads. Each node runs pods, the smallest deployable units managed by Kubernetes, which actually run containerized applications. The worker nodes also have components such as kubelet, which interacts with the control

plane to manage the pods and containers on its node, and kube-proxy, which handles network communication [32].

Multi-cluster architectural designs can be categorized into segmentation and replication strategies. In segmentation, the application is partitioned into distinct components, typically Kubernetes services, distributed across multiple clusters based on operational needs. Conversely, the replication strategy involves creating identical replicas of a Kubernetes cluster across various data centers in different locations, offering redundancy and enhancing application availability in case a cluster fails [60].

2-Level Architecture with no additional control layer

In the expanded multi-cluster orchestration model, each cluster centers around a main node that acts as a hub for control and communication, similar to traditional architectures. However, these main nodes across different clusters are interconnected, as seen in figure 2.1, enabling them to communicate and collaborate effectively. This network of main nodes can redistribute tasks and share resources based on real-time requirements and available capacity, optimizing workload balance and resource usage.

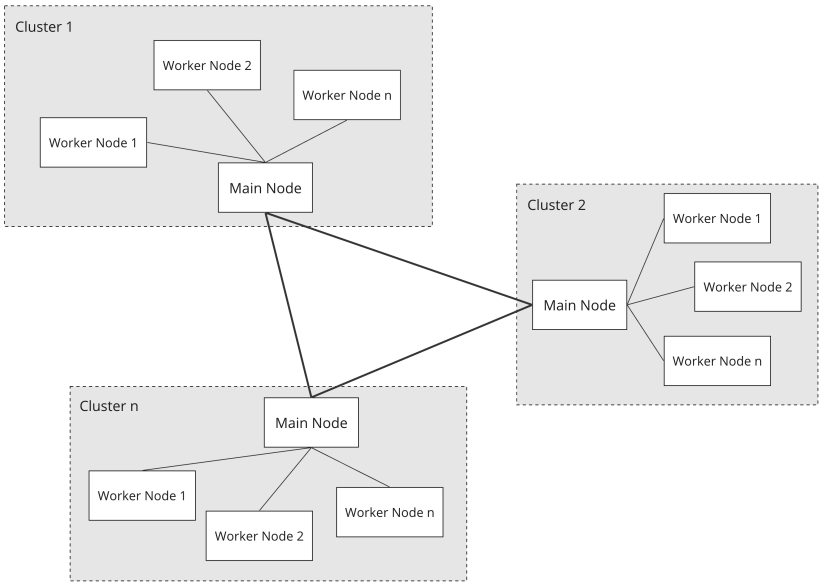


Figure 2.1: Architecture Multi-Cluster: 2-Layer

Incorporating elements from distributed database systems, this architecture also introduces a dynamic leadership and state management system [71]. A designated

leader among the main nodes would coordinate the clusters and manage the overall state from which tasks are assigned and resources are distributed. This leader is not fixed but can be re-elected based on network conditions and node availability. If the current leader node fails or becomes unreachable, a new leader is elected among the nodes to ensure continuity and reliability of operations.

This leadership model, combined with stateful awareness of each node's status and capacity, allows for intelligent decision-making in task distribution and resource allocation. By implementing redundancy and failover mechanisms, tasks can be redirected to other clusters in case of a main node failure, maintaining operational continuity and enhancing system resilience. Additionally, the 2-level architecture requires low network latencies, which is crucial for the high degree of coordination necessary among clusters to manage intensive communication effectively.

3-Level Architecture

In an extended three-tier architecture for multi-cluster orchestration, shown in Figure 2.2, the structure starts with worker nodes at the base level, responsible for performing operational tasks and handling the direct workload. At the next level up, master nodes manage these worker nodes and monitor intra-cluster operations, forming the control plane for each cluster. At the top of this hierarchy is a superordinate "Control Plane" that serves all clusters and acts as the management level for the control levels of the individual clusters. This higher-level node simplifies orchestration by abstracting the complexity of coordinating multiple control levels, enabling centralized management and oversight of different cluster environments. This architecture facilitates strategic resource allocation and task scheduling across clusters and improves system monitoring and resilience by dynamically reallocating resources and managing disaster recovery. This three-tiered approach significantly improves scalability and operational efficiency by centralizing control, making it ideal for managing complex, distributed systems [35].

This architecture is also used by well-known tools such as Kubefed and Karmada. Kubefed, known as Kubernetes Federation, connects multiple clusters to synchronize resources under a single central management structure [34]. Karmada works similarly by distributing and managing workloads across different clusters and using policies to control resource allocation and workload placement without centralizing overall operational control [26]. Both tools can be integrated into the higher levels of this architecture model. Other tools, such as KubeAdmiral, use the same architecture and also have a host cluster that is in control [31]

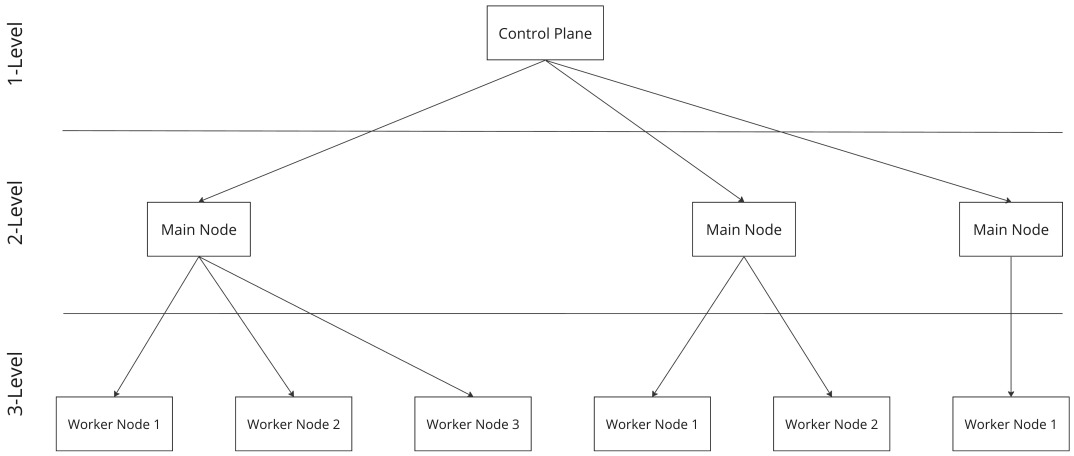


Figure 2.2: Architecture Multi-Cluster: 3-Layer

4+ Level Architecture

In addition, there is the potential for further refinement by introducing a fourth or more layers, as shown in Figure 2.3, which could enable geographic and logical separation within the multicluster architecture. This layer could enable policies and configurations tailored to specific geographic regions or logical divisions within an organization, as predicted in Cloud Edge Continuum [2], increasing flexibility and security.

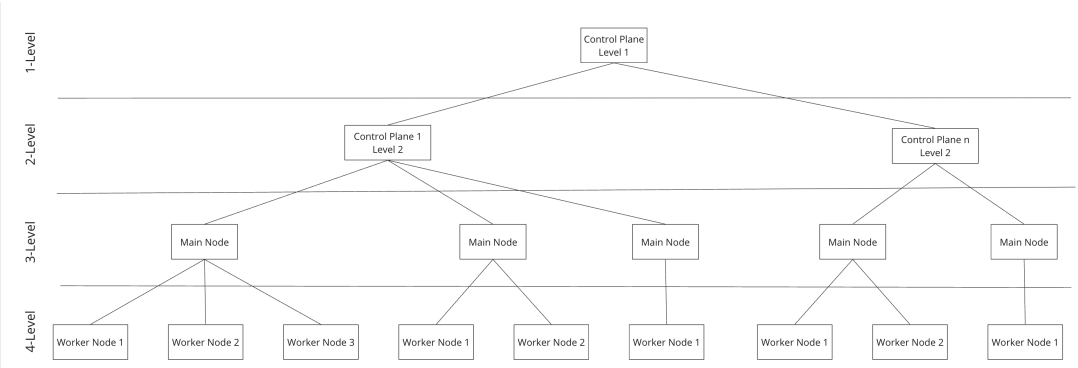


Figure 2.3: Architecture Multi-Cluster: 4(+)-Layer

2.1.2 Versatility in Cluster Management

The concept and versatility of cluster management in a multi-cluster orchestration environment are central to achieving high efficiency and adaptability. This section looks at the strategic configurations and operational dynamics that enhance the versatility of cluster management. By examining and citing various facets, from uniformity to the diversity of cluster configurations, monitoring, control mechanisms, and dynamic scaling, a comprehensive overview is provided of how different approaches can be harmoniously integrated to fulfill the diverse requirements of complex systems. Understanding these aspects is crucial for developing robust and scalable systems precisely tailored to the specific requirements of different operational scenarios.

Uniformity and Diversity in Cluster Configurations

With multi-cluster orchestration, clusters can either be configured uniformly or set up differently, with each approach catering to different operational requirements and preferences. Standardized configurations ensure consistency across all clusters and simplify management, deployment, and maintenance. This standardization is particularly beneficial for companies looking for a streamlined approach where each cluster behaves predictably and integration between clusters is seamless. On the other hand, the variety of cluster configurations allows customization to specific requirements such as processing power needs, storage capacity, and geographical distribution. This flexibility supports customized solutions that effectively handle different workloads and operational requirements.

However, the diversity of cluster configurations also brings a number of challenges, especially when clusters differ not only in their specifications but also in their underlying architectures (e.g., x86 or ARM) or their deployment environments (e.g., edge, cloud, and fog computing). These differences can complicate resource management and require the separation of clusters to optimize performance and efficiency. In addition, different environments can also pose networking challenges. The latency and robustness of the connection to edge devices are usually significantly weaker than in cloud data centers [16]. Furthermore, the variability of clusters - from clusters running on Kubernetes to those using alternative orchestration platforms - can exacerbate these challenges. This necessitates the adoption of a universal deployment language such as TOSCA (Topology and Orchestration Specification for Cloud Applications) to ensure interoperability and optimized management across diverse and complex environments. This universal approach helps to reduce potential resource disparities and simplify the orchestration of multicluster configurations [37]

Multi-Cluster Monitoring and Control

Monitoring and controlling multiple clusters are critical components that are distinct from orchestration. Effective monitoring ensures that performance metrics, system health, and other critical data are consistently assessed, enabling proactive management and timely intervention. Control mechanisms, in turn, allow policies to be enforced and necessary adjustments to be made to maintain system stability and efficiency. Together, these processes form the backbone of a robust management strategy that ensures multicluster environments run smoothly and meet the rigorous demands of modern computing tasks.

Scheduling Policies Across Clusters

Cross-cluster scheduling strategies are fundamental to optimizing the distribution and utilization of resources in a multicluster environment. Potential factors include geographic location, resource availability, and specific vendor constraints. Organizations can ensure that the most appropriate clusters process workloads by adapting scheduling policies to these parameters. This adaptive approach not only improves response times and reduces latency but also helps to meet compliance requirements and utilize resources cost-effectively [3, 67].

2.1.3 Practical Applications and Use Cases

Multi-cluster orchestration offers transformative solutions across various industries by enabling more efficient management of resources and streamlined operations. This section illustrates multi-cluster orchestration's practical applications and significant benefits through two detailed use cases. Each example demonstrates the critical role that sophisticated orchestration plays in not only meeting the unique demands of different sectors but also enhancing the capabilities of organizations to respond dynamically to their operational challenges. From global e-commerce to healthcare data analysis, these use cases provide insights into how multi-cluster orchestration can be strategically implemented to achieve scalability, compliance, and performance optimization.

Global E-Commerce Platform

In the rapidly evolving global e-commerce world, organizations face the challenge of providing customers with a seamless and responsive online shopping experience across different continents. To manage this complexity, leading e-commerce companies should utilize multi-cluster orchestration and separate clusters strategically placed in key regions worldwide. This architecture is critical for optimizing local access speeds

and complying with regional data protection laws, enhancing customer trust and satisfaction.

The orchestration framework implements geo-specific scheduling policies that intelligently route user requests to the nearest data center. This minimizes latency and dynamically adapts to the geographical distribution of user traffic throughout the day. For example, during peak shopping times in Europe, the system automatically redirects European traffic to local clusters, while data centers in less busy time zones can handle night-time processing tasks.

Autoscaling plays a crucial role in coping with fluctuations in demand in the e-commerce sector, especially during sales or promotional events. Each cluster has autoscaling capabilities that automatically adjust computing resources based on real-time traffic data. This ensures that the platform can handle sudden spikes in user activity without performance degradation, ensuring a smooth and efficient shopping experience for all users.

In addition, multi-cluster monitoring is essential to the e-commerce platform's operational strategy. It provides a centralized dashboard that offers real-time insights into the health and performance of clusters across all regions. This allows the IT team to quickly identify and resolve potential downtime or bottlenecks before they impact customers. The monitoring system also helps with proactive maintenance and fine-tuning of resources to ensure that each cluster operates at maximum efficiency and continues to meet the evolving needs of a global customer base.

Healthcare Data Analysis

In the healthcare sector, data analysis and patient privacy are paramount. A healthcare analytics service leverages multi-cluster orchestration to manage sensitive patient data across multiple jurisdictions. This orchestration allows the service to meet various compliance requirements tailored to specific regions, such as the California Consumer Privacy Act (CCPA) [9] in the US and General Data Protection Regulation (GDPR) [18] in Europe, ensuring that patient data is handled securely and in accordance with local laws.

As part of multi-cluster orchestration for healthcare analytics, different clusters can be strategically deployed in different countries to manage sensitive patient data in accordance with local regulations. This approach ensures that data management policies are specifically tailored to the strict requirements of each region.

For example, a cluster in the United States would be configured to comply with the CCPA, which sets the standard for protecting sensitive patient data. This cluster would implement specific security measures, such as data encryption at rest and in transit, secure access controls, and detailed audit capabilities.

In contrast, a cluster based in a European Union country would comply with the GDPR. Data protection is ensured through strict consent protocols, the right to erasure, and data minimization principles. The GDPR also stipulates that data may only be transferred to countries outside the EU if adequate protection is guaranteed.

By utilizing multi-cluster orchestration, healthcare analytics services can dynamically route data processing tasks to the appropriate cluster based on the geographic location of the data source and the specific data regulations of that region. This helps comply with local laws and optimizes the processing and analysis of healthcare data by minimizing latency and maximizing resource usage efficiency.

In addition, the orchestration system can address data sovereignty issues by preventing the cross-border transfer of data unless it is absolutely necessary and legally permissible. The service can also implement region-specific data retention policies and security protocols to further enhance the protection of patient data.

2.2 Multi-Cluster Communication

There are various approaches to establishing successful communication between clusters. This also depends heavily on the circumstances in which the clusters find themselves. If all clusters are located in cloud environments, other techniques may be used compared to the edge environment.

2.2.1 Container Networking

The basic concept of container networking must be explained to illustrate the different communication technologies. When a container is started, a separate Linux network namespace is created for each container. This is separate from the host network and is only connected via a network bridge, creating basic isolation and security. After the container has created the network namespace, the engine calls the selected container network interface (CNI) when initiating a container. The CNI receives a link to the newly created namespace of the container and further information, and depending on the type of CNI and the defined rules, certain network rules are created in the network namespace. Well-known CNIs are Calico or Flannel. CNI is a firmly defined interface that every container engine understands, and consequently, the CNIs can be exchanged.

Sidecar containers [8] are an alternative approach for communication between clusters. These separate containers run alongside an application container to perform isolated peripheral tasks such as logging, proxying, and configuration management. Sidecar containers are created and terminated in synchronization with the application containers they support and have the same lifecycle management as the parent container. By offloading non-functional requirements such as network handling to sidecar

containers, applications can focus on their core tasks without being burdened by these additional requirements. By abstracting the complex network logic from the application code, sidecar containers simplify the development and maintenance of microservices.

2.2.2 Inter-Cluster Networking Solutions

A common approach for service communication is service mesh [36]. Service Mesh works by deploying Envoy proxies as sidecars next to each microservice in a Kubernetes cluster. These proxies intercept all network traffic between microservices and allow the Service Mesh to manage the communication without changing the application code. It provides a control plane for configuring these proxies and manages traffic routing, load balancing, service discovery, security, and observability. The control plane dynamically configures the proxies to enforce policies, collect telemetry data, and secure communications between services with mutual TLS. The best-known provider is Istio [54], which also offers the option of multi-cluster mesh. For this, Istio must be installed on each cluster, and the Istio Deamon must also be able to access the Kubernetes API of other clusters to ensure a pod-to-pod connection between clusters. However, Istio can only be used for Kubernetes clusters and not for alternative tools.

Another approach is manually configuring routing tables and entries to enable cluster communication. With this method, routes must be explicitly defined. Manually setting up routing tables involves specifying each cluster's IP addresses and subnet masks, defining gateway addresses, and creating static routes that specify how packets should traverse the network. This approach allows fine-grained control over network paths and can be tailored to specific infrastructure requirements. However, it is labor-intensive and complex, leading to misconfigurations and potential network issues.

While this method offers flexibility and direct control over network routing, it lacks the automation and dynamic adaptability offered by solutions such as sidecar containers and service meshes. It is often used in simpler or smaller environments where the expense of implementing automated solutions is not justified

There are also overlay mesh options that can be realized with various tools. One of the best-known tools is Submariner [57]. Submariner is an open-source project developed to connect Kubernetes clusters and enable seamless communication between the clusters. It creates an overlay network that allows services in different clusters to communicate as if they were on the same network. Submariner achieves this by setting up encrypted tunnels between clusters using IPsec or WireGuard to ensure secure data transmission. Submariner's network path varies depending on the origin and destination of the IP traffic. In all cases, traffic between two clusters is routed via the configured cable driver between the gateway nodes selected by the leader in each cluster. If the source pod is on a worker node that is not the elected gateway node,

traffic destined for the remote cluster is routed through the submariner VXLAN tunnel (vx-submariner) to the local cluster gateway node. On the gateway node, the traffic is forwarded to the remote cluster via the configured tunnel. Once the traffic reaches the destination gateway node, it is forwarded based on the destination CIDR: via the CNI-programmed network if it is a pod network or via the facility configured via kube-proxy if it is a service network. Submariner is still only available for Kubernetes applications, but tunneling can be used, as with Oakestra's networking component. Oakestra implements a semantic overlay networking system, which supports robust service interactions across private networks without heavy overheads [3].

2.3 Edge Cloud Continuum

The Edge Cloud Continuum (ECC) was initially referenced in 2017 [45]. Subsequent to its introduction, it has been extensively cited across various scholarly articles, often with divergent interpretations or under alternative nomenclatures. This chapter aims to provide a comprehensive overview of the Edge Cloud Continuum, elucidating the diverse facets pertinent to its understanding [30].

Before delving into the specifics of the Edge-to-Cloud Continuum (ECC), we will briefly elucidate the concepts of Edge and Cloud Computing.

Edge Computing is a distributed computing paradigm that processes and stores data at the network's periphery, closer to the data sources such as IoT devices. This approach enables faster response times and reduces bandwidth requirements by minimizing the need to transmit large volumes of data to a central data center. [10].

Cloud Computing is a model that enables ubiquitous, convenient, on-demand access to a shared pool of configurable computing resources that can rapidly be provisioned at any time and from any location via the Internet or a network [44].

The Cloud-to-Edge Continuum represents a seamless spectrum of computing resources and services that extends from centralized cloud data centers to decentralized edge locations. This architecture enables the shifting of data processing and application logic to where it is most effective—whether in the cloud for compute-intensive tasks and global scalability [68] or at the network edge for real-time responsiveness and low latency [38]. The continuum provides a flexible platform that allows for dynamic distribution and scaling of services across different network layers to ensure optimal performance, cost-efficiency, and user experience. By integrating cloud and edge computing, organizations can leverage the massive computational power and storage capacity of the cloud while also capitalizing on the physical proximity to data sources and end-users to meet the demands of modern, data-driven applications [47].

Edge, Fog, Cloud & Cloudlet

These are the basic components of a Cloud-To-Edge Continuum. As shown in table 2.1, each component has its own characteristics and advantages [58].

Component	Characteristics
Cloud node	• High latency • High computing power • Global geographic coverage • Long distance • Single point of failure
Cloudnet node	• Medium latency • Mobility • Medium geographic coverage • Medium distance • Medium computing power
Edge Cloud	• Low latency • Limited geographic coverage • Short distance • Single point of failure • Medium computing power • Specific placement within a network
Fog node	• Low latency • Limited geographic coverage • Short distance • Limited computing power • Specific cyber-physical capabilities

Table 2.1: Basic Components Cloud To Edge Continuum [58]

The push for decentralization is driven by the massive amounts of data produced by IoT devices and the need for immediate, context-aware processing that traditional cloud setups can't always efficiently support. Edge computing frameworks, such as fog computing and cloudlets, are developed to provide reliable, low-latency services to end-users. Specifically, cloudlets bridge the gap between mobile users and distant cloud servers by offering a more dependable connection. In contrast, mobile edge computing (MEC) adapts to the dynamic conditions of mobile networks, focusing on connectivity and contextual awareness of the deployed services.

Yet, adopting these technologies in commercial settings introduces challenges like

ensuring scalability and managing the security risks associated with IoT devices. Enterprises require resilient systems that can seamlessly handle potential network instabilities and data overflows, turning to configurations where edge computing predominately handles the data traffic, supported by cloud services when necessary.

The study in [61] demonstrates the feasibility of conducting data analytics and machine learning across hybrid Edge-to-Cloud infrastructures. However, further development and optimization are required to realize its full potential.

2.3.1 Reference Architectures

This thesis aims to develop a reference architecture for next-generation frameworks. While various approaches to cloud-edge reference architectures exist, the integration and resulting architecture presented in the latter part of this thesis align closely with these established frameworks.

Pillar Name	**Description**
Security	Ensuring the protection of data and resources from unauthorized access and threats.
Scalability	The ability to handle increasing workloads by adding resources.
Open	Utilizing open standards and technologies to enhance compatibility and interoperability.
Autonomy	Systems operating independently with minimal human intervention.
RAS (Reliability, Availability, Serviceability)	Ensuring systems are reliable, available, and easily maintainable.
Agility	The capability to quickly adapt to changes and new requirements.
Hierarchy	Organizing system components in a structured manner for better management and scalability.
Programmability	Enabling systems to be easily programmed and customized to meet specific needs.

Table 2.2: Key Pillars of OpenFog Reference Architecture

One notable example is the OpenFog Reference Architecture, originally designed for fog computing but also applicable to edge cloud architectures. This architecture is underpinned by eight key paradigms detailed in Table 2.2. These paradigms ensure

a robust framework capable of supporting diverse and complex applications. By leveraging these principles, the architecture can meet the demands of modern cloud-edge environments.

The Platform for Universal and Lightweight Cloud-Edge Orchestration (PULCEO) [6] offers a ready-to-use cloud-edge reference architecture. PULCEO decouples optimization from infrastructure, promoting generality, reusability, and comparability of orchestration algorithms. Although this architecture was not implemented in the current work, it presents a detailed conceptual framework. The platform offers a public API for continuous interaction with the infrastructure, enabling efficient application placement and scaling. PULCEO is based on the OpenFog reference architecture, aiming to implement all eight paradigms and further extend it with its own innovative concepts.

SODALITE@RT is an open-source framework designed to address the challenges of dynamic orchestration of IoT applications on heterogeneous cloud-edge infrastructures. It uses TOSCA [37] to describe application and infrastructure topology and behavior in a standardized, portable manner. This enhances application portability and reusability across different infrastructures. Infrastructure-as-Code (IaC) automates resource provisioning and management, utilizing technologies like Ansible for consistent deployments. The meta-orchestrator coordinates various resource orchestrators for deployment and management. Its monitoring system collects metrics and triggers alarms to enable adaptive application management at runtime. The high-level architecture is shown in Figure 2.4 and describes the role of the Meta Orchestrator. [35]

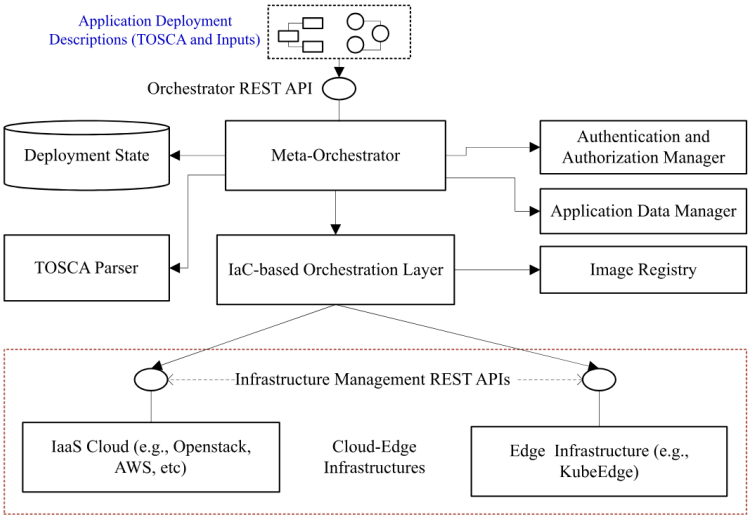


Figure 2.4: High Level Architecture SODALITE@RT [35]

2.3.2 Paradigms

Edge cloud orchestration and communication represent two critical dimensions within the edge cloud continuum (ECC). These aspects are afforded significant emphasis in this study and are therefore addressed in dedicated chapters, specifically 2.3.3 and 2.3.4. Additional paradigms are enumerated in the subsequent section.

Heterogeneity

The Edge-Cloud Continuum (ECC) features highly heterogeneous hardware, spanning from large data centers to small, network-connected sensors and microcontrollers. Any device with basic computational and network capabilities can integrate into ECC. The software on these devices varies widely, including everything from firmware without an Operating System (OS) to full and container operating systems [30]. The ECC also encounters heterogeneity in communication protocols, which can vary significantly across devices and layers within the continuum [19].

Computational Hierarchy

To systematically categorize entities within the Edge-Cloud Continuum (ECC), it is structured into a "computational hierarchy" with three main levels. The top level, the cloud, features extensive resources and high latency, which makes it ideal for heavy computational tasks in large data centers. The intermediate fog or edge level comprises nodes like fog nodes or cloudlets, providing localized computing and storage. The lowest device level directly interfaces with users and includes a range of devices from smartphones to specialized sensors [30].

Virtualization

The use of containerized microservices is emphasized to enhance portability and efficiency in deploying applications across diverse computing environments, from cloud data centers to edge devices. Containers help manage and orchestrate lightweight, independent pieces of software, which is essential for the dynamic nature of edge computing.

2.3.3 Edge Cloud Communication

Edge communication enables direct, efficient interactions between devices located at the edge of the network, significantly improving response times to the user and reducing bandwidth utilization between edge devices. [43] This approach is characterized by

direct device-to-device communication and includes technologies such as multi-access edge computing (MEC) and fog computing. [43] With the introduction of 5G technology and upcoming 6G technology, edge communication will be significantly strengthened as it provides the required speed, capacity, and low latency and supports real-time applications such as autonomous driving and smart city infrastructure. [69]

Cloud communication in data centers relies on a sophisticated network of interconnected servers arranged into clusters. Virtualization technology is crucial here, enabling flexible, scalable environments by dividing physical servers into multiple virtual machines. This setup is controlled through network protocols and APIs, facilitating secure and efficient data transfers. Mobile Cloud Computing (MCC) facilitates the offloading of computational tasks from mobile devices (MDs) to the more robust cloud center (CC), where tasks are processed centrally. Yet, MCC presents inherent challenges, notably the extended propagation distance from the CC to the user, which is disadvantageous for latency-sensitive applications [24, 25]. Software-defined networking (SDN) and Network Functions Virtualization (NFV) enhances network management, promoting dynamic resource distribution and enhanced service provision.

Converging the edge and cloud communication paradigms creates a hybrid model known as cloud-edge communication. This integrated approach is essential for cloud edge clusters where services require communication between edge devices and cloud centers. In this context, mobile edge computing (MEC) has developed. MEC is about shifting the storage and processing of certain data-intensive applications from the centralized cloud to the network's edge in the immediate vicinity of the user.[42].

In Mobile Edge Computing (MEC), offloading compute-intensive tasks to MEC servers for execution in the cloud can lead to significant energy savings for mobile devices and effectively alleviate core network congestion [24, 25]. However, the limited battery life of mobile devices and the increasing need for low-latency applications pose additional challenges in energy consumption and latency when offloading tasks. The proximity of application delivery to users significantly impacts performance and user experience [13], moving applications close to users can significantly improve results.

In addition, using cloudlets offers clear advantages for users' quality of experience (QoE) of users [17, 22]. These studies compare the performance of different applications at different levels of a three-tier hierarchy and show the practical benefits of cloudlets in improving service delivery. The European Telecommunications Standards Institute (ETSI), which originally coined the term Mobile Edge Computing, has renamed it Multi-Access Edge Computing (MEC) to reflect the wider interest of mobile and non-mobile operators in providing services at the network's edge. MEC not only offers low latency but also extends compute and storage capacity directly to end users and Internet of Things (IoT) devices at the edge of the Radio Access Network (RAN), providing key cloud computing services close to the point of data generation and consumption

[59]. By considering communication and computing resources, optimal solutions were designed to reduce energy cost and latency [12]. These data may be transmitted for storage and processing on conventional (centralized) clouds, even though the majority of data can be pre-processed at the network edge [11]

Network Resilience

Network resilience is critical to cloud-to-edge communication, ensuring reliable connectivity and service continuity in diverse and often challenging operational environments. Several factors influence the resilience of such networks:

Heterogeneity Cloud-edge networks are inherently heterogeneous, comprising many devices, from high-capacity cloud servers to resource-constrained edge devices. This diversity necessitates robust protocols and adaptive strategies to manage communication effectively across various hardware and software platforms [4]. The heterogeneity of devices in edge networks can lead to challenges in balancing and scheduling, necessitating advanced solutions that adaptively distribute traffic across servers to mitigate disparities and enhance overall system performance [72].

Bandwith In edge environments, communication bandwidth is often limited, leading to performance impairments. [41]

Scalability As demand fluctuates and the number of edge devices grows, the network must dynamically scale to accommodate changes without compromising performance or security. [41]

Security Given the distributed nature of cloud edge architectures, securing data in transit and at rest, protecting against breaches, and ensuring data privacy is paramount. Implementing robust encryption, authentication protocols, and continuous monitoring can help protect sensitive data across distributed nodes. [62]

Interoperability Ensuring interoperability through standardized protocols and interfaces is crucial in cloud-edge systems, given the many devices and platforms involved. This facilitates seamless communication and integration across the diverse components of the network [63].

Mobility Ensuring service mobility poses a major challenge to network resilience, as maintaining optimal end-to-end session connectivity through-

out service usage is difficult, especially for mobile users who frequently change anchor points, e.g., from one edge node to another. Distributed mobility management (DMM) provides a solution by managing user mobility and facilitating the migration of MEC services to edge nodes near mobile users. [66]

2.3.4 Edge Cloud Orchestration

Edge Cloud Orchestration provides a solution for optimizing computing tasks by dynamically distributing workloads across the edge and cloud resources, thereby improving execution performance. An orchestrator is a software component determining applications' optimal placement and scheduling.

A cloud edge orchestrator can schedule tasks on locally available edge resources or forward them to a cloud system based on resource properties and security permissions. This process can include aggregation of tasks prior to routing to the cloud or disaggregation of tasks prior to distribution to edge resources. An orchestrator at the edge is recommended to support the assignment of tasks to edge and/or cloud resources. resources. In addition, a further fault tolerance can be built if edge applications are replicated more frequently by the edge orchestrator. [53]. It is emphasized that such an orchestrator demands a more detailed understanding of application behavior and support for federation, given the high likelihood that edge environments will span multiple providers. [65]. The study presented in [46] examines how applications and their specific service characteristics impact resource allocation in cloud environments.

In the study [66], three primary limitations and tasks for the Edge Orchestrator are identified. Firstly, resource allocation requires ensuring that sufficient resources are available during scheduling. Secondly, service placement involves determining the optimal locations and quantities for deploying applications. Thirdly, edge selection identifies which edge devices will most likely accommodate the load. Additionally, reliability is a critical concern due to the numerous potential sources of error arising from network stability issues.

2.4 Existing Orchestration Tools, Frameworks, and libraries

The following section examines a range of tools currently used for multi-cluster orchestration and cloud-to-edge orchestration. It will focus on dissecting these tools' architecture and core functionalities, highlighting some of the most recognized or promising options available today. This review will provide insights into each selected tool's technical foundations and operational capabilities.

2.4.1 Multi-Cluster Orchestration

This section discusses current multi-cluster orchestration tools and explains their architectures.

KubeFed

KubeFed [34], short for Kubernetes Federation, is designed to manage multiple Kubernetes clusters. The project aims to synchronize resources across these clusters, enabling managing applications to be deployed across different environments from a single control point. It extends the capabilities of individual Kubernetes clusters by providing mechanisms for coordinated configuration, deployment, and management of applications on a global scale across clusters.

The architecture of KubeFed includes several key components: the Federation API server, which acts as the interface for operations and configuration management; the Federation controller manager, which orchestrates sync across clusters; and a cluster registry that lists the clusters part of the federation. This will not be discussed in detail because the project has already been archived and is no longer being developed.

Despite its potential, the decision to consider archiving KubeFed was driven by several critical factors. First, the project was deemed too broad and not aligned with the more focused projects preferred by the Kubernetes Special Interest Group (SIG). This broad scope made it difficult for KubeFed to attract sustained contributions and maintain a clear development roadmap. There was a significant drop in active contributions, with the project struggling to advance beyond its alpha phase despite plans for a beta release. [33].

Karmada

Karmada [26], short for Kubernetes Armada, is an advanced management system that orchestrates cloud-native applications across various clusters and cloud environments without requiring modifications to the applications. Developed under the auspices of the Cloud Native Computing Foundation (CNCF), Karmada enhances the deployment and management of applications through features like high availability, centralized multi-cloud management, failure recovery, and efficient traffic scheduling. It is designed to interact seamlessly with Kubernetes-native APIs, supporting advanced scheduling capabilities. Furthermore, it offers a variety of options, such as a multi-cluster ingress to route external traffic to the clusters or inter-cluster communication, wherein Kubernetes services can be exported and imported between clusters.

Karmada integrates components familiar to Kubernetes users, such as an API server, a scheduler, and etcd, as shown in Figure 2.5. It provides flexible connection options to

the host cluster via a push principle, where no additional components are needed in new clusters, or a pull principle, where a Karmada agent pulls updates from the host cluster. Moreover, Karmada offers versatile interaction with its API server through tools like kubectl, karmadactl, or a REST API and supports multi-cluster service discovery, enabling clusters to communicate independently of their source cluster. Karmada employs numerous Custom Resource Definitions (CRDs) to create specialized objects that facilitate the scheduling of child clusters. Furthermore, it supports cross-cluster failover of applications to maintain continuous service availability even when some clusters experience failures. This makes Karmada a powerful tool for managing complex Kubernetes environments across various deployment scenarios. To use Karmada, a hub cluster is required where all root components are deployed.

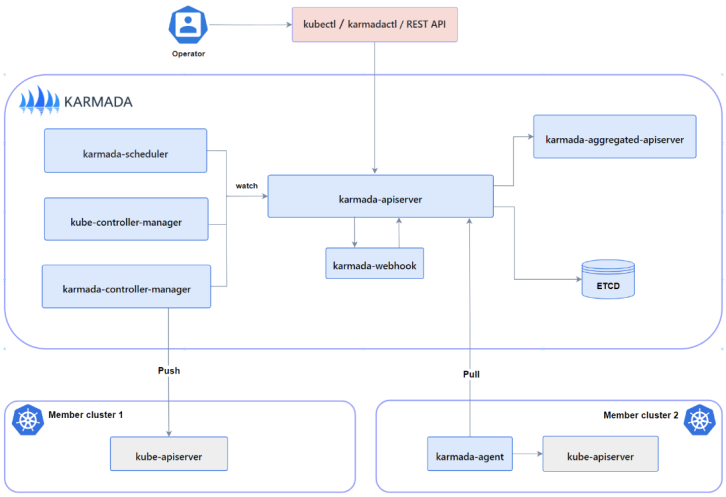


Figure 2.5: High Level Architecture Karamada [26]

KubeAdmiral

KubeAdmiral, [31], a next-generation multi-cluster orchestration engine developed by ByteDance, is designed to enhance the management of large-scale Kubernetes deployments. KubeAdmiral improves resource management and deployment rates across multiple Kubernetes clusters. This project is still in its developmental phase without official releases, indicating ongoing refinement and testing. The architecture is described in Figure 2.6. KubeAdmiral requires a hub cluster, where all the logic takes place, and the child clusters only receive propagated deployments.

Initially developed in response to the limitations of KubeFed v2, which ByteDance

used to federate resources across different business lines, KubeFed struggled with dynamic resource management and adapting to fluctuations in cluster resources. This was particularly evident with stateful services and batch jobs. In response, ByteDance created KubeAdmiral, building upon the foundational concepts of KubeFed but introducing more advanced scheduling capabilities and supporting a broader range of resource types.

Distinguished by its sophisticated scheduling framework, KubeAdmiral facilitates nuanced cluster selection and workload placement strategies. It supports automatic dependency scheduling and dynamic rescheduling based on real-time conditions. ByteDance plans to further enhance KubeAdmiral's capabilities, especially for stateful and batch computing workloads, to meet the evolving demands of modern cloud-native environments [39].

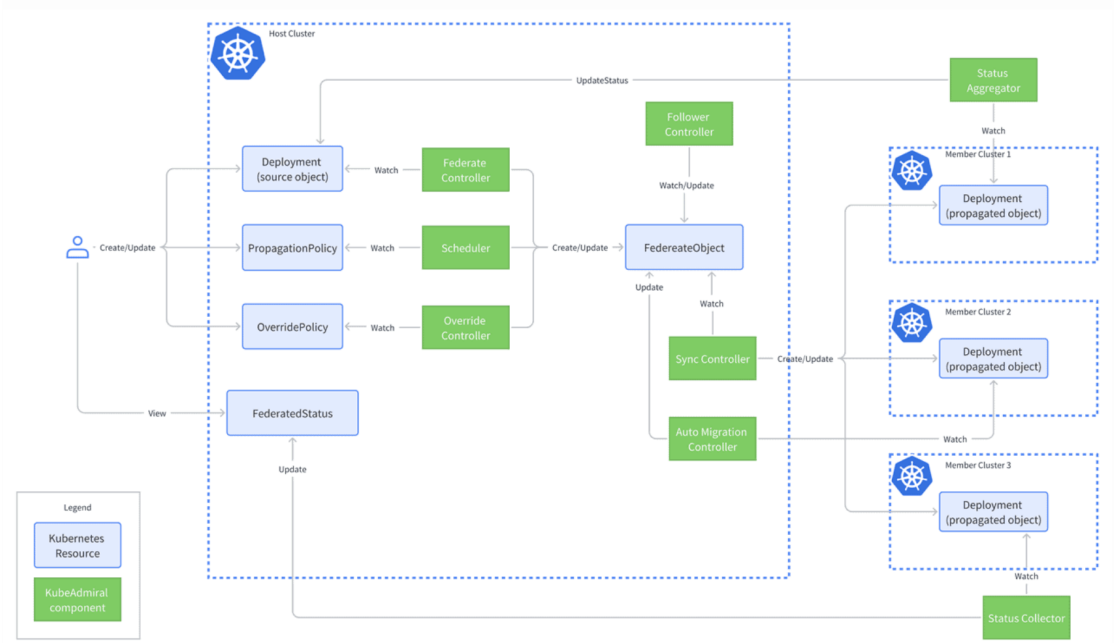


Figure 2.6: High Level Architecture KubeAdmiral [31]

Open Cluster Management

Open Cluster Management (OCM) [51] is an innovative platform for managing multiple Kubernetes clusters across different environments, ensuring efficient orchestration and management. It evolves from earlier Kubernetes federation systems, embracing a more modular and extensible "hub-agent" architecture that separates computation and

execution processes.

The system enables cluster registration, work distribution, and robust policy management, allowing clusters to be vendor-neutral. Furthermore, OCM emphasizes security and modularity in its operations, allowing for extensive customization and integration according to user or organizational requirements. It supports a broad range of operations, from simple workload deployment to sophisticated multi-cluster scheduling, underpinned by a community-driven approach to continuous enhancement and support.

The OCM system utilizes a "hub-spoke" architecture, as shown in figure 2.7, designed to efficiently manage multiple Kubernetes clusters. This architecture is divided into two main elements: the Hub Cluster, which hosts the control plane and issues management commands, and the Klusterlet, installed on each managed cluster to execute these commands. This design allows the Hub to manage numerous clusters without direct interaction. This separation of computation and execution enhances scalability, improves fault tolerance, and allows managed clusters to function autonomously, even if the hub is offline, thus ensuring continuous operations across diverse environments.

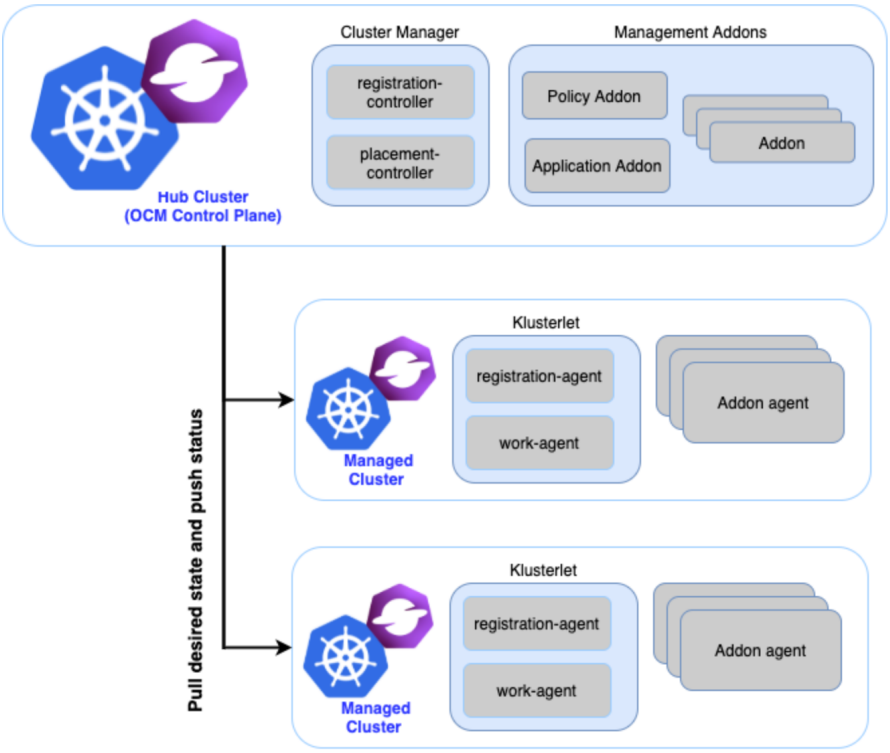


Figure 2.7: High Level Architecture OCM [51]

2.4.2 Cloud-To-Edge Orchestration

This section discusses current edge orchestration tools and explains their architectures.

k3s

K3s is a lightweight Kubernetes distribution designed to run in resource-constrained environments. Developed by Rancher Labs, now part of SUSE, K3s simplifies the Kubernetes installation and maintenance by offering a single binary of less than 100 MB that integrates most necessary components. This binary includes the Kubernetes core components like the API server, scheduler, and controller-manager and essential tools such as Containers, Flannel, and CoreDNS, streamlining the setup and reducing the system's footprint.

K3s distinguishes itself from standard Kubernetes in several ways. It is specifically tailored to simplify the setup and operation of Kubernetes, removing seldom-used features to reduce its size and complexity. Features omitted include legacy, alpha, and non-default Kubernetes features, in-tree storage drivers, and specific cloud provider integrations, typically not required in edge and IoT deployments. However, these components can be added if needed.

The architecture of K3s, shown in figure 2.8 differentiates between server and agent nodes. Server nodes, which host the k3s server service, are responsible for the cluster's control plane and datastore components. In contrast, agent nodes running the k3s agent service focus solely on running workloads in the edge environment. Both types of nodes utilize essential services like kubelet and container runtime. The architecture also integrates Flannel for network communication and Traefik as an ingress controller, simplifying external service exposure and automating HTTPS configurations through Let's Encrypt.

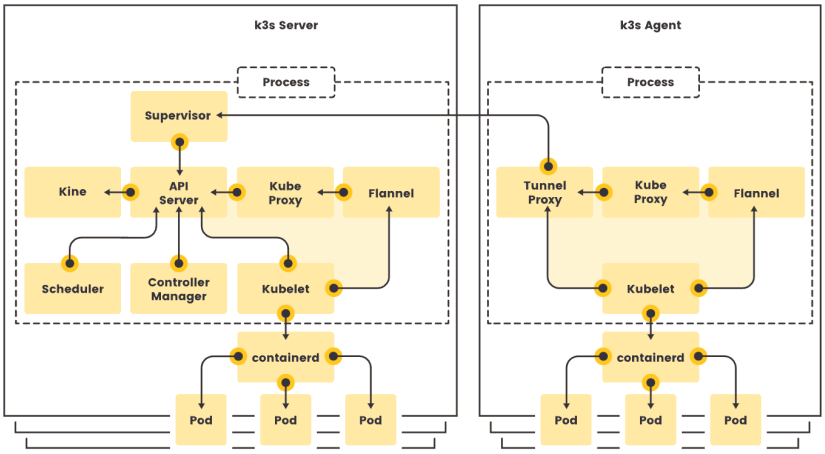


Figure 2.8: High Level Architecture K3s [55]

However, K3s maintains some of the conventional Kubernetes networking challenges. It requires a VPN to manage traffic across diverse networks, reflecting a compromise on network simplicity to maintain broad connectivity. Although K3s are designed for high availability with minimal resource impact, they follow a single-cluster model, which may lack the necessary flexibility for dynamic traffic management, which is crucial in edge computing scenarios. This setup highlights the balance K3s strikes between operational simplicity and the inherent complexities of robust network management in distributed environments [3]

KubeEdge

KubeEdge is a versatile platform designed to extend Kubernetes capabilities to edge devices. It enables better integration and management of IoT devices and edge servers, facilitating seamless cloud-edge computing. The core of KubeEdge is to handle large volumes of devices over a distributed network with reduced latency and enhanced data processing speed close to the data source.

The architecture of KubeEdge, shown in figure 2.9, includes a cloud core and edge nodes. The cloud component, running the control plane, manages the edge nodes that operate the data plane. This structure supports decentralized operation, where edge nodes process data locally, alleviating bandwidth constraints. The architecture integrates with existing Kubernetes ecosystems, allowing for familiar operational models and scalability.

Despite its benefits, KubeEdge has limitations, particularly in scenarios requiring

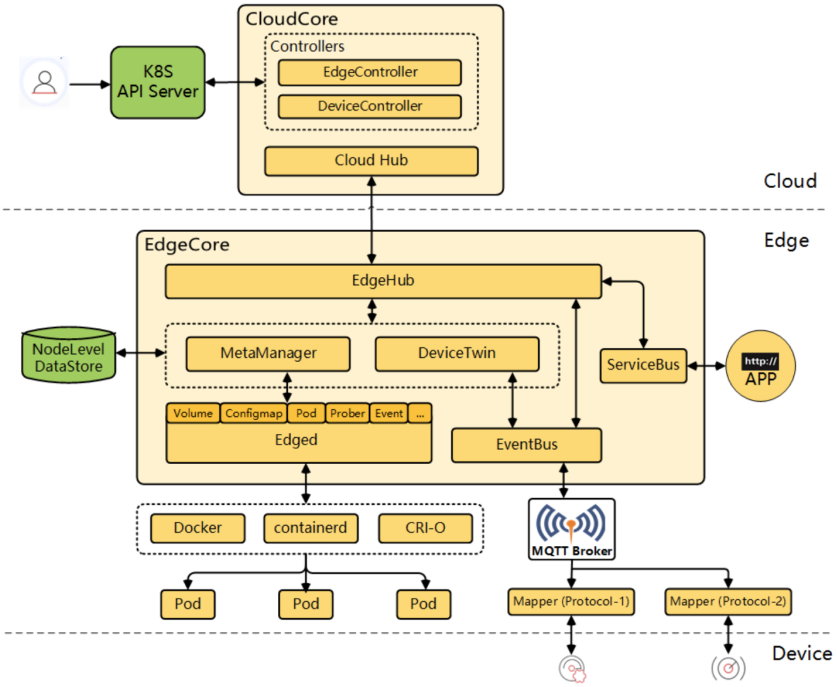


Figure 2.9: High Level Architecture KubeEdge [56]

extensive cloud-based data processing or when high interconnectivity with various cloud services is essential. Additionally, its dependency on a stable network connection for initial setup and some management tasks can pose challenges in environments with intermittent connectivity.

Oakestra

Oakestra is an innovative orchestration framework tailored for edge computing environments [3]. The framework is designed to manage the unique challenges associated with edge computing, such as lower resource capacities and the need for geographical distribution. Oakestra introduces a hierarchical orchestration system that effectively manages resources across distributed edge infrastructures. This setup allows for dynamic response to variations in resource availability and application demands, optimizing CPU and memory usage significantly better than traditional data center-oriented systems.

There are several key features within Oakestra, including federated cluster management and delegated task scheduling. These features enable efficient consolidation and management of multiple infrastructure operators, ensuring smooth operation over

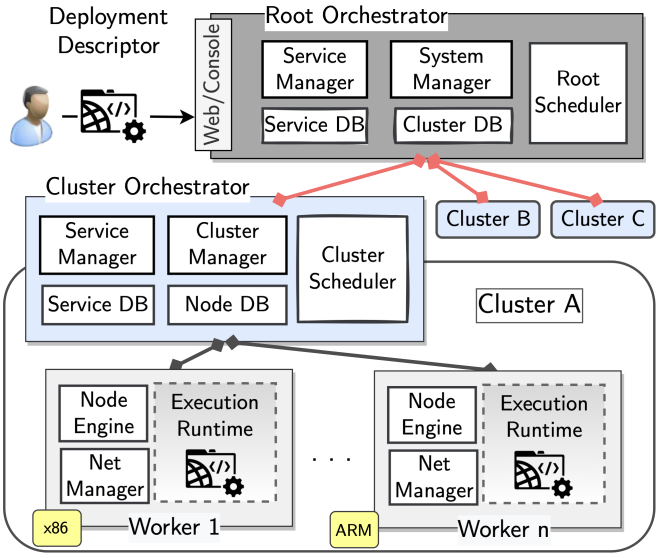


Figure 2.10: High-Level Architecture Oakestra

geographically dispersed resources. Oakestra also implements a semantic overlay networking system, which supports robust service interactions across private networks without heavy overheads. This approach reduces the complexity typically associated with edge computing and enhances the scalability and responsiveness of services deployed in edge environments. Figure 2.10 depicts the architecture of Oakestra. It provides multi-cluster orchestration functionality in an edge environment. The Root component does not need to be a cluster, as can be seen in Karamda or OCM; it is a single component that can be started independently. If it fails, the child clusters can operate autonomously.

The orchestration tool Oakestra, originally designed for edge environments, will be the foundation of my development. By integrating it with Kubernetes, we can establish a seamless cloud-to-edge continuum.

3 Oakestra/Kubernetes Integration

This chapter proposes a design for integrating cloud and edge orchestration systems. The goal is to implement multi-cluster cloud edge applications in heterogeneous infrastructures. This integration aims to leverage the capabilities of Oakestra, a multi-cluster orchestration tool primarily designed for edge environments and developed by TUM, and the current industry standard for cloud orchestration, Kubernetes.

The objective of the integration is to enable cloud-to-edge orchestration and multi-cluster orchestration. With multi-cluster orchestration, many clusters can be synchronized and managed from a single point, which can be beneficial when managing multiple clusters to deliver applications and services worldwide. Edge capabilities are beneficial when the applications' devices have limited resources and network access under conditions that differ significantly from those in data centers or cloud environments. Therefore, both must be used, on the one hand, the cloud capacities and the advantages of the edge. These two new requirements, cloud-to-edge and multi-cluster, define the next generation of orchestration tools, whose approach is presented in the next chapter.

3.1 Requirements

General Requirements

This section outlines the foundational requirements for a Multi-Cluster Cloud-Edge Orchestrator (MCCEO). These requirements are critical for the development of a stable and effective orchestrator. Derived from a systematic evaluation of both challenges and capabilities inherent to cloud-edge environments, this criteria aims to ensure optimal functionality and robustness of the orchestrator across diverse operational contexts. The architectural framework discussed later in this document directly results from these requirements, demonstrating how they inform and shape the overall system design.

R1 *Modularity* The architecture should be designed to support modular components that enable updates and changes without system interruption and facilitate the independent development and testing of individual components. This applies, for example, to the network stack,

which should be designed as an interchangeable plug-in and other system components. Ideally, the modules should follow a standardized interface protocol to optimize integration and interoperability. Finally, modularity should consider cross-platform compatibility to ensure that the components can be used in different cloud and edge infrastructures with minimal customization or at least be used as a template.

R2 *Isolation* Services and applications should be designed to operate in isolated environments to ensure security and operational integrity. This includes supporting multiple independent instances of individually addressable and segregated services.

R3 *Flexibility* Given the heterogeneous nature of edge environments, routing and resource allocation decisions should be flexible, enabling intelligent selection of instances that meet specific performance and Quality of Service (QoS) criteria.

R4 *Resilience* The architecture should support fault tolerance through strategies such as replication and component independence. Essential components such as the root and cluster orchestrator should be highly available to ensure that communication between the various services is not interrupted. If a cluster or the root fails, this should not impact the overall functionality of the distributed services. The clusters should also know which deployments need to be started. This resilience is further strengthened by geographically distributed deployment strategies that can compensate for regional failures and real-time monitoring systems that continuously analyze the system status and react proactively if necessary.

R5 *Scalability* The architecture of the Multi-Cluster Orchestrator should be designed in such a way that it enables seamless scalability. Adding many clusters to the system should be possible without affecting the overall system's performance. Scaling should be possible both at the cluster and worker levels. This also includes the ability to automatically add new clusters or reduce the size of existing ones.

R6 *Automatic Rescheduling* Should a cluster become non-operational over an extended period, rendering the services and applications hosted on it inaccessible, these should be relocated to other clusters that are operational and meet

the deployment criteria. This migration process should be automated, ensuring minimal disruption and maintaining continuity of service.

R7 *Interoperability* The architecture should be designed to ensure seamless interoperability across multiple cloud providers and support different orchestration tools on the market, such as Kubernetes. This capability is crucial for integrating different platforms without compatibility issues. It should provide a unified management interface and standardized communication protocols for efficiently handling operations in different cloud environments. The system should also support common standards and APIs to enable smooth data exchange and integration of functions between different providers.

R8 *Usability* Another requirement is that the orchestration tool is both intuitive and easy to use. It should have a command line interface (CLI) that includes authentication via a configuration file and enables all operations through the CLI tool. This requirement also extends to the provision of comprehensive and detailed documentation.

R9 *Infrastructure Independence* The system should be capable of functioning across a diverse array of devices and networks, accommodating various hardware, software, and networking conditions without dependence on uniform infrastructure settings.

Kubernetes-Oakestra Integration Requirements

In addition to the system design-specific requirements, there are also specialized requirements for integrating existing orchestration tools into the structure of a multi-cluster environment. This integration specifically pertains to using Oakestra and is limited to Kubernetes clusters. These requirements ensure that orchestration tools like Oakestra can effectively communicate and operate within a network of multiple Kubernetes clusters.

R1 *Extensibility* The functional scope of Oakestra is presently confined to delivering services via containers or unikernels. Over time, this scope is anticipated to broaden. Consequently, the integration of Kubernetes should be architected to facilitate an expansion of this functional scope. The design should enable the seamless support of additional workload types as the platform progresses.

R2 *Logical Isolation*

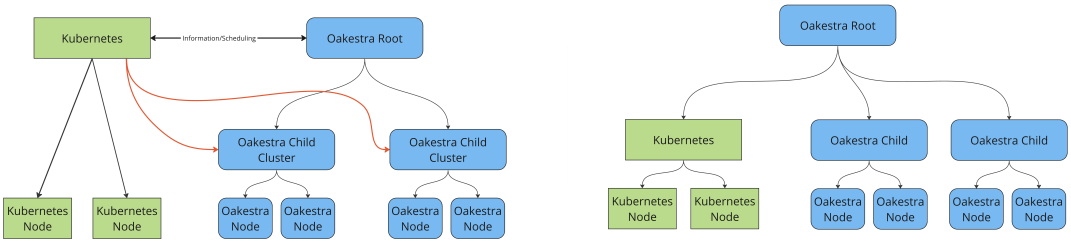
The components designed to integrate Kubernetes with Oakestra should maintain independence from the Oakestra framework. Oakestra must be capable of functioning autonomously, treating a Kubernetes cluster merely as an additional cluster within its infrastructure. No supplementary components should be required to initiate on the Oakestra platform for Kubernetes functionality. This separation allows Oakestra deployment in non-Kubernetes environments, enhancing flexibility and reducing dependencies.

3.2 System Design

The system design for Kubernetes and Oakestra combines a sophisticated architecture with several components, which are discussed and explained in this section. Before we go into the actual components and architecture in more detail, we will first present various approaches for integration.

3.2.1 Integration Approaches

Apart from the fixed paradigm for communication and orchestration of the fixed orchestration tools, Oakestra and Kubernetes, and the requirements already described, integration design has no restrictions. Various integration approaches were therefore developed. The most important of these are presented with the implemented integration in this section.



(a) Option A: Kubernetes as Root Component (b) Option B: Oakestra as Root Component

Figure 3.1: Hierarchical Design Decision

The first step was to decide the hierarchical order in which Kubernetes and Oakestra relate to each other. Kubernetes is a more widely used, well-developed, and comprehensive tool. This supports the idea of integrating Oakestra into Kubernetes as an

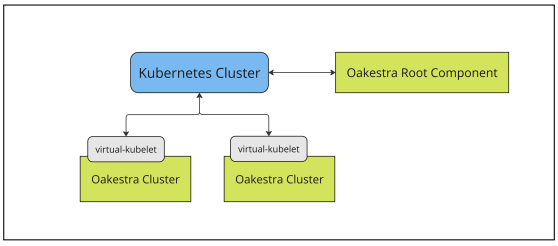
additional component, with Kubernetes remaining the primary interface and its API server being extended with Oakestra functionalities, as can also be seen in Option A in Figure 3.1.

An approach for an integration in which Kubernetes remains at the top of the hierarchy is using virtual Kubelets, Option A1 in Figure 3.2a. With virtual Kubelets, any device can be registered as a Kubernetes node, making it visible to the control plane as a Kubernetes node and used accordingly. When implemented with Oakestra, each Oakestra cluster would be integrated into Kubernetes as a Kubernetes node. Provisioning requests are directed to the Kubernetes API server, and a dedicated Oaekstra Scheduler assigns the deployments to the Oakestra Clusters. This requires the use of an additional Oakestra scheduler. The Kubernetes Control Plane communicates with the Oakestra scheduler. This integration means the complete Oakestra root logic must be integrated into the Kubernetes API.

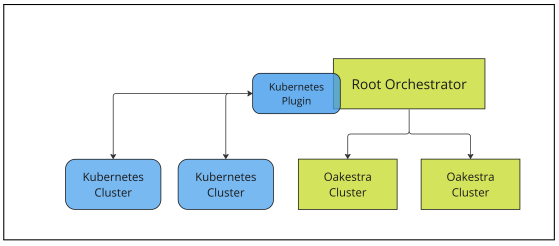
However, this approach would significantly change the Oakestra logic and require a major change in its development roadmap. For this reason, a second fundamental approach was considered: integrating Kubernetes as an additional cluster in Oakestra. See Option B in figure 3.1. If Oakestra Root is at the top hierarchy level, two different approaches have emerged.

The first approach entails modifying the Oakestra root components to enable direct communication with the Kubernetes API, Option B1 in Figure 3.2b. Kubernetes clusters could be registered alongside Oakestra clusters, but they require a separate logic fully embedded in the root. Therefore, no special provisions need to be made in the Kubernetes cluster itself; the root only needs the appropriate credentials and IP address to communicate with the Kubernetes API.

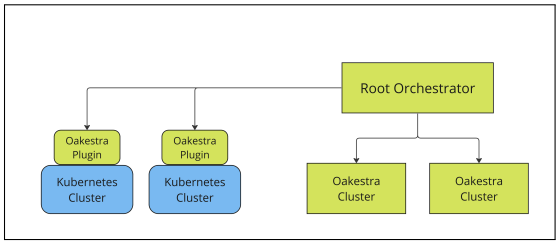
Ultimately, the chosen method was to leave the Oakestra root unchanged. Changes and the complete integration logic would only be required within the Kubernetes cluster. See Option B2 in Figure 3.2c. The advantage of this approach is that no additional logs need to be added to the root component. The main difference to the previous option is that all the integration logic is in Kubernetes, not in Oakestra.



(a) Option A.1: Oakestra as Scheduler



(b) Option B.1: Integration Logic in Oakestra



(c) Option B.2: Integration Logic in Kubernetes

Figure 3.2: Integration Approaches Kubernetes-Oakestra

3.2.2 System Components

An abstract architecture is presented first in figure 3.3, which serves as a blueprint for an Oakestra-Kubernetes integration. Later in the chapter, the specific implementation details are explained. This approach facilitates a structured understanding of the system's theoretical framework and practical application. All components necessary for the integration are deployed across two namespaces: oakestra-system and oakestra-controller-manager.

Oakestra Agent

A key extension of this integration is the development of the Oakestra Agent, which works in a similar way to the Oakestra Cluster Orchestrator. These components are essential for every Kubernetes cluster integrated into Oakestra, enabling communication and coordination with the root component.

One difference between the Oakestra Agent in Kubernetes and the Oakestra Cluster Orchestrator is their communication mechanisms with the worker nodes. In Oakestra, the Oakestra Cluster Orchestrator connects directly to the registered worker nodes via MQTT to assign deployment tasks and establish direct communication channels

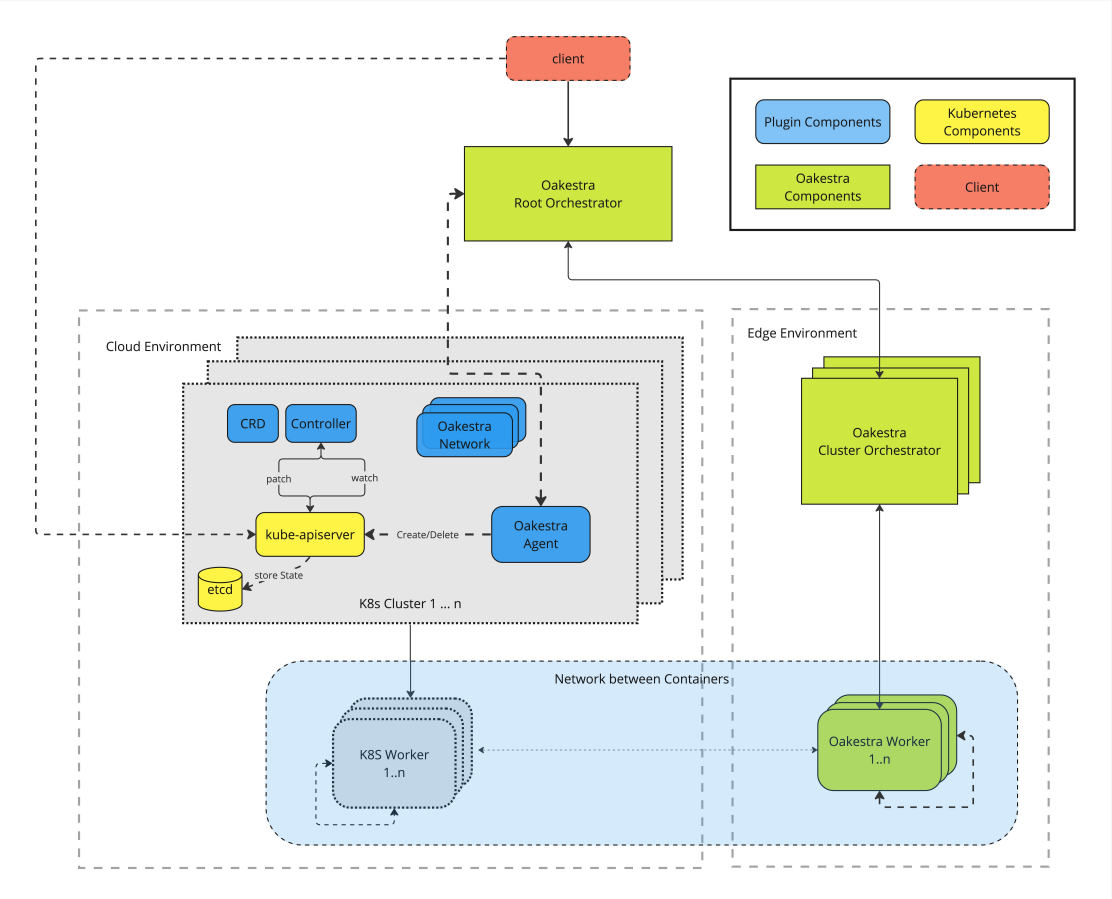


Figure 3.3: High-Level Architecture: Oakestra - Kubernetes Integration

with the worker nodes. The Oakestra agent, on the other hand, has a Kubernetes client that talks to the Kubernetes API and can start Oakestra deployments. The agent does not perform any scheduling but simply forwards the Oakestra requests to the Kubernetes API and starts the corresponding deployments. Detailed descriptions of this orchestration process can be found in chapter 3.3.2.

The root component of Oakestra requires current and recurring information about the utilization of the resources of the child clusters. For this reason, the agent has a background job that regularly transmits updates on current resource utilization to the root components by default at two-second intervals. An additional Kubernetes client is connected to the *Kubernetes Metrics API* server via an interface and queries the current utilization. The Kubernetes cluster must have an activated Metrics server so that the agent can monitor resource utilization and forward updates to the Oakestra

root orchestrator.

Another component within the agent is provided for registering a Kubernetes cluster in Oakestra. This process involves a handshake between the agent and the root component, resulting in the receipt of a cluster ID signaling successful registration. It is important to note that registered Kubernetes clusters are indistinguishable within the root component, with the exception of a 'cluster type' field that indicates whether it is a Kubernetes or Oakestra cluster. However, this distinction is not currently used in the root system but will be considered in future development.

The third important sub-component of the agent is the deployment server, which manages all the important endpoints for the root so that it can accept deployment requests. Each agent hosts a server that primarily provides two endpoints: one for deploying and one for deleting an Oakestra job

A *Kubernetes Deployment Resource* is required to initiate the Oakestra agent in a Kubernetes environment. Specifying environment variables is critical to ensure proper connectivity with the root component. Further explanations of the necessary environment variables can be found in chapter 3.5.2, which explains the use of the Oakestra plugin. The Oakestra agent was developed exclusively in the Go programming language. The container image for the agent can be accessed via the specified repository location, which is explained in more detail in the following sections.

Oakestra Cluster Network Manager

The Oakestra Cluster Network Manager is an additional component required in every Kubernetes cluster. This component serves as the equivalent of the Cluster Service Manager. No changes were required for this component to work in a Kubernetes setup. Only the corresponding environment variables need to be specified to initiate it correctly.

Two additional deployments are required to successfully start the network component: an MQTT server and a MongoDB server. Both components require a variety of Kubernetes components for full functionality. Detailed information on these requirements can be found in the Setup section in chapter 3.5.2.

Oakestra Controller Manager

A central component within the Oakestra plugin is the Oakestra Controller Manager, responsible for orchestrating and using Oakestra jobs. The Controller Manager can manage multiple Custom Resource Definitions (CRDs) and their controllers. The manager monitors the API server to recognize when a new resource has been initiated and ensures seamless integration and functionality within the orchestration process.

The mechanisms of how it works are explained in the chapter on orchestration 3.3.

Kubernetes and Oakestra Components

As already mentioned in connection with the Oakestra agent, it is essential that the Metrics Server is active so that the agent can work at full capacity. In addition to the Metrics Server, all standard Kubernetes components, such as the API Server and Kubelets running on all nodes, must also be operational under normal conditions. No changes are required for these components to function properly within the integrated system.

In addition, the diagram includes several Oakestra components critical to successful integration, including all root components. More information can be found on the Github repository [49].

Oakestra CNI

The Oakestra CNI is responsible for networking between Kubernetes and Oakestra deployments. The CNI is attached to the container at container startup and provides the appropriate routing rules and network interfaces. A more detailed explanation follows in the communication section.

Oakestra Node NetManager

For inter-cluster communication between Kubernetes and Oakestra clusters, each node has its own network component, the Node NetManager. Further details on the network can be found in Chapter 3.4.

A detailed description of the deployments in Kubernetes and all resources that are started in Kubernetes can be found in Appendix 2.

3.3 Orchestration

This section explains which components were used for the orchestration and how the logic works to start Oakestra resources in Kubernetes.

3.3.1 Oakestra Controller Manager

A Controller Manager is employed to orchestrate applications managed and launched by Oakestra. This Controller manages several Kubernetes *Operator Patterns*. Those extend the Kubernetes API to create, configure, and manage instances of complex

stateful applications on behalf of Kubernetes users. This pattern encapsulates domain-specific knowledge about the operational behaviors of these Oakestra applications, automating deployment and management that would require manual intervention.

The current logic and scope of deployable Oakestra resources are ultimately confined to *Oakestra Jobs*. Multiple jobs constitute a microservice, which can be launched as applications. *Oakestra Jobs* may have several instances, each representing the smallest logical unit, depicted either through a container or a Unikernel process. Unikernel is not further pursued in this thesis due to the lack of robust and usable Unikernel support for Kubernetes, although this will likely change. Consequently, instances are the smallest logical unit in Kubernetes, comparable to a *Kubernetes Pod*. Although a *Kubernetes Pod* may contain multiple containers, only one container is initiated in most scenarios.

In Oakestra clusters, a dedicated database contains entries specifying which jobs with many instances are deployed or scheduled in the specific cluster. A similar approach involving a proprietary database could have been implemented in Kubernetes. However, the implementation was executed using Kubernetes' extension capabilities. Additionally, Kubernetes already offers a solution for keeping track of deployed and managed resources through *etcd*, where the current status of each Kubernetes resource is stored. This suggests that this storage should also be utilized for Oakestra resources. To store custom Kubernetes resources and their statuses in etcd, Custom Resource Definitions (CRD) are used, meaning that a CRD is created for every Oakestra resource. This ensures sustainable development, as additional deployable resources from Oakestra can be accommodated by creating more CRDs.

Thus, the CRD OakestraJob is created to the *Oakestra Job*. The definition of an object of this resource can be seen in Figure 3.4. All attributes typically stored in the database of Oakestra clusters are reflected in the YAML configuration. Consequently, an OakestraJob is created for every deployment initiated by Oakestra. In the InstanceList field, the corresponding instances are stored.

Shown below in 3.4 is the YAML File to create an object of this CRD.

Figure 3.4: Example Yaml OakestraJob Custom Resource Object

```yaml
apiVersion: oakestra.oakestra.kubernetes/v1
kind: OakestraJob
metadata:
  name: example-oakestrajob
  namespace: oakestra
spec:
  application_ID: "app"
  application_name: "Example Application"
```

```
  application_namespace: "example-namespace"
  bandwidth_in: 100
  bandwidth_out: 100
  cmd:
    - "/bin/bash"
    - "-c"
    - "echo Hello, World!"
  disk: 10
  environment:
    - "ENV_VAR1=value1"
  image: "example/image:latest"
  instance_list:
    - cluster_ID: "cluster-1"
      cluster_location: "us-west"
      cpu: 2
      disk: 10
      host_IP: "192.168.1.1"
      host_port: "8080"
      instance_number: 1
      last_modified_timestamp: "2023-01-01T00:00:00Z"
      memory: 2048
      status: "Running"
      status_detail: "Instance is running"
      worker_ID: "worker-1"
  job_name: "example-job"
  memory: 2048
  microservice_ID: "service"
  microservice_name: "Example Service"
  microservice_namespace: "example-service-namespace"
  next_instance_progressive_number: 2
  port: "8080"
  state: "Running"
  status: "Active"
  status_detail: "Job is active"
  storage: 20
  vcpus: 2
  vgpus: 0
  virtualization: "docker"
  vtpus: 0
```

However, the CRD created does not initiate any pods that provide the containers and thus make the service accessible. A controller is required for this. The controller constantly checks the status of the OakestraJob resource and reads it when a resource has been created, which it recognizes by the instance list in the CRD having changed. The data in the instance list is then used to create a *Kubernetes Deployment* for each instance, creating a *Kubernetes Pod* with the corresponding container. A *Kubernetes Deployment* was chosen because it allows changes to the *Pod* definition after the *Pod* has been created, the so-called PodTemplate. If changes to the pod are required later, the PodTemplate can be changed, and the *Pod* will automatically restart.
A dedicated controller manages the CRD OakestraJobs. When new Oakestra resources need to be deployed via Kubernetes, a new CRD can be created along with a corresponding controller. This controller transforms the CRD into Kubernetes resources and initiates their deployment. Figure 3.5 illustrates the orchestration process, detailing how deployment proceeds once a Kubernetes cluster is selected within the Oakestra root. All deployments started with Oakestra are initially placed in the oakestra namespace in Kubernetes by default.

3.3.2 Orchestration Flow

To fully comprehend the orchestration flow, refer to Figure 3.5, which may also aid in implementing additional Oakestra resources. Initially, the Oakestra Service Descriptor is used to create and deploy the service. If the *Oakestra Root Scheduler* selects a Kubernetes cluster, an HTTP request containing all necessary information about the *OakestraJob* is sent to the servers of the Oakestra Agent for deployment. Subsequently, one Object of the previously defined CRD is created. Once the Custom Resource (CR) is deployed, the controller detects this change and creates the necessary Kubernetes deployments. Additionally, direct communication with the Kubernetes API Server remains feasible, allowing for fully utilizing the comprehensive functionalities offered by Kubernetes resources. This accessibility ensures that all system capabilities can be fully exploited, enhancing overall operational efficiency. The diagram also illustrates that multiple CRD and controller pairs can be added to the Controller Manager.

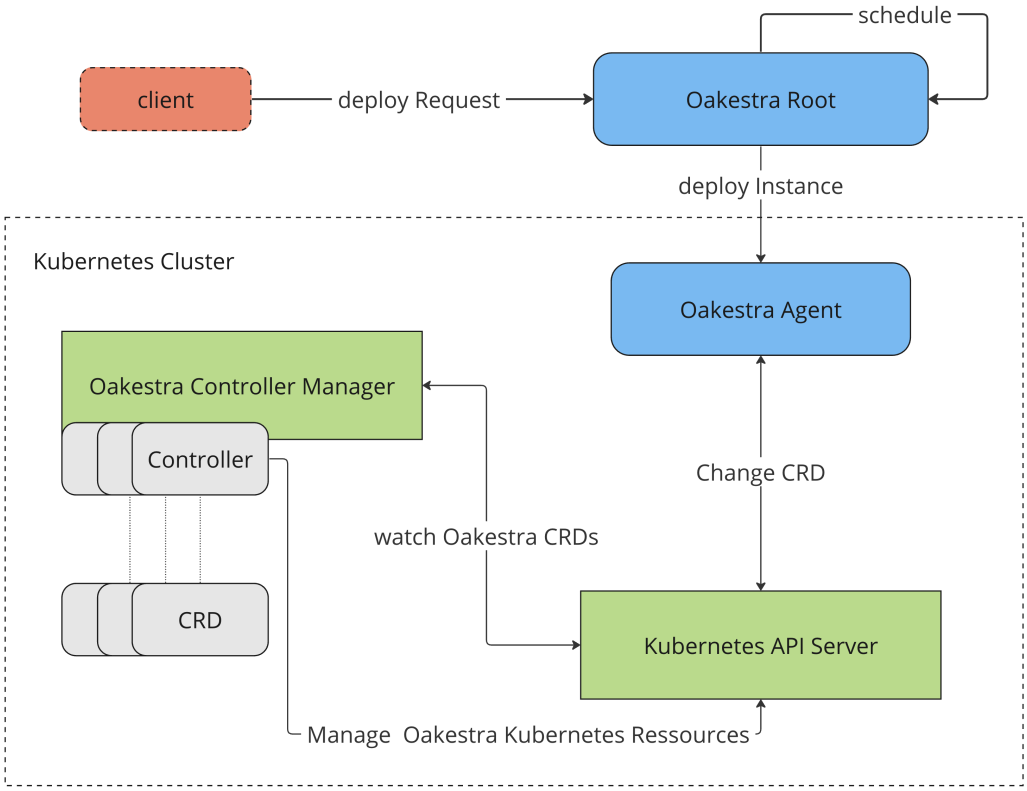


Figure 3.5: Oakestra-Kuberetes Integration: Orchestration Flow

3.3.3 State Handling

Status monitoring plays a critical role in orchestration processes. It informs the control plane whether a redeployment should be initiated or if all functions are operating correctly. Both Oakestra and Kubernetes employ their distinct state-handling mechanisms. Resource usage updates sent from clusters to the root include a summary of the status of the instances. Before this, worker nodes send an update indicating whether the scheduled instances are operational. Based on this information, a decision is made whether to redeploy or not. Kubernetes has its own state-handling system, with controllers ensuring that the status of resources always matches the specified conditions. Updates from the cluster to the root are still transmitted, but with the new status: “Status Handled by Kubernetes”. The API Server must be consulted to gain more precise insights into current issues. Thus, status handling at the cluster level is

resolved through controllers and resources.

3.3.4 Scheduling

No new scheduling methods have been employed. At the root level, the scheduling process is governed by the established Oakestra logic, while within Kubernetes, the standard scheduling logic is utilized. This approach ensures consistency and leverages existing, well-understood mechanisms to manage resource allocation and task distribution across the integrated system.

3.4 Communication

This section explains the communication between Oakestra and Kubernetes resources and inter-cluster communication.

3.4.1 Container Communication

A key objective of the integration was to enable all containers, or pods, operating within Kubernetes to communicate with Oakestra Services, necessitating inter-cluster communication. Oakestra has already implemented a solution in the form of the NetManager.
However, the NetManager interacts with Oakestra's proprietary NodeEngine and is incompatible with the standard Container Network Interfaces (CNI). Chapter 2.2.1 has already described a CNI and its use. For this reason, the Oakestra NetManager was made CNI-compatible to work with the containers in Kubernetes. In order to minimize the changes to the existing Oakestra code, i.e., the NodeManager, the NetManager component at the node level remained unchanged. In addition, an Oakestra CNI component acts as a CNI in Kubernetes and communicates with the NetManager. This means that the Oakestra CNI and the NetManager must be present on every Kubernetes node, realized with a DeamonSet. As both the CNI and the NetManager must have root rights to adjust the routing rules and create a new network interface in the host network of the node, the executables are first transferred onto the file system of the node. Afterwards, the applications start natively on the node.
While the Oakestra NetManager plugin is operational, it lacks DNS capabilities and has other vulnerabilities compared to well-known solutions like Calico and Flannel. Specifically, network isolation has not been fully implemented because all applications share the same network bridge on the node. Additionally, the semantic overlay is generally unnecessary in cloud environments, as dynamic switching of balancing policies is uncommon due to reliable and homogeneous network provisioning. Consequently, the

NetManager is not the ideal standard CNI choice in Kubernetes; instead, more proven solutions should be adopted. Although using the Oakestra CNI is feasible internally within Kubernetes, opting for a different CNI is a better decision.
However, to facilitate communication with Oakestra Services, Kubernetes containers require two CNIs: one to communicate with internal Kubernetes components and another to interact with Oakestra Services. For this purpose, Multus is utilized [21]. Multus CNI is a CNI plugin for Kubernetes that enables attaching multiple network interfaces to pods. This capability enhances network flexibility and functionality, allowing configurations to segregate and manage traffic types across different networks within the same Kubernetes cluster. Consequently, the Linux Network Namespace of the containers is modified by two CNIs. While Calico, or the chosen standard CNI, continues to manage all critical routes, the Oakestra CNI creates an additional bridge with an interface. All Oakestra service IPs in the IPv4 and IPv6 ranges are assigned to the Oakestra-specific interface and routed through the newly created network bridge to the NetManager on the node. From there, the NetManager is responsible for locating the correct address and directing the network packet accordingly. Figure 3.6 clearly illustrates what the network namespace of a container looks like.

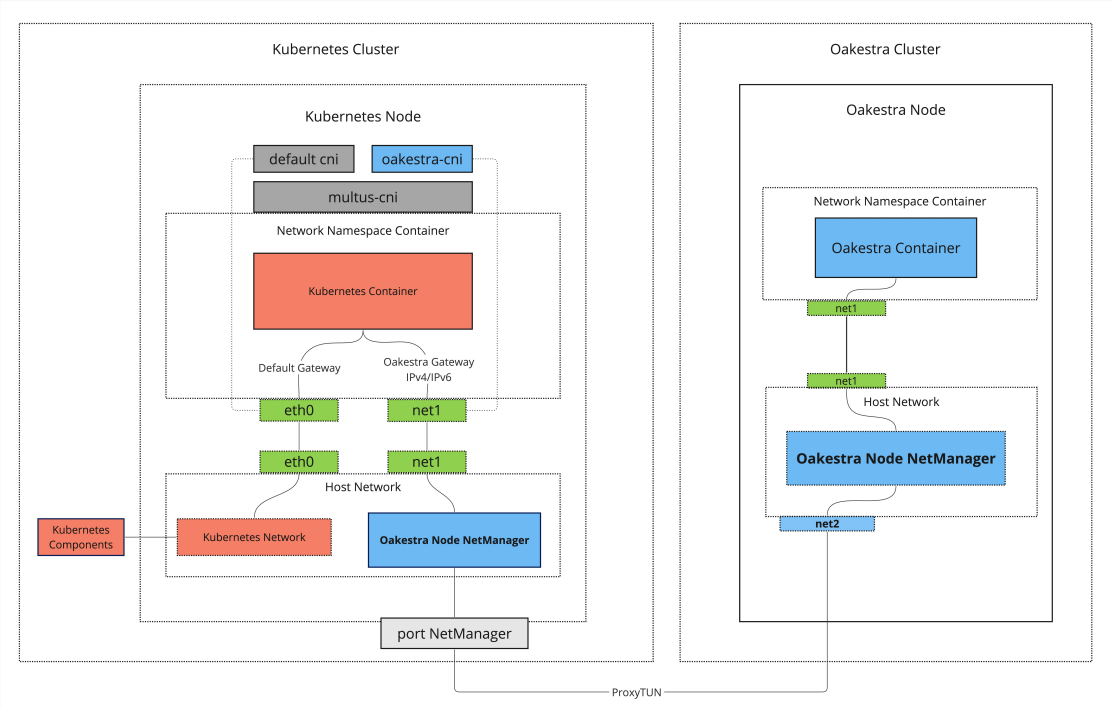


Figure 3.6: Inter Cluster Container Communication

All containers deployed via the Service Descriptor and thus through Oakestra are automatically started with the optional Oakestra CNI. It is important to note that only containers can be initially configured with the CNI at startup. No modifications to the CNIs are possible during runtime; changes can only be made during a potential restart. A specific Pod annotation can be set to enable communication between Kubernetes Pods and Oakestra. In the background, a dedicated webhook monitors all pods for this annotation. If the pod annotation: *["k8s.v1.cni.cncf.io/networks"] = "oakestra-cni"* is present, the controller automatically adds the Oakestra CNI configuration to the selected pod. This approach allows Kubernetes users to decide whether their Kubernetes-managed pods should also communicate with Oakestra Services. Without this additional CNI, Kubernetes pods cannot communicate with Oakestra services. The Oakestra CNI is registered by Multus as an additional CNI, requiring the creation of a NetworkAttachmentDefinition, a CRD generated by Multus. This NetworkAttachmentDefinition must be deployed for each namespace where containers using the Oakestra CNI will be launched with the default Kubernetes-Oakestra integration; the Oakestra CNI is installed only in the oakestra namespace. This is particularly important if Kubernetes Pods, managed by Kubernetes, are to be integrated into the Oakestra network. These Pods must be deployed in a namespace where the Oakestra CNI has been registered.

3.4.2 Communication between Components

For the registration process between the Kubernetes clusters, specifically between the Oakestra Agent and the Root component, a gRPC controller is utilized in which the various messages and their contents are explicitly defined.
An HTTP REST interface facilitates other communications between the Root and the Oakestra Agent. It is crucial that both components are accessible for this communication to be effective.
The Node NetManager and the Oakestra Cluster Service Manager interact through MQTT. This method is chosen because it mirrors the approach taken with other Oakestra clusters; therefore, no changes have been made to this protocol.

3.5 Set Up Kubernetes as Oakestra Cluster

This chapter is intended to facilitate and clarify the integration of Kubernetes into the Oakestra clusters. A variety of different components and tools are required for this integration. All necessary resources and source codes can be found at the following GitHub repository

3.5.1 Prerequisites

Several critical conditions must be met to successfully integrate Kubernetes into the Oakestra framework. Firstly, the Oakestra root must be fully operational in its standard configuration to ensure that foundational functionalities are stable and performing as intended. A standard Kubernetes deployment is required, with all components properly configured to effectively handle containerized applications' orchestration and management. Furthermore, it is essential that all nodes within the network are interconnected and maintain robust network connectivity. This network connectivity is crucial, as it enables seamless communication and data exchange between nodes, vital for the distributed operations typical of both Kubernetes and Oakestra systems.

3.5.2 Set Up Oakestra

When setting up Oakestra, deviating from the default and standard configuration is not needed. The integration implementation is stable with Oakestra version v0.4.301. The predefined settings provided by Oakestra are sufficiently robust to meet the system requirements, ensuring the platform functions effectively without additional adjustments. The complete code for the integration is available in the Oakestra GitHub group, specifically in the *plugin-kubernetes* repository [48].

3.5.3 Set Up Kubernetes

A Kubernetes cluster to be registered with Oakestra currently requires Calico as CNI; it has not yet been tested with other alternatives. In addition, the API Metrics Server must be started and functional.

3.5.4 Install Kubernetes Plugin

All these components must be installed on the Kubernetes cluster.

- Oakestra CNI
- Multus CNI
- Oakestra Cluster Service Manager
- Oakestra Node NetManager
- Oakestra Kubernetes Agent
- Oakestra Controller Manager

The commands for starting the individual components of the integration can be found in the Github repository. The various environment variables are explained in this section. Firstly, those of the Oakestra Agent and two components for the networking are Oakestra NodeNetManager and Oakestra Cluster Service Manager.

With the **Oakestra Cluster Service Manager**, the ClusterIP of the MQTT and MongoDB Kubernetes services must be specified, as seen in table 3.1. These two services must first be created in Kubernetes and then configured. Furthermore, the IP of the Oakestra root and a Kubernetes node are required.

Variable Name	Default Value	Description
MY PORT	10110	Local port which starts server
MQTT BROKER PORT	10003	-
MQTT BROKER URL	*Needs to be set*	ClusterIP of Mosquitto Service
ROOT SERVICE MANAGER URL	*Needs to be set*	IP Oakestra Root Network
ROOT SERVICE MANAGER PORT	10099	Port Oakestra Root Network
SYSTEM MANAGER URL	*Needs to be set*	IP Oakestra Root
SYSTEM MANAGER PORT	10000	Port Oakestra Root
CLUSTER MONGO URL	*Needs to be set*	ClusterIP of MongoDB Service
CLUSTER MONGO PORT	27017	-

Table 3.1: Configuration Variables for Oakestra Cluster Service Manager

The **NodeNetManager** also requires an IP of a Kubernetes node, which must be added in the config. As described in table 3.2. The NodeNetManager needs the Node IP from one of the Kubernetes nodes, regardless of which one, to communicate with the MQTT server, which is published by a NodePort service. This type of service is necessary because the NodeNetManager runs on the host network and is, therefore, not part of the Kubernetes network.

Variable Name	Default Value	Description
NODE PORT	50103	Public node port for Oakestra network
MOSQUITTO SVC SERVICE PORT	30033	-
MOSQUITTO SVC SERVICE HOST	*Needs to be set*	NodePort of one Kubernetes node

Table 3.2: Configuration Variables for Oakestra Node Netmanager

The **Oakestra Agent** requires the cluster's name and location and the Oakestra Root's IP. This information is required for registering with the root component, described in table 3.3. In addition, any IP from a Kubernetes node is required for the Oakestra Agent

to communicate with the Cluster Service Manager, as it is published as a NodePort service.

Variable Name	Default Value	Description
ROOT SYSTEM MANAGER IP	*Needs to be set*	IP Oakestra Root
ROOT SYSTEM MANAGER PORT	10000	Port Oakestra Root
ROOT SERVICE MANAGER PORT	10099	Port Oakestra Network Root
ROOT GRPC PORT	50052	Port GRPC Root
CLUSTER NAME	*Needs to be set*	Name of Cluster
CLUSTER LOCATION	*Needs to be set*	Location of Cluster
MY PORT	10100	Local port which starts server
NODE PORT	30000	Exposed public port to Root, needs to be in range 30000-32767
CLUSTER SERVICE MANAGER PORT	30330	NodePort for Cluster Service Manager
CLUSTER SERVICE MANAGER IP	*Needs to be set*	Node IP of any Kubernetes node

Table 3.3: Configuration Variables for Oakestra Agent

To uninstall Oakestra, all deployments must be deleted. A quick method is to delete the namespaces created for Oakestra during the initial setup. These include *oakestra*, *oakestra-system* and *oakestra-controller-manager*.

4 Evaluation

This chapter aims to highlight the advantages and disadvantages of the design decisions in this project, demonstrating the scenarios where this approach is expected to be sensible and outlining the path for further development of the platform.

4.1 Experimental Set Up

The testbed used for testing and comparing the Kubernetes plugin for Oakestra consists of 8 VMs. The resources are part of iCS (inovex Cloud Services) and, thus, part of the inovex company. The setup also includes 8 Raspberry Pis with different resources.

- 8 VMs Size c5Large
  - 4 CPUs
  - 4 GB of Memory
- **2 Raspberry Pi 5:**
  - Broadcom BCM2712 2.4GHz quad-core 64-bit Arm Cortex-A76 CPU
  - 512KB per-core L2 caches and a 2MB shared L3 cache
- **4 Raspberry Pi 3:**
  - Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
  - 1GB RAM
- **2 Raspberry Pi 2:**
  - A 900MHz quad-core ARM Cortex-A7 CPU
  - 1GB RAM

All virtual machines (VMs) were Linux-based, running Ubuntu 22.04 LTS on an x86 architecture. The Raspberry Pis had the headless Raspberry Pi OS installed on an ARM architecture, corresponding to the recommended version for their hardware. Worker nodes for Oakestra clusters were exclusively deployed on the Raspberry Pis to simulate

restricted edge devices. The more powerful VMs were reserved for all cloud components, including the Cloud Oakestra components. Each experiment was conducted five times over several days, with only one framework running simultaneously.

4.2 Evaluation Results

This chapter elaborates on the evaluation outcomes, detailing the concept and execution of the distinct tests. The tests examine the overhead incurred using the Oakestra tool alongside other metrics like the actual deployment time and the effects on the root component. The tests compare the Oakestra Kubernetes plugin with Vanilla Kubernetes, Karmada, and OCM. It should be noted in advance that the architectures of Karmada and OCM differ fundamentally from those of Oakestra. As a result, direct comparisons are only possible to a limited extent, as Karmada and OCM have more functional similarities than Oakestra. In addition, it should be noted that K-Oakestra refers to a Kubernetes cluster managed by Oakestra in this chapter.

Test Methodology

In the overhead tests in 4.2.1 and 4.2.2, CPU usage, memory usage, and network traffic were measured using Prometheus. CPU utilization was recorded as the total CPU time consumed in seconds over the last minute with the following Prometheus query:

```
sum(rate(container_cpu_usage_seconds_total[1m]))
```

Memory usage was assessed using the working set memory value, retrieved with the query:

```
sum(container_memory_working_set_bytes{pod=~"{SELECTED_PODS}", namespace
    =~"{SELECTED_NAMESPACE}"}) by (pod, namespace)
```

The incoming and outgoing bytes and packets were measured using the following query for networking. Prometheus aggregates the entries of all network interfaces within the container's network namespace, providing a comprehensive view of the network traffic.

```
sum(rate(container_network_transmit_packets_total[1m])) by (container,
    pod, namespace)
```

Additional measurements were conducted following similar methodologies.

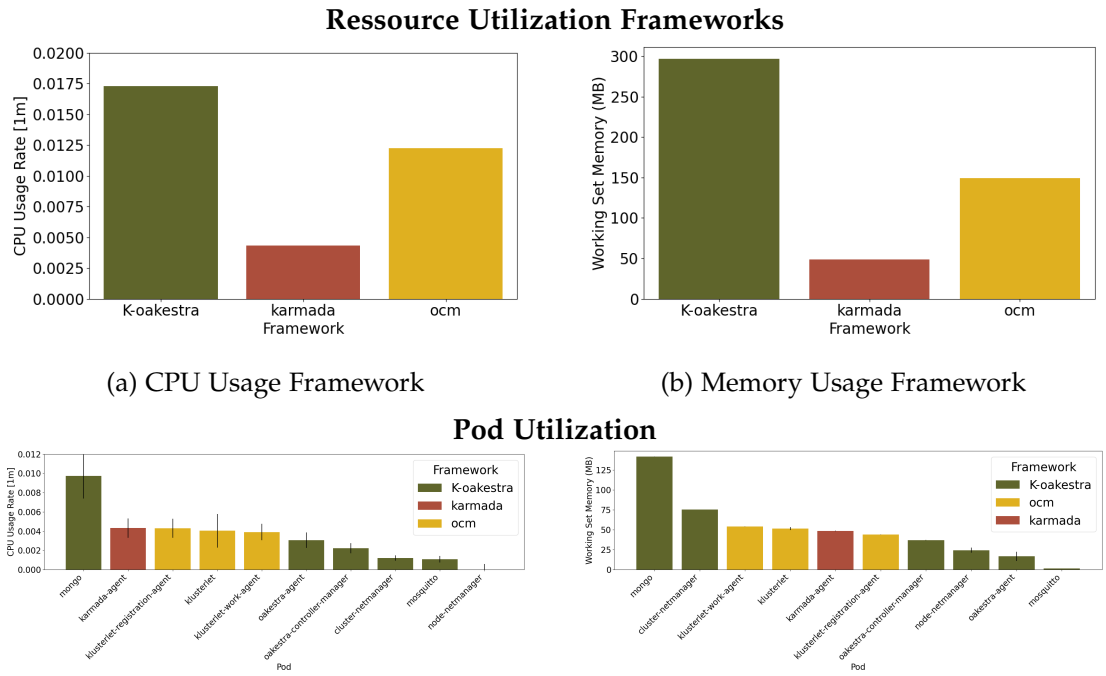
4.2.1 Overhead Plugin

Utilizing the Kubernetes Oakestra plugin initiates several components within Kubernetes, a process mirrored in Karmada and OCM implementations. In all three scenarios, components are activated within Kubernetes to link the respective clusters to the root component. These newly initiated components consume resources. Four identical Kubernetes clusters, each with six nodes, were established to ascertain the precise resource consumption of these components when idle. The first cluster operated without additional components, while the remaining three were each equipped with components for integration with Oakestra, Karmada, and OCM, respectively. Furthermore, Prometheus was installed to facilitate the extraction of specific metrics, particularly those related to containers, nodes, and network metrics. Each measurement was replicated five times, and the median of CPU and Memory measurements and the mean of network traffic were computed to mitigate anomalies.

CPU Usage and Memory Usage

The average CPU utilization across the 6 instances is very similar. All 4 tests show CPU utilization of the 6 nodes between 23,876 and 23,890, corresponding to the total CPU seconds utilized per minute. This value can be divided by the number of nodes to calculate the average utilization per node. This small difference is insignificant, as CPU utilization can fluctuate within this range. Therefore, it can be concluded that none of the three tools requires exceptionally high CPU utilization. To determine more precise differences, we should examine the utilization of the pods.
Figure 4.1a shows the overall median CPU utilization of the plug-in components required to integrate into the multicluster tool. In other words, the median of the individual utilization of the pods added together. It can be seen that K-Oakestra has the highest CPU utilization at 0.017292 seconds of usage per minute, which is relatively low but 4x higher than Karmada with 0.004315 seconds. OCM falls in between, with a value of 0.012236 seconds. Figure 4.1c further illustrates this finding by detailing the CPU used per pod. It can be observed that K-Oakestra deploys more single components, leading to the highest CPU consumption, mainly due to MongoDB, which takes 0.009723 seconds, accounting for 56% of the total value.
The analysis of average memory usage reveals that K-Oakestra consistently consumes the most resources, averaging around 296.39 MB. This is depicted in Figure 4.1b, which illustrates the median working set bytes per pod for each framework. Karmada consumes only 48,56 MB, whereas OCM uses approximately 149,21 MB. When analyzed by individual pods, MongoDB is the most resource-intensive, consuming 141,55 MB. Each Node-Netmanager consumes 24,43 MB; with six instances, the total consumption

is 146,58 MB. Replacing the SQL database could save resources here and thus consume resources similar to OCM. It should be noted that Node-NetManager works across six instances but is only shown once in the overall graphic. This means that the resource consumption of the Node-NetManager still has to be multiplied by the number of nodes in the cluster.



(a) CPU Usage Framework

(b) Memory Usage Framework

(c) CPU Usage Framework Pods

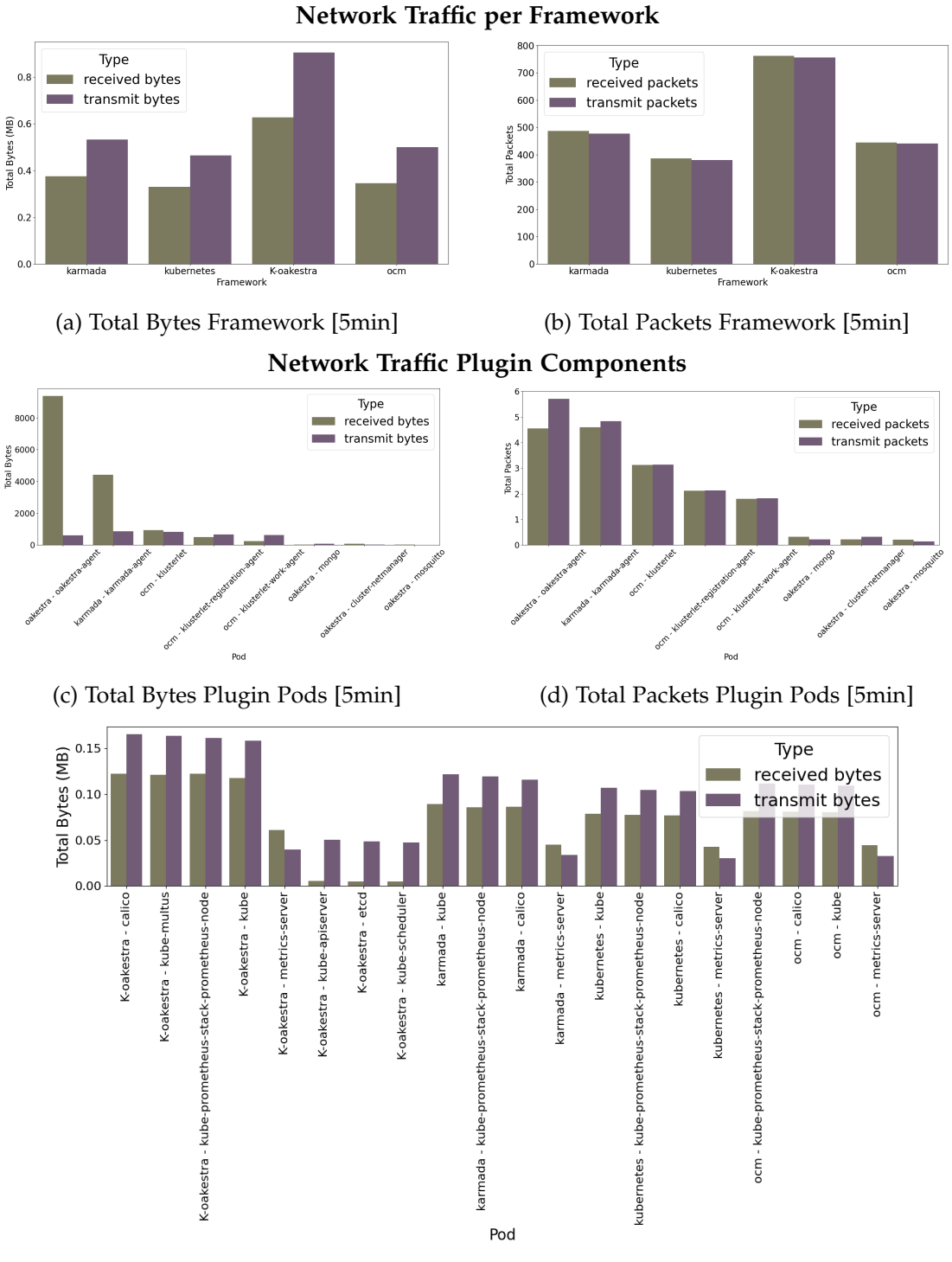
(d) Memory Usage Framework Pods

Figure 4.1: Resource Utilization Overhead Plugin

Network

The network analysis examines the bytes and the number of transmitted packets within the cluster. A lower number indicates better performance, as reduced traffic signifies less network utilization. The measurements were carried out with Prometheus, whereby all bytes and packets of all container network interfaces are added up. The measurements indicate the number of bytes and packets over a 5-minute interval. The evaluations in Figure 4.2 (a) and (b) show that the cluster in which the K-Oakestra plugin was started receives and sends the most bytes and packets. As expected, Kubernetes has the least network traffic. Examining container orchestration frameworks revealed that K-Oakestra exhibited the greatest network traffic volume. The Oakestra framework

received 0.627 MB and transmitted 0.905 MB. Karmada followed with a noticeably lower traffic profile, registering 0.374574 MB received and 0.532 MB transmitted. OCM displayed minimal traffic consumption among the frameworks, recording 0.346 MB received and 0.500 MB transmitted. Kubernetes has the lowest traffic workload, with 0.330 MB received and 0.464 MB transmitted. Figure 4.2c details the components responsible for this discrepancy. This includes the Oakestra Agent, which receives 9386 bytes in 605 packets, mainly triggered by the Metrics Server, which constantly sends information to the Oakestra Agent. However, this is not the only reason K-Oakestra generates the most traffic. Figure 4.2e shows the 20 pods with the highest network traffic among all pods in all four clusters. 8 of these can be attributed to Oakestra. This is partly due to the additional components required to launch the Kubernetes plugin. These components include multus-cni and the cert-manager. The increased communication triggered by these components leads to increased activity in Calico, Prometheus, and the Kube server. As a result, the total network capacity used by K-Oakestra (1,54 MB) significantly exceeds that of Kubernetes (0.795 MB) by 1,9 times. In comparison, Karmada (0.906 MB) exceeds Kubernetes by 1,15 times, and OCM (0.846 MB) exceeds Kubernetes by 1,06 times.



(a) Total Bytes Framework [5min]

(b) Total Packets Framework [5min]

(c) Total Bytes Plugin Pods [5min]

(d) Total Packets Plugin Pods [5min]

(e) Pod Network Traffic across frameworks [20 highest, 5min]

Figure 4.2: Network Traffic Overhead Plugin

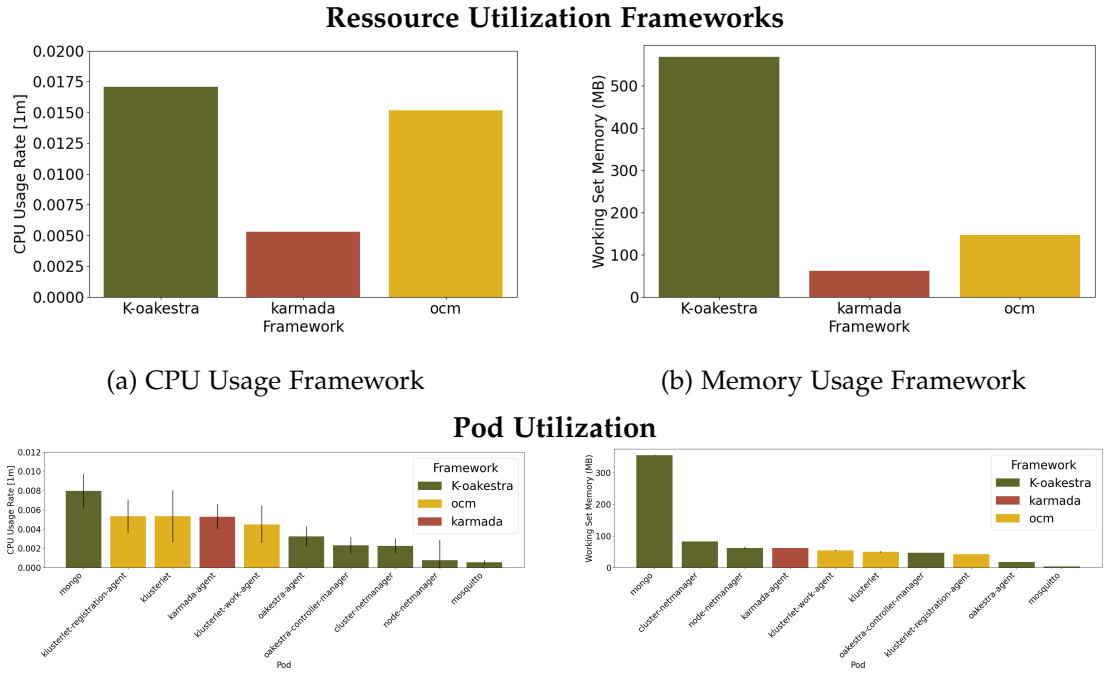
4.2.2 Overhead Management

All deployment requests proceed through this central unit by centralizing management of the Kubernetes cluster via a root component. This test assesses whether administration through a root component incurs additional overhead during execution. For this purpose, a load test was designed, generating 10 clients and 10 servers. The client images were crafted individually, containing a minimal version of a Go HTTPS client and a Go HTTP server. Each client initiates 10 users who send requests to the server concurrently. The server, in turn, executes a computationally intensive task to simulate load. In the Karmada and OCM clusters, the Kubernetes network is utilized. In the Oakestra cluster, both the Kubernetes and the Oakestra network are employed to evaluate the specific overhead introduced by the Oakestra network.
A second test setup was used to evaluate communication overhead between two clusters within the Oakestra network. In this setup, two Kubernetes clusters managed by Oakestra were subjected to the same load test as in single-cluster tests, where the applications can be deployed on two different clusters. This evaluation was conducted exclusively with Oakestra due to the distinct communication mechanisms in Karmada and Open Cluster Management (OCM) and the critical role of the root component in these architectures. Therefore, directly comparing metrics for inter-cluster communication is not feasible.

CPU Usage and Memory Usage

As in the previous test, the median CPU load of the individual pods is added and evaluated. In the Oakestra scenario, load test communication occurs exclusively via the Oakestra network. Later, there is a comparison between the Oakestra network and the Kubernetes network. In Figure 4.3, it can again be seen that K-Oakestra requires the most resources. CPU-wise, K-Oakestra uses 0.0170 seconds of CPU time, which is 3,2 times more than Karmada's 0.0052 seconds and 1,12 times more than OCM's 0.01516 seconds, as shown in subplot (a). Subplot (c) shows the individual CPU Usage time of the plugin pods. As in the OverheadPlugin Test, MongoDB is the largest factor in resource consumption. It takes 0.0079 seconds and is therefore responsible for 46% of the total Oakestra plugin CPU usage. In general, however, there is an increase in resource consumption compared to the overhead plugin test in which no implementations were started. K-Oakestra has CPU Usage decreased by 2%, indicating that the Oakestra Plugins do not increase CPU usage when deploying services. The node-netmanager, rising from zero to 0.000906 seconds per instance, is the only component that shows an increase. With 6 instances, this totals 0.0012 seconds. Karmada's CPU usage increases by 1,22 to 0.005273 seconds, while OCM's CPU usage

increases by 1,23 to 0.015162 seconds. Subplot 4.3b depicts the Memory Usage of the complete framework. In the load test, K-Oakestra's memory usage increased by 1,91 times to 567,72 MB, significantly exceeding both competitors. It is noteworthy that the NodeNetManager was included only once. This time, it consumed 62,12 MB per instance, totaling 372,72 MB for six instances. This usage surpasses that of MongoDB, which used 355,47 MB. Additionally, the Cluster Service Manager's memory usage increased to 82,35 MB. Karmada's memory usage increased by 1,27 times to 61,71 MB. In contrast, OCM experienced a 2% decrease, indicating that its memory usage remained stable. During the load test, K-Oakestra's RAM usage is x9,19 times higher than Karmada's, and compared to OCM, K-Oakestra uses x3,86 times more memory. The main reasons are the NodeNetManager, which has to run on every node, and MongoDB. Together, these account for 73% of the total memory consumption.



(a) CPU Usage Framework

(b) Memory Usage Framework

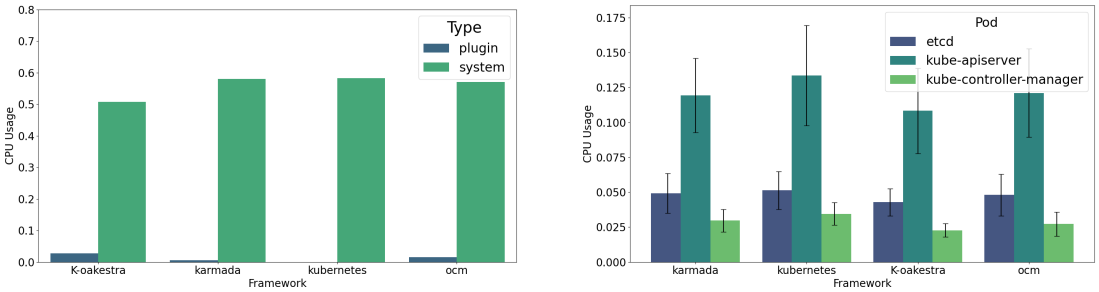
(c) CPU Usage Framework Pods

(d) Memory Usage Framework Pods

Figure 4.3: Resource Utilization Overhead Management

To evaluate the indirect impact on CPU utilization within the system, we examine the namespace pods collectively. The load test deployments are excluded from this analysis. Figure 4.4a displays two bars per framework: one representing the sum of the medians of the pods from all namespaces associated with the plugin and the other representing

the sum of the medians of the pods from namespaces related to the system, including kube-Prometheus-stack and kube-system.
In this analysis, Kubernetes, without any installed plugins, shows a CPU utilization of 0.582643 seconds for system components, closely aligning with Karmada, which registers 0.579607 seconds for system components and 0.005212 seconds for the plugin. Conversely, OCM records 0.570737 seconds for the system and 0.015162 seconds for the plugin. K-Oakestra has the highest CPU utilization for the plugin at 0.026987 seconds but the lowest for system components at 0.507274 seconds, resulting in an 8,9% reduction in system component consumption. This reduction is primarily due to the lower resource consumption of the Kube API server, as depicted in Figure 4.4b, where it requires 0.133678 seconds for Kubernetes and only 0.108241 seconds for K-Oakestra. Additionally, the kube-controller manager shows a reduction, requiring 0.034494 seconds for Kubernetes and only 0.022677 seconds for K-Oakestra. This is mainly attributed to the reduced number of API server requests within the Oakestra network.



(a) Plugin vs. System CPU Usage

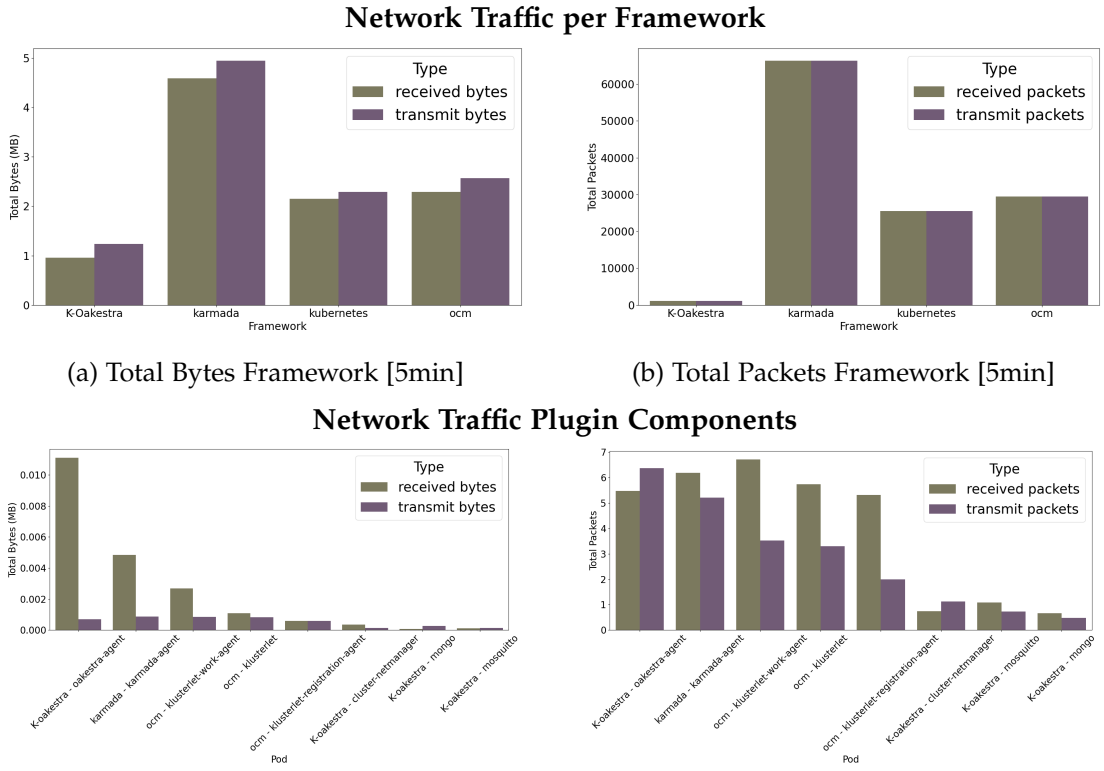
(b) Kubernetes Pods CPU Usage

Figure 4.4: Indirect CPU Usage Impact

The Oakestra plugin facilitates communication through the Oakestra network and the Kubernetes network. To determine the impact of these networks on CPU load, load tests were conducted separately for each network. The median CPU utilization for the system components was 0.549240 seconds, while for the plugin, it was 0.016159 seconds. This suggests the Oakestra Plugin requires fewer CPU resources when utilizing the Kubernetes network. However, the overall CPU usage increases due to the system components.

Network

n the overhead management test, network utilization varied significantly across different frameworks. The test considered the number of bytes and packets transmitted within the cluster. Notably, lower values indicate less network utilization, which is preferable. Figure 4.5a illustrates that Karmada exhibited the highest network utilization, with 4,59 MB received and 4,95 MB sent. This utilization is 2,19 times greater than Kubernetes, which recorded 2,15 MB received and 2,29 MB sent. OCM's network utilization was closely aligned with Kubernetes, with 2,57 MB received and 2,29 MB sent. In contrast, K-Oakestra demonstrated the lowest network utilization, with 0.96 MB received and 1,24 MB sent, amounting to 49,5% of Kubernetes' utilization. This indicates that Oakestra's network sends significantly fewer messages, as evidenced by the packet counts shown in Figure 4.5b.



(a) Total Bytes Framework [5min]

(b) Total Packets Framework [5min]

(c) Total Bytes Plugin Pods [5min]

(d) Total Packets Plugin Pods [5min]

Figure 4.5: Network Traffic Overhead Management

The individual network traffic of the pods and plugins is depicted in Figure 4.5c. The

Oakestra agent again exhibits the highest traffic, increasing by a factor of 1,24 to reach 11644 bytes received, which accounts for 86% of the total network traffic of the plugin's pods. Notably, the components of OCM demonstrate a significant rise; the total traffic of the OCM plugin increases by a factor of 1,83, from 3780 to 6929 bytes. Similarly, Karmada shows a rise in traffic by a factor of 1,13, increasing from 5296 to 5991 bytes. The total traffic for K-Oakestra also increases, with a factor of 1,31, from 10218 bytes to 13487 bytes.

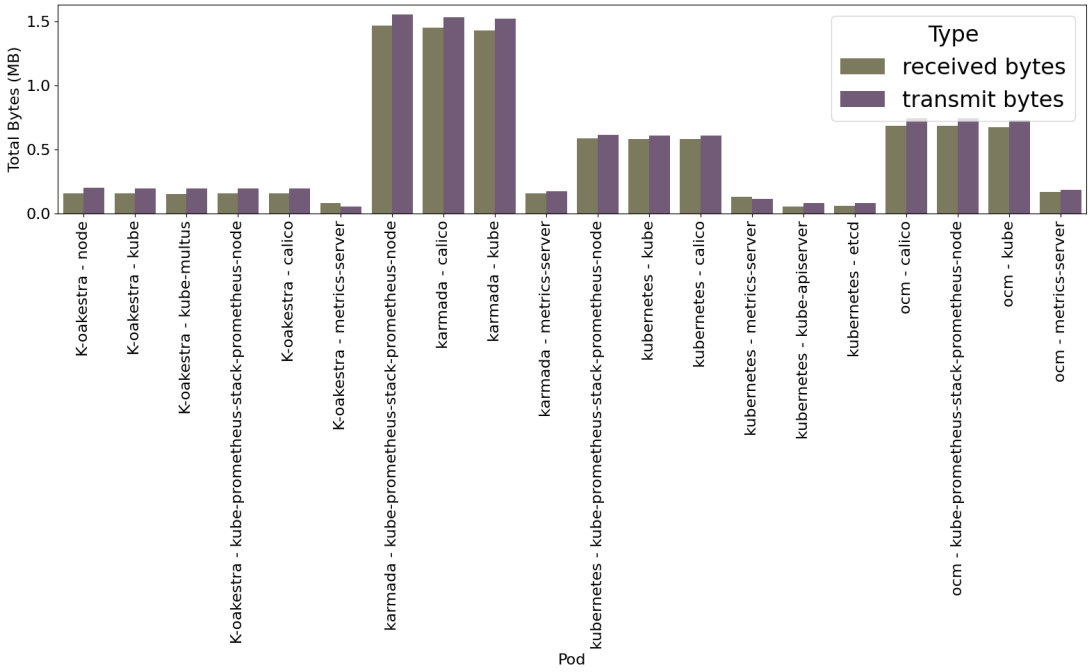


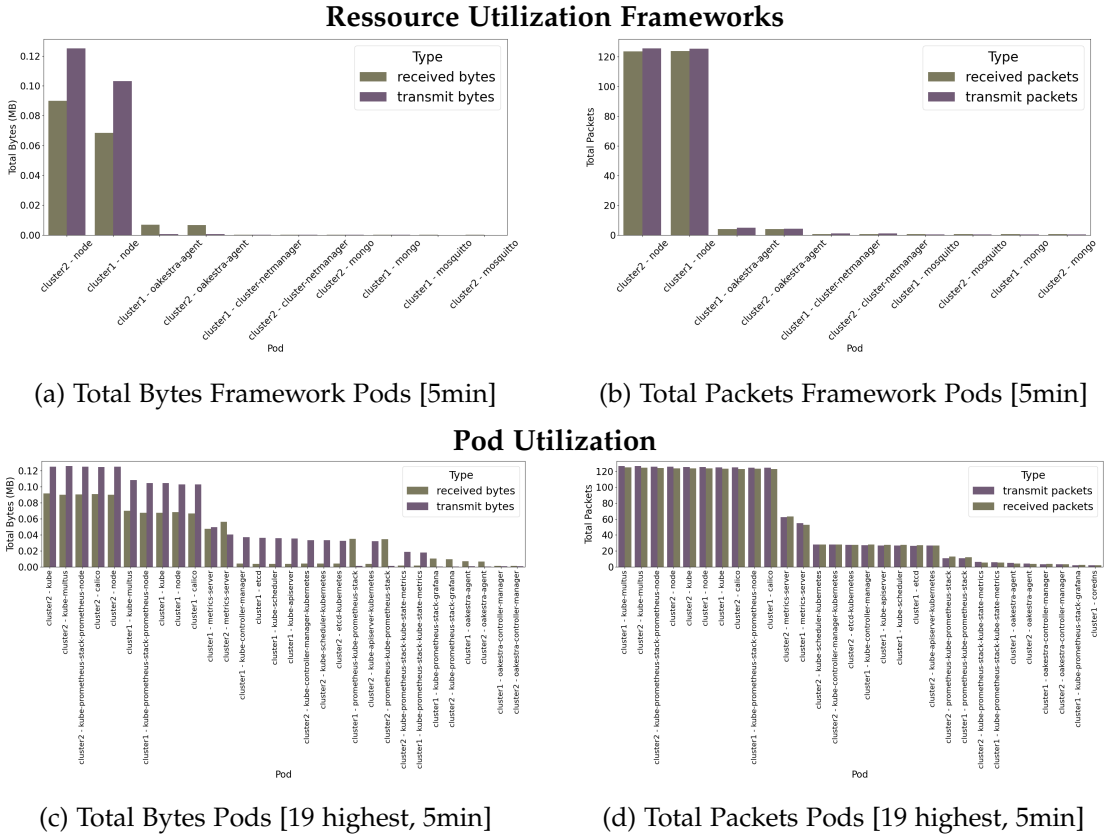
Figure 4.6: Pod Network Traffic across frameworks [20 highest, 5min]

Figure 4.6 illustrates that three specific pods generate significant traffic in all four scenarios: the Calico Nodes, the Prometheus Stack, and the Kube API Server. In the Oakestra scenario, two other pods exhibit high traffic levels: the Node-Netmanager, which operates on each node, and the Kube-Multus pod, which also runs on every node.

Inter Cluster Commmunication

Figure 4.7 displays the test results of communication between two clusters using the Oakestra Network. Although there is no exact equivalent for comparison, the data indicate that the communication patterns are remarkably similar to those observed in

a single cluster. This similarity suggests that multi-cluster communication does not introduce significant overhead.
Due to the very different architecture in OCM and Karmada, there is no direct comparison to the legacy alternatives here; only the results of K-Oakestra are shown and described.



(a) Total Bytes Framework Pods [5min]

(b) Total Packets Framework Pods [5min]

(c) Total Bytes Pods [19 highest, 5min]

(d) Total Packets Pods [19 highest, 5min]

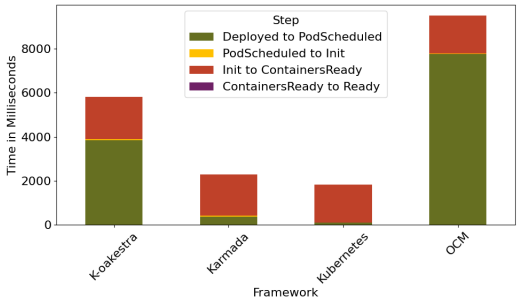
Figure 4.7: Network Traffic Inter-Cluster Communication Oakestra

Utilizing Prometheus, a comprehensive measurement of messages received across all interfaces within the container network namespace was conducted (Fig. 4.7). A significant traffic increase toward the Node Netmanager was observed as anticipated. This phenomenon can be attributed to the higher client-to-server ratio within Cluster 2, resulting in increased requests. The subsequent visualization 4.7c depicts the 19 pods exhibiting the most prominent network activity across both clusters. This data reveals a noticeably lower level of network traffic in Cluster 1, which aligns with the reduced number of deployments deployed within that cluster. Interestingly, the analysis indicates a comparable number of packets transmitted across both clusters.

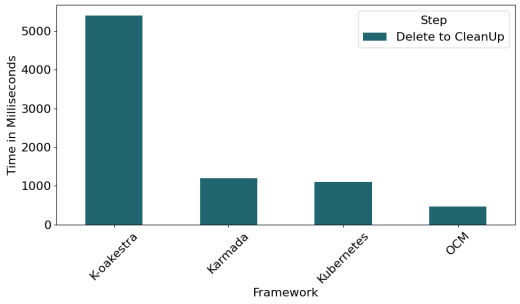
4.2.3 Deployment Time

The duration required to deploy and shut down resources typically holds minor practical importance, yet it is scrutinized in this test. The objective is to determine if management by a root component affects the speed of these operations. For this purpose, 100 small Go binaries are initiated, each designed to start and await only a SIGTERM command. A small HTTP server is also implemented, providing a /ready endpoint. Kubernetes engages this endpoint through the ReadinessProbe to ascertain any potential disparities. Across all four tools, 100 deployments are executed to evaluate this aspect.
Figure 4.8a delineates the time required to initiate a deployment, segmented into four consecutive sub-stages. These stages correspond to status changes read from the Kubernetes Pod. Notably, the "Deployed to Pod Scheduled" phase, highlighted in blue, exhibits the most significant variance among the technologies. This phase spans from the client's request to the pod's scheduling by the Kubernetes cluster deploying the service. It is observed that Vanilla Kubernetes expedites this process, requiring minimal time from request to scheduling due to the absence of intermediary instances. In contrast, the other three technologies experience substantial delays due to additional overhead, with OCM taking 7770 milliseconds as the mean value to trigger a service deployment and K-Oakestra taking 3850 milliseconds, Karmada taking 363,33 milliseconds, whereas vanilla Kubernetes only takes 86,6 milliseconds. It is also worth mentioning that these measurements were taken under the condition of 100 consecutive deployments; the time required for a single deployment would be shorter.
The remaining three phases, internal to Kubernetes, show similar patterns among the technologies. Interestingly, Karmada takes the longest during the second step. The third step, "Init to Container Ready," is nearly identical across all four technologies. The final step is deemed negligible in its duration.
The second figure 4.8b regarding deployment time illustrates the mean duration required to delete one single deployment when 100 deployments are deleted at the same time. K-Oakestra takes the longest to complete this process, with 5400 ms. Whereas OCM takes 473,3 ms, Karmada 1200.0 with ms is very close to Kubernetes with 1110.0 ms. The depicted time span from the deletion request to the point where Kubernetes confirms the absence of the deployment upon the query. Additionally, it is interesting that OCM completes this task the fastest, surpassing even Vanilla Kubernetes in speed.
The third plot 4.8c for DeploymentTime presents the aggregate time necessary to initiate and delete 100 deployments. Consistently, Vanilla Kubernetes is identified as the fastest in both starting and deleting deployments. OCM takes the most time to deploy and delete (91000 ms), although the deletion time of a single deployment was fast, indicating longer periods between requests. Kubernetes deletes all deployments in 25666 ms and

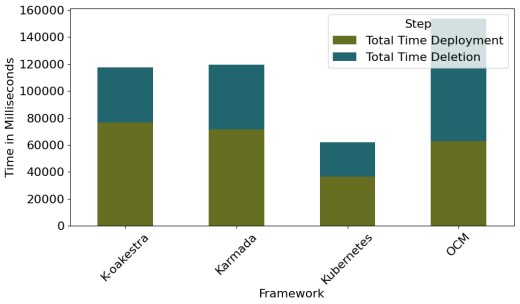
Karmada in 48333 ms. It is noteworthy that K-Oakestra achieves the fastest deletion time of 40,666 milliseconds. This is unexpected, given that individual deployments take the longest time. However, due to Oakestra's architecture, many deployments can be deleted simultaneously, resulting in a significantly quicker overall deletion process. It is also important to mention that deleting a single deployment is faster than the measured time for deleting 100 deployments. K-Oakestra takes 76666 milliseconds to start all deployments, while Karmada takes 71333 milliseconds, exhibiting similar performance. OCM completes the process in 62666 milliseconds, whereas Kubernetes is the fastest, with a deployment time of 36333 milliseconds. OCM takes 2,47 times longer in total deployment deletion time than Kubernetes, while K-Oakestra and Karmada take only 1,9 times longer.



(a) Single Deployment Time 100x Services



(b) Single Clean Up Time 100x Services

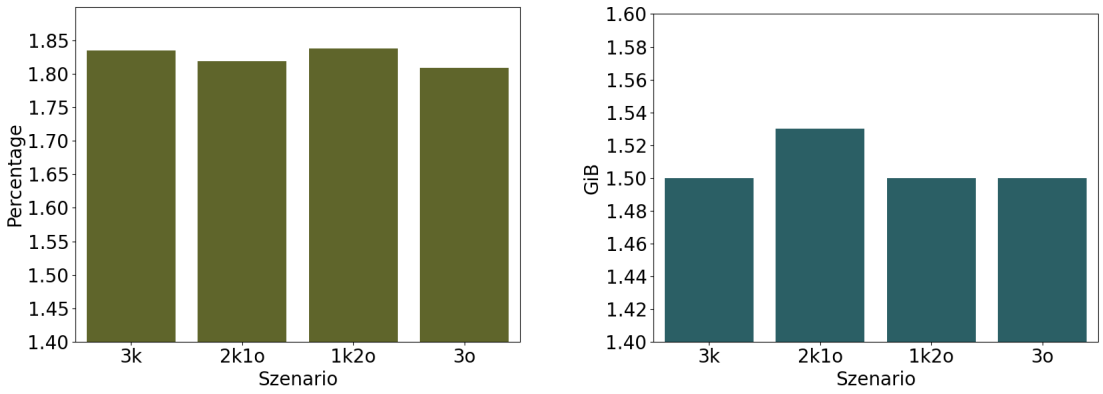


(c) Total Time Deployment and Clean Up 100x Services

Figure 4.8: Deploy Undeploy Orchestration Frameworks

4.2.4 Impact Oakestra Root

Thanks to the integration capabilities with Kubernetes, Kubernetes clusters can be linked to the Oakestra Root. The architecture of this integration is crafted so that there is no distinction for the Oakestra Root, whether it is connected to an Oakestra or a Kubernetes cluster. This test explores and assesses whether Kubernetes clusters influence the root component differently than Oakestra clusters. To facilitate this investigation, 3 Kubernetes and 3 Oakestra clusters were established. At any given time, three clusters were connected to the root. Initially, three Kubernetes clusters were connected, followed by a sequential replacement of one Kubernetes cluster with an Oakestra cluster, and so forth. During the test, a workload is initiated to create a realistic scenario, similar to the one used in overhead management but with significantly fewer deployments.
Figure 4.9a illustrates the CPU utilization of the root component in Oakestra during the login processes of various clusters. It is evident that there are no substantial fluctuations in CPU utilization, suggesting that the deployment of Kubernetes clusters does not significantly impact the root component's performance. In each scenario, the CPU utilization ranges between 1,8194% and 1,8347%, indicating that the differences are only in the hundredths of a percent.



(a) CPU Usage

(b) Memory Usage

Figure 4.9: Resource Usage Impact on Oakestra Root

A minor peak is observed in the memory utilization for scenario 2, Kubernetes 1 Oakestra, as shown in figure 4.9b. All scenarios, except for 2k1o, have a utilization of 1,5 GiB. In the 2k1o scenario, the utilization increased to 1,53 GiB, representing a 2% increase. This anomaly cannot be attributed to the cluster type, suggesting other underlying factors that did not interfere with the plugin. Given that similar impacts

are recorded for both three Kubernetes and three Oakestra clusters, as shown in Figure 4.9b, it can be inferred that the influence remains consistent regardless of whether the clusters are Kubernetes or Oakestra.
Therefore, it can be concluded that Kubernetes clusters do not exert an increased impact on the root component in terms of both memory and CPU utilization, consistent with the intended architecture

5 Conclusion

The primary objective of this work is to integrate a Kubernetes cluster into the Oakestra Multi-Cluster Tool. The complete source code is open source and available in the Oakestra GitHub group, specifically within the *plugin-kubernetes* repository [48]. The outlined integration primarily relies on a Kubernetes cluster that is designed to emulate an Oakestra cluster. It offers identical endpoints and services to those found in a traditional Oakestra setup. Consequently, the root does not require differentiation among clusters since they all provide equivalent interfaces. The implementation utilized standard Kubernetes extension mechanisms, such as custom resource definitions and controllers, constituting the operator pattern. This approach enables the incorporation of any Oakestra resource into the integration using the operator pattern without necessitating additional logic. Moreover, with minor modifications, Oakestra's pre-existing network infrastructure and technology were employed. Specifically, a CNI (Container Network Interface) intermediary layer was introduced between the container and the Oakestra network component, rendering the CNI component compatible. This network component additionally establishes a gateway that routes all IP addresses within the subnet of all service IPs to the network component of Oakestra, thus facilitating connectivity between containers.
The evaluation compares this approach to vanilla Kubernetes and two other multi-cluster Kubernetes orchestrators. Recognizing that each Kubernetes Multi-Cluster Orchestrator adheres to a distinct architectural framework is crucial. These orchestrators utilize a central hub cluster that serves as the root, with significant logic concentrated within the hub cluster rather than distributed across individual clusters, unlike in the Oakestra framework. This architectural difference results in a less-than-ideal comparison between the tools.
The current integration implementation is compatible with the default CNI Calico and the container engine CRI-O [15]. Other container engines, such as Docker, may handle the network namespaces of containers differently, potentially causing issues with the Oakestra CNI. Additionally, it is crucial to ensure that Calico does not use the newly created interfaces for the Oakestra CNI when autodetecting the interface of the container network namespace. Specifically, with Calico, you can specify which interfaces should be overwritten. If an alternative CNI is used instead of Calico, no Oakestra interface must be utilized.

It should also be noted that Oakestra is presently a research project, and, as such, it lacks some of the robustness and functionality found in Kubernetes. However, Oakestra is under continuous development.
The main advantages of integrating the two tools are detailed in the Evaluation Chapter. For instance, using Oakestra CNI significantly reduces network traffic, as illustrated in Figure 4.5. Additionally, the integration of the Kubernetes cluster has minimal impact on Oakestra Root, with both CPU and memory usage remaining identical, as shown in Figure 4.9. Furthermore, Figure 4.8c demonstrates that while deployment by Oakestra increases compared to standalone Kubernetes, its performance remains comparable to two other multi-cluster tools.

5.1 Limitations and Future Work

The current state of the multi-cluster methodology presents several advantages, yet it also encounters certain limitations that require further consideration. Among these limitations is the limited capability of the Oakestra API. To fully leverage the extensive functionalities offered by Kubernetes, the existing Oakestra API cannot be utilized with the service descriptor. Instead, the respective Kubernetes API server must be employed. A potential enhancement involves expanding the API of the Root Orchestrators to ensure that the registration of various clusters also extends the API. Alternatively, developing a robust, general API—which aligns with the current direction but requires further expansion—could be pursued.
Identifying the ServiceIP when a container is initiated and deployed by Kubernetes is currently somewhat challenging. Available options for identifying the ServiceIP include consulting the database or querying the Oakestra API. An additional annotation could be added to the pod containing this value, facilitating the integration of the Oakestra network within a Kubernetes environment.
Another deficiency in the integration is the potential for discrepancies between the Kubernetes cluster database and the Oakestra Root. This issue arises if the Oakestra Root does not delete Oakestra resources within Kubernetes. Additionally, manual deletions of an object from the CRDs can lead to status inconsistencies that result in errors, as these changes are not reflected in the Root. The precise cause of this undesirable behavior warrants further investigation.
The integration of the Kubernetes API into the Oakestra API could be further developed by filtering requests and routing them to a Kubernetes cluster. This represents a potential extension wherein the APIs are integrated within the root component, encompassing all associated clusters. In future developments, this strategy aims to streamline interactions and data flow between the root component and individual clusters.

An additional feature that should be incorporated into the root component of a Multi-Cluster Cloud to Edge Orchestrator is the capability to influence the scheduling decisions within the cluster by utilizing diverse types of clusters. Specifically, some clusters may be deployed in cloud environments, while others operate exclusively in edge locales. This distinction should be utilized as an additional criterion for scheduling tasks. Furthermore, enabling the grouping of clusters based on specific characteristics could enhance the management and operational efficiency of the orchestrator. This would allow for clusters to be grouped depending on the group's attributes, facilitating targeted deployment and resource allocation strategies.
The deployment of Oakestra components on Kubernetes using a Helm chart represents a significant advancement that simplifies the tool's usability and management. This approach means that the user merely needs to retrieve and install the Helm chart, streamlining the setup process and enhancing user experience.
Other possible improvements for the integration include reducing resource consumption. For instance, replacing the MongoDB database could be beneficial. Opting for a database with lower resource consumption or exploring alternative storage methods may offer advantages. Utilizing Kubernetes extensions such as Custom Resource Definitions could be a viable option for storing the state of the Oakestra network.

List of Figures

2.1 Architecture Multi-Cluster: 2-Layer 9
2.2 Architecture Multi-Cluster: 3-Layer 11
2.3 Architecture Multi-Cluster: 4(+)-Layer 11
2.4 High Level Architecture SODALITE@RT [35] 20
2.5 High Level Architecture Karamada [26] 26
2.6 High Level Architecture KubeAdmiral [31] 27
2.7 High Level Architecture OCM [51] 28
2.8 High Level Architecture K3s [55] 30
2.9 High Level Architecture KubeEdge [56] 31
2.10 High-Level Architecture Oakestra 32

3.1 Hierarchical Design Decision 36
3.2 Integration Approaches Kubernetes-Oakestra 38
3.3 High-Level Architecture: Oakestra - Kubernetes Integration 39
3.4 Example Yaml OakestraJob Custom Resource Object 42
3.5 Oakestra-Kuberetes Integration: Orchestration Flow 45
3.6 Inter Cluster Container Communication 47

4.1 Resource Utilization Overhead Plugin 55
4.2 Network Traffic Overhead Plugin 57
4.3 Resource Utilization Overhead Management 59
4.4 Indirect CPU Usage Impact 60
4.5 Network Traffic Overhead Management 61
4.6 Pod Network Traffic across frameworks [20 highest, 5min] 62
4.7 Network Traffic Inter-Cluster Communication Oakestra 63
4.8 Deploy Undeploy Orchestration Frameworks 65
4.9 Resource Usage Impact on Oakestra Root 66

List of Tables

2.1 Basic Components Cloud To Edge Continuum [58] 18
2.2 Key Pillars of OpenFog Reference Architecture 19

3.1 Configuration Variables for Oakestra Cluster Service Manager 50
3.2 Configuration Variables for Oakestra Node Netmanager 50
3.3 Configuration Variables for Oakestra Agent 51

Bibliography

[1] M. Ahuja, N. Sukhavasi, S. Choudhury, K. A. Das, K. Singi, K. Dey, and V. Kaulgud. "MCDA Framework for Edge-Aware Multi-Cloud Hybrid Architecture Recommendation." In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. DOI: 10.1145/3551349.3559501.

[2] J. Arulraj, A. Chatterjee, A. Daglis, A. Dhekne, and U. Ramachandran. "eCloud: A Vision for the Evolution of the Edge-Cloud Continuum." In: *Computer* 54.5 (2021), pp. 24–33. DOI: 10.1109/MC.2021.3059737.

[3] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9.

[4] M. Beck, M. Werner, S. Feld, and T. Schimper. "Mobile Edge Computing: A Taxonomy." In: Jan. 2014.

[5] K. Bilal, S. U. R. Malik, S. U. Khan, and A. Y. Zomaya. "Trends and challenges in cloud datacenters." In: *IEEE Cloud Computing* 1.1 (2014), pp. 10–20. DOI: 10.1109/MCC.2014.26.

[6] S. Bohm and G. Wirtz. "PULCEO - A Novel Architecture for Universal and Lightweight Cloud-Edge Orchestration." In: *2023 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2023, pp. 37–47. DOI: 10.1109/SOSE58276.2023.00011.

[7] D. Bringhenti, R. Sisto, and F. Valenza. "Security automation for multi-cluster orchestration in Kubernetes." In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. 2023, pp. 480–485. DOI: 10.1109/NetSoft57336.2023.10175419.

[8] B. Burns and D. Oppenheimer. "Design Patterns for Container-based Distributed Systems." In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, June 2016.

[9] California Office of the Attorney General. *California Consumer Privacy Act (CCPA)*. https://oag.ca.gov/privacy/ccpa. Accessed on: 15 April 2024. 2024.

[10] K. Cao, Y. Liu, G. Meng, and Q. Sun. "An Overview on Edge Computing Research." In: *IEEE Access* 8 (2020), pp. 85714–85728. DOI: 10.1109/ACCESS.2020.2991734.

[11] H. Chang, A. Hari, S. Mukherjee, and T. Lakshman. "Bringing the cloud to the edge." In: *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE. 2014, pp. 346–351.

[12] X. Chen, L. Jiao, W. Li, and X. Fu. "Efficient multi-user computation offloading for mobile-edge cloud computing." In: *IEEE/ACM transactions on networking* 24.5 (2015), pp. 2795–2808.

[13] S. Clinch, J. Harkes, A. Friday, N. Davies, and M. Satyanarayanan. "How close is close enough? Understanding the role of cloudlets in supporting display appropriation by mobile users." In: *2012 IEEE international conference on pervasive computing and communications*. IEEE. 2012, pp. 122–127.

[14] Cloud Native Computing Foundation. *Multicluster Management - CNCF Technology Radar June 2021*. Accessed: [Insert today's date]. June 2021.

[15] *CRI-O: Lightweight Container Runtime for Kubernetes*. https://cri-o.io/. Accessed: 2024-06-05.

[16] Q. Duan, S. Wang, and N. Ansari. "Convergence of Networking and Cloud/Edge Computing: Status, Challenges, and Opportunities." In: *IEEE Network* 34.6 (2020), pp. 148–155. DOI: 10.1109/MNET.011.2000089.

[17] D. Fesehaye, Y. Gao, K. Nahrstedt, and G. Wang. "Impact of cloudlets on interactive mobile cloud applications." In: *2012 IEEE 16th international enterprise distributed object computing conference*. IEEE. 2012, pp. 123–132.

[18] GDPR Info. *General Data Protection Regulation (GDPR)*. https://gdpr-info.eu. Accessed on: 15 April 2024. 2024.

[19] P. Gkonis, A. Giannopoulos, P. Trakadas, X. Masip-Bruin, and F. D'Andria. "A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues." In: *Future Internet* 15.12 (2023). ISSN: 1999-5903. DOI: 10.3390/fi15120383.

[20] M. Goudarzi, S. Ilager, and R. Buyya. "Cloud Computing and Internet of Things: recent trends and directions." In: *New Frontiers in Cloud Computing and Internet of Things* (2022), pp. 3–29.

[21] K. N. P. W. Group. *Multus-CNI: A CNI meta-plugin for multi-homed pods in Kubernetes*. https://github.com/k8snetworkplumbingwg/multus-cni. Accessed: 2024-06-10. 2024.

[22] K. Ha, P. Pillai, G. Lewis, S. Simanta, S. Clinch, N. Davies, and M. Satyanarayanan. "The impact of mobile multimedia applications on data center consolidation." In: *2013 IEEE international conference on cloud engineering (IC2E)*. IEEE. 2013, pp. 166–176.

[23] J. Hong, T. Dreibholz, J. Schenkel, and J. Hu. "An Overview of Multi-cloud Computing." In: Mar. 2019, pp. 1055–1068. ISBN: 978-3-319-98284-7. DOI: 10.1007/978-3-030-15035-8_103.

[24] W. Huang, Y. Huang, S. He, and L. Yang. "Cloud and edge multicast beamforming for cache-enabled ultra-dense networks." In: *IEEE Transactions on Vehicular Technology* 69.3 (2020), pp. 3481–3485.

[25] L. Jiao, R. Friedman, X. Fu, S. Secci, Z. Smoreda, and H. Tschofenig. "Cloud-based computation offloading for mobile devices: State of the art, challenges and opportunities." In: *2013 Future Network & Mobile Summit* (2013), pp. 1–11.

[26] Karmada Project. *Karmada: Visualization and Management for Kubernetes*. https://github.com/karmada-io/karmada. Accessed on: 15 April 2024. 2024.

[27] J. Ke. *Kubeadm Scripts: Scripts & Kubernetes manifests for Kubeadm Kubernetes cluster setup*. https://github.com/JakobKe/kubeadm-scripts. Accessed: 2024-06-10. 2024.

[28] J. Ke. *Oakestra Kubernetes Plugin Evaluation*. https://github.com/JakobKe/oakestra-kubernetes-plugin-evaluation. Accessed: 2024-06-10. 2024.

[29] T. Kormaník and J. Porubän. "Exploring GitOps: An Approach to Cloud Cluster System Deployment." In: *2023 21st International Conference on Emerging eLearning Technologies and Applications (ICETA)*. 2023, pp. 318–323. DOI: 10.1109/ICETA61311.2023.10344182.

[30] K. Kritikos, P. Skrzypek, and M. Różańska. "Towards an Integration Methodology for Multi-Cloud Application Management Platforms." In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '19 Companion. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 21–28. ISBN: 9781450370448. DOI: 10.1145/3368235.3368833.

[31] KubeAdmiral. *KubeAdmiral: Centralized Management for Kubernetes Clusters*. https://kubeadmiral.io. Accessed on: 15 April 2024. 2024.

[32] Kubernetes. *Kubernetes: Production-Grade Container Orchestration*. https://kubernetes.io. Accessed on: 15 April 2024. 2024.

[33] Kubernetes Community. *KubeFed Github - Kubernetes Federation*. https://github.com/kubernetes-retired/kubefed. Accessed on: 15 April 2024. 2024.

[34] Kubernetes Federation. *KubeFed: Kubernetes Cluster Federation*. https://kubernetes.io/docs/concepts/cluster-administration/federation/. Accessed on: 15 April 2024. 2024.

[35] I. Kumara, P. Mundt, K. Tokmakov, et al. "SODALITE@RT: Orchestrating Applications on Cloud-Edge Infrastructures." In: *Journal of Grid Computing* 19 (2021), p. 29. DOI: 10.1007/s10723-021-09572-0.

[36] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities." In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2019, pp. 122–1225. DOI: 10.1109/SOSE.2019.00026.

[37] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri. "TOSCA Solves Big Problems in the Cloud and Beyond!" In: *IEEE Cloud Computing* (2018), pp. 1–1. DOI: 10.1109/MCC.2018.111121612.

[38] F. Liu, G. Tang, Y. Li, Z. Cai, X. Zhang, and T. Zhou. "A Survey on Edge Computing Systems and Tools." In: *Proceedings of the IEEE* 107.8 (2019), pp. 1537–1562. DOI: 10.1109/JPROC.2019.2920341.

[39] G. Liu. *KubeAdmiral: next-generation multi-cluster orchestration engine based on Kubernetes*. https://www.cncf.io/blog/2023/11/24/kubeadmiral-next-generation-multi-cluster-orchestration-engine-based-on-kubernetes/. Accessed on: 15 April 2024. 2023.

[40] M. Lu, L. Wang, Y. Wang, Z. Fan, Y. Feng, X. Liu, and X. Zhao. "An Orchestration Framework for a Global Multi-Cloud." In: *Proceedings of the 2018 Artificial Intelligence and Cloud Computing Conference*. AICCC '18. Tokyo, Japan: Association for Computing Machinery, 2018, pp. 58–62. ISBN: 9781450366236. DOI: 10.1145/3299819.3299823.

[41] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino. "Scalability and Performance Evaluation of Edge Cloud Systems for Latency Constrained Applications." In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 286–299. DOI: 10.1109/SEC.2018.00028.

[42] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. "A survey on mobile edge computing: The communication perspective." In: 19.4 (2017), pp. 2322–2358.

[43] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief. "A Survey on Mobile Edge Computing: The Communication Perspective." In: *IEEE Communications Surveys & Tutorials* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201.

[44] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. en. 2011-09-28 2011. DOI: https://doi.org/10.6028/NIST.SP.800-145.

[45] D. Milojicic. "The Edge-to-Cloud Continuum." In: *Computer* 53.11 (Nov. 2020), pp. 16–25. ISSN: 1558-0814. DOI: 10.1109/MC.2020.3007297.

[46] M. Mohammed and A. Haqiq. "Dynamic resource allocation for service in mobile cloud computing with Markov modulated arrivals." In: *International Journal of Modeling, Simulation, and Scientific Computing* 12.05 (2021), p. 2150038. DOI: 10.1142/S1793962321500380.

[47] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hästbacka, and D. Taibi. "Cloud Continuum: The Definition." In: *IEEE Access* 10 (2022), pp. 131876–131886. DOI: 10.1109/ACCESS.2022.3229185.

[48] Oakestra. *Oakestra Kubernetes Integration Plugin*. https://github.com/oakestra/plugin-kubernetes. Accessed: 2024-06-05. 2024.

[49] Oakestra. *Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing*. https://github.com/oakestra/oakestra. Accessed: 2024-06-11. 2024.

[50] J. Opara-Martins, R. Sahandi, and F. Tian. "Critical review of vendor lock-in and its impact on adoption of cloud computing." In: *International Conference on Information Society (i-Society 2014)*. 2014, pp. 92–97. DOI: 10.1109/i-Society.2014.7009018.

[51] Open Cluster Management Authors. *Open Cluster Management: A Modular Platform for Kubernetes Multi-Cluster Orchestration*. https://open-cluster-management.io/. Accessed on: 15 April 2024. 2023.

[52] D. Petcu. "Multi-Cloud: expectations and current approaches." In: *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds*. MultiCloud '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 1–6. ISBN: 9781450320504. DOI: 10.1145/2462326.2462328.

[53] I. Petri, O. Rana, A. R. Zamani, and Y. Rezgui. "Edge-Cloud Orchestration: Strategies for Service Placement and Enactment." In: *2019 IEEE International Conference on Cloud Engineering (IC2E)*. 2019, pp. 67–75. DOI: 10.1109/IC2E.2019.00020.

[54] I. Project. *Istio*. https://istio.io/. Accessed: 2024-05-20. 2024.

[55] K. Project. *K3s*. https://k3s.io/. Accessed: 2024-05-05. 2024.

[56] K. Project. *KubeEdge*. https://kubeedge.io/. Accessed: 2024-05-05. 2024.

[57] S. Project. *Submariner*. https://submariner.io/. Accessed: 2024-05-20. 2024.

[58] V. Prokhorenko and M. Ali Babar. "Architectural Resilience in Cloud, Fog and Edge Systems: A Survey." In: *IEEE Access* PP (Feb. 2020), pp. 1–1. DOI: 10.1109/ACCESS.2020.2971007.

[59] J. Qadir, B. Sainz-De-Abajo, A. Khan, B. Garcia-Zapirain, I. De La Torre-Diez, and H. Mahmood. "Towards mobile edge computing: Taxonomy, challenges, applications and future realms." In: *Ieee Access* 8 (2020), pp. 189129–189162.

[60] Redhat. *Multi-Cluster Kubernetes Architecture*. Accessed on 2024-04-21. 2023. URL: https://www.redhat.com/architect/multi-cluster-kubernetes-architecture (visited on 04/21/2024).

[61] D. Rosendo, A. Costan, P. Valduriez, and G. Antoniu. "Distributed intelligence on the Edge-to-Cloud Continuum: A systematic literature review." In: *Journal of Parallel and Distributed Computing* 166 (2022), pp. 71–94. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.04.004.

[62] L. Shi, Z. Wang, and Y. Zeng. "Edge Network Security Risk Control Based on Attack and Defense Map." In: *Journal of Circuits, Systems and Computers* 30.03 (2021), p. 2150046. DOI: 10.1142/S0218126621500468.

[63] T. Sigwele, Y. F. Hu, M. Ali, J. Hou, M. Susanto, and H. Fitriawan. "An Intelligent Edge Computing Based Semantic Gateway for Healthcare Systems Interoperability and Collaboration." In: *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2018, pp. 370–376. DOI: 10.1109/FiCloud.2018.00060.

[64] Y. Singh, F. Kandah, and W. Zhang. "A secured cost-effective multi-cloud storage in cloud computing." In: *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. 2011, pp. 619–624. DOI: 10.1109/INFCOMW.2011.5928887.

[65] S. Svorobej, M. Bendechache, F. Griesinger, and J. Domaschka. "Orchestration from the Cloud to the Edge." In: *The Cloud-to-Thing Continuum: Opportunities and Challenges in Cloud, Fog and Edge Computing* (2020), pp. 61–77.

[66] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella. "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration." In: 19.3 (2017), pp. 1657–1681. DOI: 10.1109/COMST.2017.2705720.

[67] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth. "mck8s: An orchestration platform for geo-distributed multi-cluster environments." In: *2021 International Conference on Computer Communications and Networks (ICCCN)*. 2021, pp. 1–10. DOI: 10.1109/ICCCN52240.2021.9522318.

[68] J. S. Ward and A. Barker. *A Cloud Computing Survey: Developments and Future Trends in Infrastructure as a Service Computing*. 2013. arXiv: 1306.1394 [cs.DC].

[69] S. B. Weinstein, Y.-Y. Lou, and T. R. Hsing. "Intelligent Network Edge with Distributed SDN for the Future 6G Network." In: *2021 IEEE International Conference on Microwaves, Antennas, Communications and Electronic Systems (COMCAS)*. 2021, pp. 261–265. DOI: 10.1109/COMCAS52219.2021.9629105.

[70] Y. Wu. "Cloud-edge orchestration for the Internet of Things: Architecture and AI-powered data processing." In: *IEEE Internet of Things Journal* 8.16 (2020), pp. 12792–12805.

[71] Z. G. Yang, L. L. Alejandro, C. I. Vergara, R. R. Maaliw, A. S. Alon, R. S. Evangelista, R. P. L. Rivera, and R. C. D. Santos. "Multi-Active Multi-Datacenter Distributed Database Architecture Design based-on Secondary Development Zookeeper." In: *2022 International Conference on Emerging Technologies in Electronics, Computing and Communication (ICETECC)*. 2022, pp. 1–6. DOI: 10.1109/ICETECC56662.2022.10069506.

[72] W. Zhang, A. Sharma, and T. Wood. "EdgeBalance: Model-Based Load Balancing for Network Edge Data Planes." In: *3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 20)*. USENIX Association, June 2020.

Appendix

1 Reproducibility

The complete measurement took place in a set-up of 8 VMs. The VMs were wiped each time to change the frameworks. The tests were carried out over the course of three weeks.

1.1 Set Up

The setup of the cluster is already described in the evaluation chapter. The tools used for this are KubeAdm with Calico, and this repository [27] was specifically used to quickly set up the clusters. To initiate the cluster setup, execute `commom.sh <ip>` on each node and `master.sh <ip>` on the master node. The `master.sh` script returns a command to register the worker nodes with the master node.
After a Kubernetes cluster has been set up, a Multi-Cluster Tool, such as Oakestra, Karmada, or OCM, must be installed. Additionally, a hub cluster is required for Karmada and OCM. Installation instructions can be found in the respective wiki or the README files of the plugins.
It is important that the Metrics Server and the Prometheus stack are deployed in the child cluster. This is necessary because the same script, located in /metrics/metrics.py, is used for all measurements in the overhead tests. This script requires access to the Prometheus server, which can be made accessible via a kubectl port-forward.
Metrics.py always creates a results folder containing all measurements. The specific measurements taken can be found within the script. This results folder can then be saved elsewhere. For more information about the test setup, refer to chapter 4.

1.2 Execution

A public repository is required for the tests [28]. A brief explanation is provided for each test, highlighting key considerations and aspects to monitor. The structure of the repository is as follows:

- **Tests**: Contains all the scripts required to perform the tests.

- **Evaluation**: Contains scripts for processing results and generating plots.
- **Results**: Stores all the raw test results.
- **Metrics**: Includes the scripts for performing the measurements.

Overhead Plugin

With the Overhead Plugin, no additional scripts are required. Only the plugins are installed, and measurements are taken using the metrics.py script.
Once completed, the measurements are saved in the corresponding subfolder within the results directory. These results are subsequently processed during the evaluation phase.

Overhead Management

In Overhead Management, each framework has an associated script to initiate and terminate deployments after a specified interval.
Several considerations must be addressed. Firstly, the configuration files for Karmada and OCM must be in their respective directories. These files should be manually saved in the same directory as the script. Furthermore, the root components of the clusters must always be accessible. In the case of Karmada and OCM, a hub cluster must be started separately. The IP of the respective root must only be stored with Oakestra. For OCM and Karmada, it is already stored in the config.
It is also important that a service.yaml file is present in each folder; this must be initiated in the respective cluster to ensure that the servers for the load test can be addressed. Additionally, within the Oakestra Setup, it can be decided whether the communication should occur via the Kubernetes or the Oakestra network. To specify this, enter the appropriate URL in the script: either a suitable service IP from the Oakestra network or the IP of the Kubernetes service.
The overall process is consistent: everything must be set up initially, and then the script can be executed. Once all deployments are active, the metrics.py script can be run to take the measurements.

Root Impact

With Root Impact, the various cluster setups must be initiated manually. Once a cluster setup is ready, the script in the Root Impact folder should be executed. There are two scripts in this folder: one for starting the workload and another for measuring CPU and RAM usage. The workload script must be executed first, and the measurement

script should be run only after active workloads. It is crucial to use the correct URL of the Oakestra Root.

Deployment Time

In the deployment test, there is a dedicated folder for each framework. The scripts within these folders execute the entire test logic. However, the respective script must be executed separately for each framework. A CSV file containing all timestamps is saved in the same folder.

1.3 Evaluation

All results are saved in the /results folder. The relevant results are further saved in another folder, /evaluation. In this folder, the results are converted into graphics for analysis.
The structure is consistent for each test. There is a subdivision for the respective focus of the graphic. Each folder contains a script, calculate.py, consolidating the .csv files from all frameworks and creating a new .csv with the relevant information. Additionally, several scripts generate plots, each focusing on a different aspect.

2 Detailed Architecture Kubernetes Deployments