BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

ORCHESTRATIONS AND RUNTIME ENVIRONMENTS OF SERVICES IN EDGE CLOUD DEPLOYMENT

ORCHESTRACE A BĚHOVÁ PROSTŘEDÍ SLUŽEB PŘI NASAZENÍ V EDGE CLOUD

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

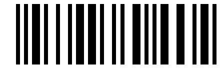
AUTHOR — **Bc. DAVID HURTA**
AUTOR PRÁCE

SUPERVISOR — **RNDr. MAREK RYCHLÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment



161900

Institut: Department of Information Systems (DIFS)
Student: **Hurta David, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Cybersecurity
Title: **Orchestrations and Runtime Environments of Services in Edge Cloud Deployment**
Category: Parallel and Distributed Computing
Academic year: 2024/25

Assignment:

1. Get familiar with the concepts of cloud computing and its variant: edge computing or edge cloud. Find out what types of devices are suitable for deployment in an edge cloud and what run-time environments for edge cloud services are suitable for such devices.
2. Design an example of a suitable application for edge cloud environments, e.g., applications for distributed data processing from sensors (using Hadoop, Spark or Flink). Implement the application.
3. Explore existing technologies for deploying, operating and managing services in the edge cloud, focus on container virtualisation tools (e.g., Kubernetes/K8s, K3s, KubeEdge, Fledge).
4. Design and implement a tool for deploying applications to the environments of the various technologies listed above and their monitoring. Test the tool and technology on the example of the application implemented in the previous step and compare the difficulty of deployment and using technology and a possible performance impact.
5. Describe and evaluate the results, suggest possible improvements and publish the whole project as open-source.

Literature:

- Xiong, Y., Sun, Y., Xing, L., Huang, Y.: "Extend Cloud to Edge with KubeEdge," 2018 IEEE/ACM Symposium on Edge Computing (SEC), 2018, pp. 373-377. Available at: https://doi.org/10.1109/SEC.2018.00048
- Goethals, T., De Turck, F., Volckaert, B. (2020). FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices. In: Hsu, CH., Kallel, S., Lan, KC., Zheng, Z. (eds) Internet of Vehicles. Technologies and Services Toward Smart Cities. IOV 2019. Lecture Notes in Computer Science(), vol 11894. Springer, Cham. Available at: https://doi.org/10.1007/978-3-030-38651-1_16
- Chang, H., Hari, A., Mukherjee, S., Lakshman, T. V.: "Bringing the cloud to the edge," 2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), 2014, pp. 346-351. Available at: https://doi.org/10.1109/INFCOMW.2014.6849256
- Wang, J., Pan, J., Esposito, F., Calyam, P., Yang, Z., Mohapatra, P.: Edge Cloud Offloading Algorithms: Issues, Methods, and Perspectives. ACM Comput. Surv. 52, 1, Article 2 (January 2020), 23 pages. Available at: https://doi.org/10.1145/3284387

Requirements for the semestral defence:
Items 1, 2, and 3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Rychlý Marek, RNDr., Ph.D.**
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: 1.11.2024
Submission deadline: 21.5.2025
Approval date: 22.10.2024

Abstract

The thesis aims to explore edge cloud computing and its existing management technologies for services to help edge cloud cluster administrators make a more informed decision when determining a technology for their cluster. The thesis has created a project for simple automated provisioning and deployment of Kubernetes, K3s, MicroK8s, and KubeEdge clusters using a unified interface and tooling for deploying and monitoring services in these clusters. The thesis has also compared deployment difficulty, usage, and performance impact on nodes and services between the technologies using an implemented demonstration application based on Apache Kafka technology. The developed software has been published as open-source. The thesis has identified significant differences between the technologies through statistical measurements, data visualization, and statistical tests of collected performance metrics. The developed tooling, comparisons, and analyzed data enable cluster administrators to explore and compare the technologies easily to make a more informed decision concerning the management technology.

Abstrakt

Tato práce má za cílem prozkoumat edge cloud computing a jeho existující technologie pro správu služeb, aby pomohla správcům edge cloud klastrů činit informovanější rozhodnutí při výběru technologie pro jejich klastr. V rámci práce byl vytvořen projekt pro jednoduché automatizované zprovoznění a nasazení klastrů Kubernetes, K3s, MicroK8s a KubeEdge pomocí jednotného rozhraní a nástroje pro nasazení a monitorování služeb v těchto klastrech. Práce taktéž porovnala obtížnost nasazení, používání a vliv na výkon výpočetních uzlů a služeb mezi technologiemi pomocí implementované demonstrační aplikace využívající technologii Apache Kafka. Vyvinutý software byl publikován jako open-source. Práce identifikovala významné rozdíly mezi technologiemi prostřednictvím statistických měření, vizualizací dat a statistických testů shromážděných výkonnostních metrik. Vyvinuté nástroje, porovnání a analyzované data umožňují správcům klastrů snadno prozkoumat a porovnat zmíněné technologie pro učinění informovanějších rozhodnutí ohledně řídící technologie.

Keywords

Cloud Computing, Edge Computing, Edge Cloud Computing, Virtualization, Containerization, Kubernetes, KubeEdge, K3s, MicroK8s, Cluster Deployment, Services Deployment, Services Monitoring, Terraform, Ansible, Comparison

Klíčová slova

Cloud Computing, Edge Computing, Edge Cloud Computing, Virtualizace, Kontejnerizace, Kubernetes, KubeEdge, K3s, MicroK8s, Nasazení Klastrů, Nasazení Služeb, Monitorování Služeb, Terraform, Ansible, Porovnání

Reference

HURTA, David. *Orchestrations and Runtime Environments of Services in Edge Cloud Deployment*. Brno, 2025. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Marek Rychlý, Ph.D.

Rozšířený abstrakt

Cílem této práce je prozkoumat edge cloud computing a jeho existující technologie pro nasazení, provoz a správu služeb, aby podpořila správce edge cloud výpočetních klastrů činit informovanější rozhodnutí při výběru vhodné technologie. Práce usiluje o dosažení cíle vytvořením nástrojů na nasazení těchto technologií v edge cloud prostředí, dále nástrojů pro nasazení a monitorování služeb v těchto prostředí a také porovnáním obtížnosti zprovoznění a používání technologií a jejich vlivu na výpočetní uzly a nasazené služby.

Za účelem naplnění stanoveného cíle byly prozkoumány paradigmata cloud computing, edge computing a jejich sjednocení v podobě edge cloud computing. Tento přístup kombinuje výhody obou paradigmat pro využití výkonných centralizovaných výpočetních center s nízkou latencí dosaženou zpracováním dat přímo u jejich zdroje. V rámci práce byly dále zkoumány vhodné typy zařízení napříč edge cloud prostředím a běhová prostředí služeb pro takové zařízení se zaměřením na virtualizaci pomocí virtuálních strojů a kontejnerizace.

Demonstrační aplikace využívající technologii Apache Kafka byla navrhnuta a implementována jako příklad mikroslužbové aplikace navržené přímo pro edge cloud prostředí. Aplikace se skládá ze služeb produkující metriky, které reprezentují senzory mimo cloudové prostředí. Tato data jsou navržena pro zpracování u jejich zdroje. Předzpracovaná data jsou následně nahrány do cloudu, kde jsou dále zpracovány. Tato aplikace představuje demonstrační příklad využití výhod edge cloud computingu.

Byly prozkoumány existující technologie pro nasazení, provoz a správu služeb v edge cloud prostředí se zaměřením na nasazení aplikací pomocí kontejnerů. Pozornost byla věnována zejména technologii Kubernetes a odlehčeným distribucím K3s, k0s, MicroK8s, dále rozšíření KubeEdge a také populární alternativě Nomad.

Následně byly prozkoumány způsoby nasazení těchto technologií od manuálních a automatizovaných nástrojů po klastry vytvořené cloudovými poskytovateli. Manuální nástroje jsou dostupné u všech technologií a umožňují přesnou konfiguraci klastru správcem. Automatizované nástroje umožňují škálovatelné nasazení a klastry spravované cloudovými poskytovateli představují nejjednodušší způsob obstarání klastrů požadované technologie. Většina populárních edge cloud technologií však není podporovaná napříč všemi typy nástrojů. Správce je tedy nucen pro zkoumání a experimentování s technologiemi využít větší množství odlišných nástrojů nebo investovat úsilí a čas využitím manuálních nástrojů. Tato problematika nasazení edge cloud technologií byla podrobněji popsána a bylo navrženo řešení pro automatizované nasazení klastrů edge cloud technologií, nasazení služeb v těchto klastrech a shromáždění metrik pro následné porovnání technologií pomocí individuálních komponent za cílem pomoci správcům edge cloud klastrů jednoduše prozkoumat a porovnat technologie.

V rámci práce byly navrhnuty a implementovány nástroje pro jednoduché automatizované zprovoznění a nasazení klastrů Kubernetes, K3s, MicroK8s a KubeEdge pomocí jednotného rozhraní. Toho bylo dosáhnuto využitím technologie Terraform pro zajištění výpočetních prostředků u cloud poskytovatele a technologie Ansible pro konfiguraci výpočetních uzlů za cílem nasazení zmíněných technologií. Terraform projekt obstarává zajištění prostředků jako jsou virtuální stroje, firewall a vyvažovač zátěže dle specifikací uživatele. Ansible projekt je následně schopen nasadit specifikovanou technologii pomocí instalace nutných závislostí, konfigurací komponent, instalací technologie a pomocných služeb. Tyto projekty byly plynule integrovány pro jednoduché použití, avšak fungují i jako samostatné nástroje. Taktéž byla implementována aplikace pro nasazení a monitorování služeb v klastrech těchto technologií, která komunikuje s příslušným serverem aplikačního rozhraní pro nasazení příslušných objektů a načtení dat pro následné formátování za cílem monitorování služeb. Celý projekt byl průběžně integrován a testován pro zajištění kvality a stability

nástrojů pomocí testování celého systému od začátku do konce pomocí nástroje GitHub Actions. Software pro shromažďování a porovnání výkonnostních metrik těchto klastrů byl vyvinut. Prometheus server a příslušné služby jsou nasazeny pro sběr metrik klastru a služeb. Agregované metriky využití běžících klastrů jsou shromážděny do perzistentní databáze. Software na načtení shromážděných výkonnostních metrik a následné porovnání byl vyvinut za cílem porovnání vlivu technologií na výkon uzlů a služeb. Software pro tvorbu statistických měření, vizualizaci rozdělení dat a výpočet statistických testů pro zjištění statisticky významných rozdílů byl vyvinut. Práce následně porovnala obtížnost nasazení, používání a vliv na výkon výpočetních uzlů a služeb mezi technologiemi pomocí implementované demonstrační aplikace.

Práce identifikovala významné rozdíly mezi technologiemi prostřednictvím statistických testů, měření a vizualizací dat shromážděných výkonnostních metrik. Byly nalezeny tendence ve vlivu využití procesoru a paměti uzlů a také byly identifikovány statisticky významné rozdíly mezi uzly technologií a to především mezi uzly řídící roviny. Také tendence a významné rozdíly využití procesoru a paměti byly nalezeny mezi jednotlivými kontejnery demonstrační aplikace při nasazení jednotlivými technologiemi. Vyvinutý software byl publikován jako open-source. Pomocí vyhodnocení porovnání technologií a vyvinutých nástrojů je správce schopen na základě svých požadavků a plánovaných služeb k nasazení snadno prozkoumat a porovnat zmíněné technologie a učinit informovanější rozhodnutí při výběru.

Orchestrations and Runtime Environments of Services in Edge Cloud Deployment

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Mr. Marek Rychlý. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
David Hurta
May 19, 2025

Acknowledgements

I would like to wholeheartedly express my gratitude to my supervisor, Mr. Marek Rychlý, for his invaluable feedback and guidance throughout the course of the thesis.

Contents

1 Introduction **4**

2 Extending the Cloud to the Edge **6**
2.1 Cloud Computing . . . 6
2.2 Edge Computing . . . 10
2.3 Edge Cloud Computing . . . 11

3 Demonstration Monitoring Application for Edge Cloud **18**
3.1 Design of the Application . . . 18
3.2 Implementation of the Application . . . 21

4 Managing Container Services in Edge Cloud **24**
4.1 Kubernetes . . . 24
4.2 Lightweight Kubernetes Distributions . . . 26
4.3 KubeEdge . . . 27
4.4 Nomad . . . 28
4.5 Summary . . . 29

5 Analysis of Edge Cloud Cluster Deployment **30**
5.1 Existing Deployments Tools . . . 30
5.2 Challenges of Edge Cloud Cluster Deployment . . . 34
5.3 Addressing the Challenges . . . 35

6 Design of Edge Cloud Services Deployment Project **38**
6.1 Cluster Deployment . . . 38
6.2 Services Deployment and Monitoring . . . 41
6.3 Cluster Metrics Collection . . . 42
6.4 Development and Testing . . . 43
6.5 Summary . . . 46

7 Implementation of Edge Cloud Services Deployment Project **48**
7.1 Provisioning of Resources . . . 48
7.2 Installation of Management Technologies . . . 49
7.3 Deployment and Monitoring of Services . . . 54
7.4 Development and Testing . . . 55
7.5 Cluster Metrics Comparison . . . 58
7.6 Summary . . . 60

8 Comparison of Explored Edge Cloud Technologies **61**

8.1 Deployment and Usage 61
8.2 Performance Impact on Services and Nodes 62
8.3 Summary 67

9 Conclusion **70**

Bibliography **71**

List of Figures

2.1 An edge cloud computing architecture adapted from [3]. 12
2.2 A distributed edge cloud architecture adapted from [10]. 13
2.3 Cloud and edge layers adapted from [26]. 14
2.4 An overview of virtual abstractions adapted from [21]. 15
2.5 An anatomy of an edge application adapted from [10]. 17

3.1 A high-level design of the overall architecture. 19
3.2 The application architecture represented as two Kafka clusters. 20

4.1 A diagram of the Kubernetes architecture adapted from [13]. 25
4.2 The KubeEdge architecture adapted from [19]. 28

5.1 Schema of the project for deployment and monitoring of edge cloud services. 37

6.1 Visualized Terraform stages adapted from [40]. 39
6.2 Visualization of main Ansible components adapted from [29]. 41
6.3 An overview of deployment and monitoring of services. 42
6.4 An overview of cluster metrics collection and visualization. 43
6.5 A high-level overview of a single test run. 45
6.6 A simplified architecture of the overall project upon continuous integration. 47

7.1 An example of provisioned DigitalOcean resources by the project. 49
7.2 A diagram of the relationships between the components of the project. . . . 50
7.3 An example of monitoring services using the implemented CLI application. 55
7.4 A diagram of the demonstration application using Kubernetes API objects. 56
7.5 A successful CI run overview upon publishing a change. 58
7.6 A schema of the table used for storing the collected metrics. 59

8.1 A summary of the median CPU usage across nodes and technologies. 63
8.2 A summary of the median memory usage across nodes and technologies. . . 63
8.3 A distribution of CPU usage between nodes and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. 64
8.4 A distribution of memory usage between nodes and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. 65
8.5 A summary of the median CPU usage between containers and technologies. 66
8.6 A summary of the median memory usage between containers and technologies. 66
8.7 A distribution of CPU usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. 68
8.8 A distribution of memory usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. . . 69

Chapter 1

Introduction

Cloud computing and edge computing have become increasingly popular across industries in recent years. The power of centralized data centers enables the simple provisioning of a plethora of resources and services with the click of a button. Models such as infrastructure, platforms, and software as a service have empowered organizations worldwide. On the other side, edge computing has enabled services by processing data close to its source on behalf of cloud services. To truly reach their potential, edge cloud computing aims to seamlessly incorporate the paradigms to enable services with the power of data centers and the low latency and preprocessing due to proximity.

However, deploying, operating, and managing services across vastly different environments is not a simple task. Elaborate technologies to address these issues exist; however, deployment, usage, and management of these technologies require considerable time investment for training due to their complex nature. This fact may intimidate beginner cluster administrators or demotivate exploration of technologies other than easily deployable ones.

The thesis aims to explore edge cloud computing and its existing management technologies for services to help edge cloud cluster administrators make a more informed decision when determining a technology for their cluster by enabling them to explore the technologies easily and providing a comparison based on difficulty in deployment, usage, and performance impact on nodes and services.

Managing services across nodes may seem simple at first. However, beneath the surface of these technologies lies a complex ecosystem of fascinating components and technologies. From containerization and networking to even the management of whole pools of nodes using an application programming interface. These technologies empower the banking, pharmacy, and technology sectors, among others, to achieve more. The intriguing nature of these technologies and their importance in today's world are one of the primary reasons for my interest in this thesis, as I am motivated by a genuine curiosity about the field.

The following chapter aims to explore and describe the key concepts and characteristics of cloud computing, edge computing, and their unification, edge cloud computing. Among others, the suitable types of devices and runtime environments are explored, as this issue is important for managing services across these environments. Chapter 3 designs and implements a microservices demonstration edge cloud application based on Apache Kafka technology to practically showcase the benefits of the computing paradigm. Chapter 4 explores existing technologies for deploying, operating, and managing services in the edge cloud environment. Chapter 5 analyzes tools for the deployment of the technologies and discusses their challenges. Chapter 6 designs a project to address these challenges using a number of components to enable cluster administrators to easily deploy edge cloud clus-

ters and services using a unified interface. A design for continuous integration and testing of these components and a subsequent comparison of the performance impact on nodes and services by the technologies are also described. The implementation of the project using technologies such as Terraform and Ansible is described in detail in Chapter 7. A comparison of difficulty in deployment and usage between explored technologies is made in Chapter 8, including a comparison of performance impact on nodes and services using data visualization and statistical measurements and tests. Finally, Chapter 9 concludes the thesis and suggests possible future advancements.

Chapter 2

Extending the Cloud to the Edge

Multinational corporations, software engineers, and even computer science students are no longer limited to deploying their applications on local premises to achieve their goals. Maintaining a cluster of machines on a local premise can become an expensive and complicated task. Cloud computing solves many of these issues and enables users to do even more. However, cloud computing does not solve all use cases. Edge computing focuses on processing data at the edge of a network to reduce overall latency for immediate actions that otherwise would not be possible using cloud computing.

The goal of this chapter is to examine in more detail cloud and edge computing and edge cloud computing, a paradigm whose goal is to unify the cloud and the edge. The concepts of cloud and edge computing paradigms are explored. Subsequently, edge cloud computing is examined in more detail, as well as applications and suitable types of devices and runtime environments for edge cloud environments.

2.1 Cloud Computing

The terms cloud and cloud computing have evolved in the last few decades, becoming increasingly popular and used. As such, different entities provide different definitions. This subchapter focuses on describing these terms and the concepts behind them. This subchapter is taken from [11] if not specified otherwise.

2.1.1 Key Concepts and Terms

Cloud computing has emerged as a new paradigm in distributed computing, which is a field that studies distributed systems. A distributed system consists of multiple autonomous machines, each having its own private memory, communicating through a computer network, where the exchange of information is accomplished through message passing. [12]

As mentioned, there exist multiple definitions for cloud computing. The National Institute of Standards and Technology (NIST) defines cloud computing as follows.

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."* [22]

This is a rather descriptive definition, with characteristics and the models described in more detail in the original paper. However, this thesis will adopt a more concise definition of cloud computing introduced by Thomas Erl, Ricardo Puttini, and Zaigham Mahmood [11].

> *"Cloud computing is a specialized form of distributed computing that introduces utilization models for remotely provisioning scalable and measured resources."*

This definition is mentioned to be in line with various definitions put forth by other organizations within the cloud computing industry. The following are some of the most important terms used in cloud computing.

Cloud

The term cloud itself refers to a distinct IT environment that is designed to remotely provision scalable and measured IT resources. An IT resource represents a physical or virtual IT-related artifact that can be either software-based, such as a virtual server or a software program, or hardware-based, such as a physical server or a network device. A cloud is further distinguished by ownership, size, and access.

It is also important to distinguish between cloud-based and on-premise IT resources. On-premises IT resources are IT resources located on the premises, in other words, within the organizational boundaries of a conventional IT enterprise. On-premise IT resources cannot be cloud-based IT resources, and vice versa. On-premise resources can be moved to the cloud to become cloud-based. However, the main idea is to differentiate between IT resources located in a cloud and within an organization that utilizes a cloud.

Cloud Provider

A party that provides cloud-based IT resources is referred to as a cloud provider. A cloud provider is often obligated to provide IT resources as agreed upon in an SLA (Service Level Agreement) guarantees. Thus, cloud providers are responsible for the provided resources and their availability.

Cloud Consumer

Cloud providers provide IT resources to their customers, also called cloud consumers. These two parties have a formal contract or arrangement that specifies the details of a given contract. Cloud consumers are then able to utilize cloud-based IT resources provided by a given cloud provider.

2.1.2 Cloud Characteristics

An IT environment must consist of the following essential characteristics to be considered an effective cloud. This allows the cloud consumers to fully utilize cloud computing.

- **On-Demand Usage**: A cloud consumer is able to self-provision cloud-based IT resources. This process can be automated, which results in no human involvement from the cloud consumer or cloud provider. Thus, IT resources can be self-provisioned on demand. This enables the service-based and usage-driven features found in the mainstream clouds.

- **Broad Access**: Cloud services are widely accessible to cloud consumers. To achieve this, a cloud service may require support for various cloud service consumers that may utilize a variety of devices, transport protocols, and security technologies. This may require that cloud service architectures be designed with particular cloud service consumers in mind.
- **Multi-tenancy and Resource Pooling**: Multitenancy is the ability of a service to serve multiple different consumers while securely isolating them from each other. In cloud computing, this means that several cloud consumers can use the same IT resource simultaneously, while each remains unaware that it can be used by other cloud consumers. In single-tenant environments, each cloud consumer has a separate IT resource. A cloud provider pools its IT resources to serve cloud consumers using multi-tenancy models. Using multi-tenancy technologies, IT resources can be dynamically assigned according to demand. Resource pooling allows cloud providers to pool large-scale IT resources to serve multiple cloud consumers.
- **Elasticity**: Elasticity is the ability of a cloud provider to transparently and automatically scale the IT resources of cloud consumers. This is done as required or as determined by the cloud consumer or the cloud provider. The pooling of IT resources enables the cloud provider to effectively scale the resources of cloud consumers.
- **Measured usage**: Measured usage is the ability of a cloud provider to monitor the usage of its IT resources. This enables the cloud provider to potentially charge cloud consumers as agreed upon by the parties. However, measured usage is not limited to billing purposes. It encompasses the monitoring of cloud providers' IT resource usage and related usage reporting for both cloud providers and cloud consumers.
- **Resiliency**: Resiliency refers to the redundancy of IT resources. Cloud consumers can take advantage of the resilience of cloud-based IT resources to increase the reliability and availability of their applications.

2.1.3 Cloud Service Models

Cloud service models, also called cloud delivery models, represent specific packaged combinations of IT resources that a cloud provider can offer to cloud consumers. It is also possible to provide combinations of these models. The most widely established and common models are the following.

Infrastructure as a Service

The Infrastructure as a Service (IaaS) model provides a self-contained IT environment consisting of infrastructure-related IT resources such as hardware, network, connectivity, and operating systems. These resources are typically virtual and packaged into bundles. Virtualization simplifies the scaling and customization of the infrastructure.

The main idea of IaaS is to provide cloud consumers with a high level of control over the infrastructure provided. Thus, the provided IT resources are usually not configured by the cloud provider, and the responsibility is placed on the cloud consumer. This is ideal for cloud consumers who require a high level of control over the overall cloud environment.

Platform as a Service

The Platform as a Service (PaaS) model provides a configured and ready-to-use cloud environment. The environment usually consists of packaged products and tools to support custom applications deployed by a cloud consumer. This simplifies the use of a cloud environment for cloud consumers, as the administration of the underlying infrastructure is managed by a cloud provider. However, this grants a lower level of control to cloud consumers, as they can be limited in controlling the underlying IT resources responsible for the platform.

Software as a Service

The Software as a Service (SaaS) model provides a software program as a service. These software programs are configured to act as a shared cloud service and are available as a service or as a utility. These services are widely available to a range of cloud consumers. A cloud consumer can utilize these services as agreed upon by the parties. Typically, the services are commercially provided, and the measured usage is billed to the respective cloud service consumers. The main idea is to only provide the software as a service, and thus, the cloud consumer has little to no level of control over the implementation of the service.

2.1.4 Cloud Deployment Models

A cloud can be categorized into four main types, distinguished primarily by ownership, size, and access. The most common models are as follows.

Public cloud

A public cloud represents a publicly accessible cloud. IT resources available in a public cloud are generally offered to cloud consumers at a cost or are commercialized through other means. The cloud provider is responsible for the creation and maintenance of its public cloud and IT resources.

Public clouds were the first class of clouds that were implemented and offered. They account for the first expression of cloud computing, in which they are the realization of the canonical view of cloud computing, where services offered are made available to anyone, from anywhere, anytime through the Internet. [1]

Community cloud

Community clouds are similar to public clouds; the main difference is the limited access to its IT resources. A community cloud can be accessed only by its specific community of cloud consumers. The cloud can be jointly owned by its community members or by a cloud provider that provides limited access to a provisioned cloud. However, membership in a community cloud does not automatically guarantee access to or control of all its cloud-based IT resources. The main characteristic of a public cloud is its public access by anyone. In the community cloud, users outside a community cannot access its cloud unless allowed.

Private cloud

A private cloud is owned by a single organization and enables the organization to utilize cloud computing technology. Thus, cloud computing is used to centralize access to IT

resources owned by the organization. The organization is a cloud provider and also a cloud consumer. Different departments within the organization can assume different roles.

Cloud-based and on-premise IT resources are not the same concept in a private cloud. It is important to distinguish them correctly. Although the private cloud may physically reside on the organization's premises, its IT resources are still considered cloud-based as long as they are remotely accessible to cloud consumers.

Hybrid cloud

A hybrid cloud combines two or more different cloud deployment models. The deployment of such architectures presents a problem due to disparities in different cloud environments.

However, the complexities associated with a hybrid cloud are worth solving. For example, the hybrid cloud can take advantage of the best characteristics of public and private clouds. Hybrid clouds allow an organization to utilize existing IT infrastructure and process sensitive data within the premises using a private cloud model, and dynamically provision IT resources using a public cloud model. [1]

2.2 Edge Computing

Cloud computing empowers users to remotely provision scalable resources as needed using existing cloud infrastructure. The cloud computing paradigm has many benefits and is ideal for certain use cases. However, one of its flaws is the fundamental fact that data must be transmitted from its source to the cloud, processed, and subsequently sent to end users. This creates a latency that results from the transmission of the information itself. Cloud providers solve this problem to an extent by building geographically dispersed clouds. This decreases latency, but there are certain use cases in which latency is a critical aspect and using clouds is not acceptable. Introducing edge computing, where computing is done as close to the source of data as possible to provide multiple benefits, including low latency.

Edge computing has recently emerged as a new computing paradigm. This causes the meaning of edge computing to slightly differ in industry and academia. Although exact definitions and meanings may differ from each other, the main idea remains the same. In edge computing, the computation should occur at the proximity of data sources [2].

Traditionally, using cloud computing, real-time data is collected from various sources. The data is then propagated to a cloud for processing and storage. However, processing data in this way is not sufficient when processing needs to be done in real time. Processing data in proximity to their source reduces latency and bandwidth and increases security. This can be achieved by placing the computing resources at the edge of the network near the data sources. Performing the processing at the edge of the network near the data sources reduces latency, and thus, data becomes actionable in near real-time. Data can also be aggregated at the edge before being transferred to the cloud to reduce bandwidth. [20]

Some of the main benefits of edge computing thus, are [20]:

- Reduction in computation latency.
- Reduction in network bandwidth.
- Preservation of data security and privacy.
- Reliability in network failures.

- Reduction in operating costs due to communication, storage, and processing.

The following is a formal definition of edge computing that will be followed in this thesis.

> *"Edge computing refers to the enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services."* [2]

Edge computing should be considered as a complement to cloud computing and not a replacement. When cloud computing fails to deliver acceptably due to its nature, edge computing can be utilized in addition for edge-related tasks, if applicable.

2.3 Edge Cloud Computing

The term edge cloud computing refers to the combination of edge computing and cloud computing. To reduce service latency, it is natural to integrate edge computing or edge cloud computing into the design and engineering of a system. Edge cloud computing combines the advantages of both edge computing and cloud computing. In the edge cloud computing architecture, an edge device can offload complicated tasks to an edge server within its proximity. Based on resource utilization and task requirements, the edge server can choose to complete the task execution entirely by itself or perform part of the task execution and then request remote cloud servers to continue task processing. The local information collected by a set of edge servers located in different geographic locations can also be transmitted to the cloud for an optimal global decision. Therefore, edge cloud computing has the ability to reduce service latency through edge computing and increase computation capacity through cloud computing. An architecture of an edge cloud computing solution is available in Figure 2.1 to showcase the differences between the computing paradigms. [3]

A tremendous amount of data is generated at the edge of the network, for example, video captured by smartphones, traffic from sensors, and surveillance video feeds from computers. The data is usually transported back to the cloud for storage and processing, which requires high-bandwidth connectivity to the central cloud. However, a significant portion of the data can undergo preprocessing and compression. For example, user video can be compressed, communication can be coded, and surveillance video can be filtered for incidents. With preprocessing, the transport cost can be significantly reduced. However, the current cloud computing model, which is centered around centralized data centers, does not support the large-scale computation required at the edge of the network. [10]

The above cases indicate that while high-performance applications require a cloud, moving the cloud to the edge, either completely or partially, can result in lower latency and bandwidth usage for the end user. This necessitates a new model where the centralized cloud is enhanced with a presence at the edge to accommodate the emerging edge-based applications that involve intensive computation and data processing. [10]

In this thesis, a hybrid cloud architecture model known as Edge Cloud, introduced by Hyunseok Chang *et al.* [10], will be adopted. This model was designed to provide end-user services with low latency, efficient use of bandwidth, and resiliency on a global scale. The Edge Cloud aims to bring the cloud closer to the end user, whether they are at home or in an enterprise setting, by utilizing compute nodes contributed by both users and operators at the edge. The Edge Cloud provides a split cloud application architecture by seamlessly interconnecting edge networks with data center networks, thereby enabling latency-sensitive

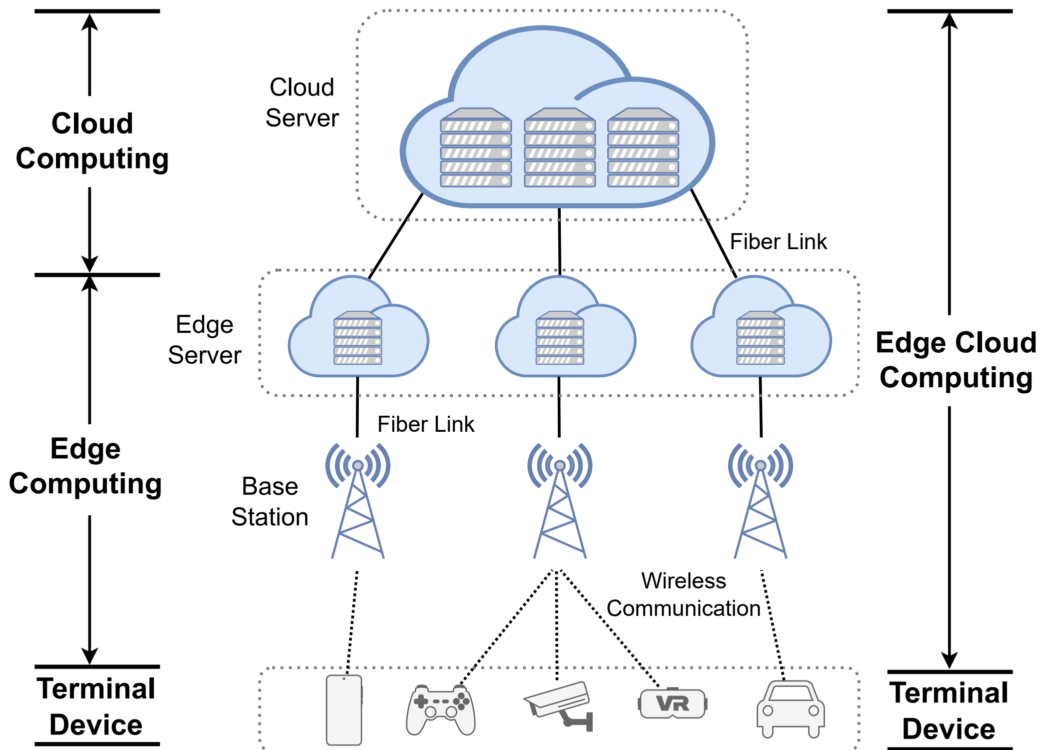


Figure 2.1: An edge cloud computing architecture adapted from [3].

computation and user interaction components close to end users at the edge, while hosting additional heavy-duty processing and database components in the cloud. [10]

The edge cloud architecture benefits end users from the low latency between the users and the Edge Cloud, which is now part of their local networks. The latency benefit is particularly significant for users far from the cloud. Another benefit is the bandwidth reduction provided by preprocessing at the edge, which reduces the bandwidth requirements between the edge and the cloud. The Edge Cloud is able to access various edge resources, such as sensors and computers, which are not visible outside the edge network. Another benefit is the introduced resiliency. In case edge resources become temporarily unavailable, edge components of an application can fall back to the more reliable cloud. In contrast, if an edge component can provide the basic functionality of an application itself, the application can continue to function even without the cloud. The Edge Cloud can provide graceful degradation of service, which is a valuable feature for any cloud application. [10]

2.3.1 Architecture

An Edge Cloud is a federation of data center nodes in the cloud, along with attached nodes at the edge of the network. An edge network refers to a part of the Internet to which end users can directly connect, such as home or enterprise networks or WiFi hotspots. An edge node refers to a compute or storage node attached to the edge network and federated to a cloud's data center. Edge nodes in the same edge network are grouped to form an edge zone. All nodes in the same edge zone are assumed to be part of the same network address

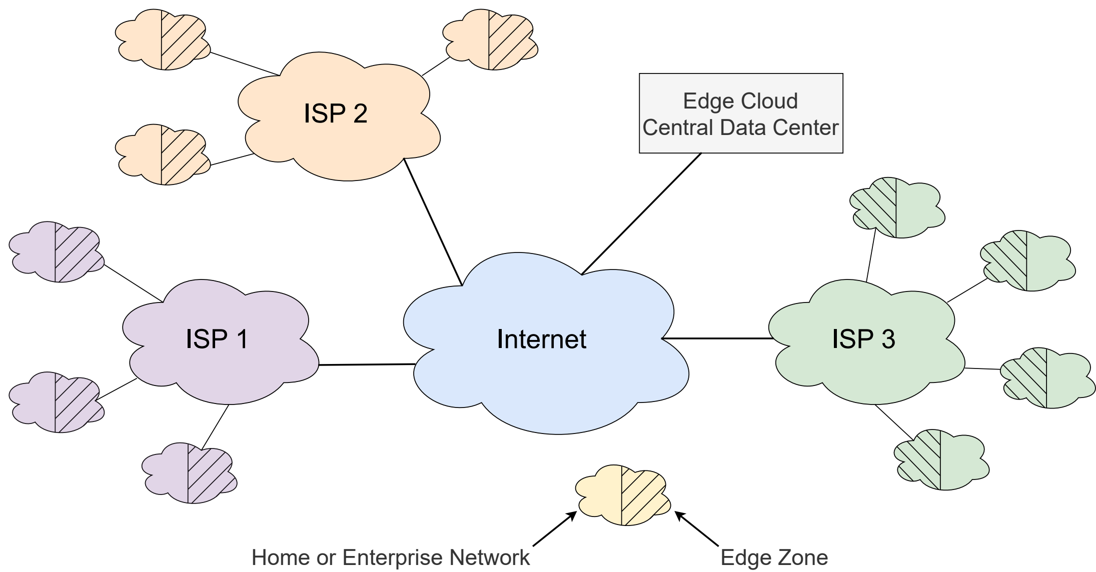


Figure 2.2: A distributed edge cloud architecture adapted from [10].

space managed by an edge owner. The architecture is shown in Figure 2.2. The Edge Cloud is the federation of the cloud's data center nodes along with all the edge zones. The Edge Cloud operator is assumed to have a preexisting, traditional IaaS data center cloud. By adding Edge Cloud functionality, the Edge Cloud operator can now extend the cloud's capabilities to deploy applications at the edge networks. [10]

The interpretation of the term edge may vary with context. Jie Cao, Quan Zhang, and Weisong Shi [2] define the term edge as any computing and network resources along the path between data sources and cloud data centers. The architecture of a unified cloud and edge computing solution can be broken down into layers to showcase how data is processed in different contexts. The visualization of these layers is represented in Figure 2.3. The layers can be described as follows [26]:

- Cloud Layer: This layer represents a cloud, which can be used for extensive data storage and intensive computations.

  Data centers often consist of standardized commodity servers that have substantial processing and storage capacity. Standardized racks with interconnections for power, network, and internal cooling, support for different hardware processing architectures, power-efficient multi-core architectures that encapsulate hundreds of processing cores, redundant and capable of being replaced without turning off system components, such as hard disks, power supplies, and network interfaces, specialized storage systems, all these technologies and more can be found in data centers. [11]

- Near edge: This layer represents the network devices that move data between the cloud layer and the far edge layer. These include 5G networks, radio devices, telecommunication devices, and similar devices. The name of the layer refers to the location of these devices relative to the cloud. These devices tend to be near the cloud.

- Far edge: This layer represents the edge servers. This layer is often responsible for collecting data from various devices and processing the data at the edge. The layer

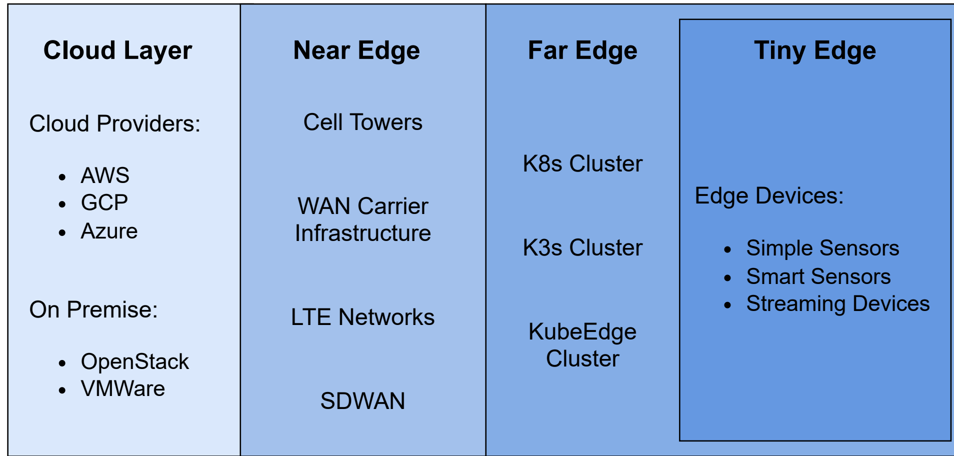


Figure 2.3: Cloud and edge layers adapted from [26].

can exchange data between the cloud and the edge layer. These devices tend to be far from the cloud; hence the name far edge.

The most important components in this layer are the edge servers and gateways. An edge server has higher computational and storage capacity relative to an edge device. These servers tend to be general-purpose compute nodes or a cluster of compute nodes. Edge gateways perform network functions such as protocol mapping, network termination, tunneling, and firewall. [20]

- Tiny edge: This layer is a subdivision of the far edge layer. This layer represents the devices that exchange data with the edge servers on the far edge, such as smart-watches, cameras, sensors, etc.

  These devices, also called edge devices, are often designed to perform a specific task cost-effectively and therefore have limited compute and storage resources. Their main objective is to collect information, send it upstream, or transfer it downstream. [20]

It is worth noting that the processor architecture is an important aspect when designing an edge computing system. One of the most popular architectures is the ARM architecture because of its lower energy consumption. Edge servers and edge devices tend to have less computational power compared to data centers in the cloud. For example, Intel processors consume more energy and provide more computational power compared to ARM processors. However, ARM processors may provide more value than Intel processors when considering both their performance and energy efficiency for their cost. Companies are thus interested in designing micro data centers using ARM processors. For the same reason, some companies even migrate their workloads to devices using ARM processors. [26]

However, edge systems can be identical to the systems used in enterprise data centers or devices more similar to hardened embedded computers. Regardless of the hardware, the main goal is for the hardware to serve the intended workload and use cases. [21]

The selection of various devices is crucial when designing a system that may span several hundred nodes and various environments. However, the runtime environments that host and support the actually deployed applications also play a significant role.

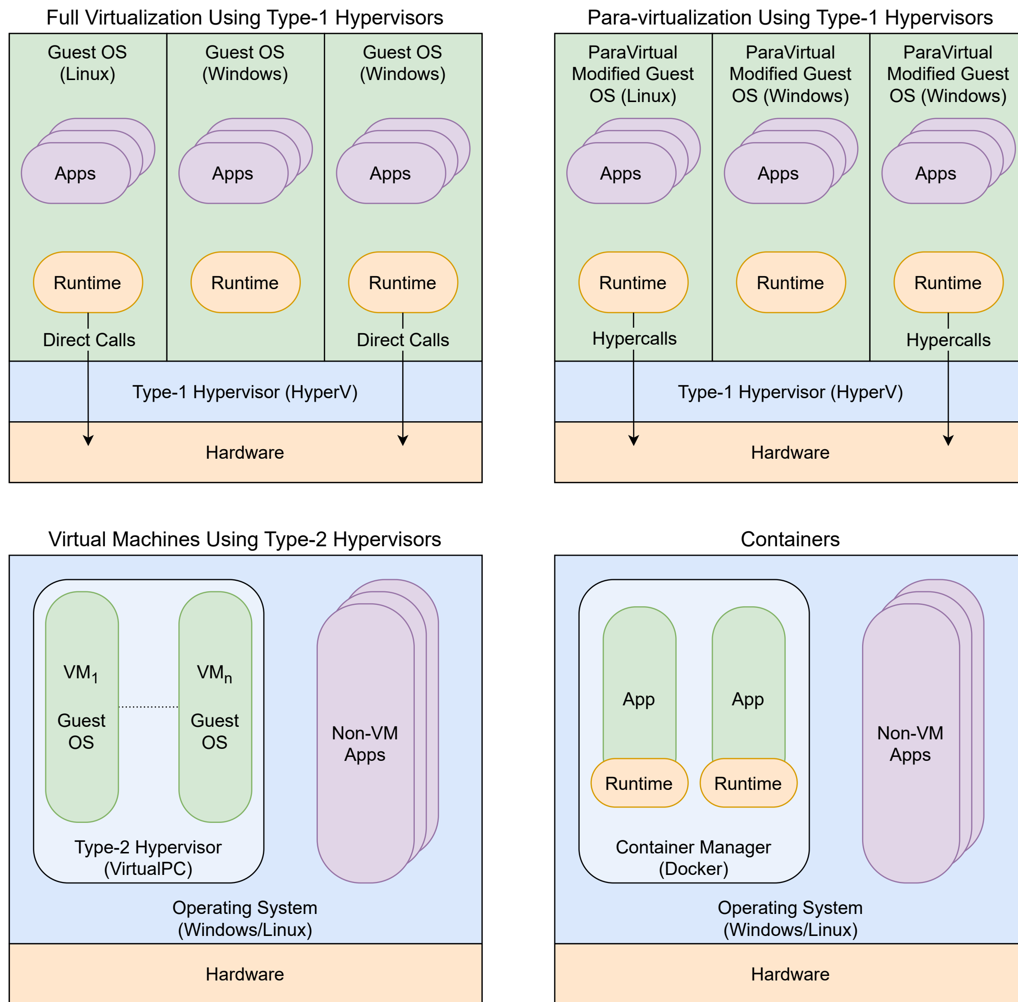


Figure 2.4: An overview of virtual abstractions adapted from [21].

2.3.2 Runtime Environments

As the number of edge servers and edge devices grows, their management becomes a complex issue. The runtime environments in which the applications are deployed should be robust, controlled, and responsive. The environment should be able to receive, reimagine, and run software. A central service should be able to manage and monitor the environment. Finally, the environment should be able to report back information about the application reimaging. These properties can be achieved using virtualization technologies, such as virtual machines and containers. An overview of these technologies is shown in Figure 2.4. [21]

Virtual Machines

Virtual machines have already been widely used in cloud computing. A virtual machine usually refers to a hypervisor that emulates the underlying hardware. [2]

Hardware virtualization provides a hardware abstraction and is generally capable of running any software that can run on bare metal. It uses a hypervisor to manage one or

more virtual machines on the processor and can support virtual replication of hardware to multiple virtual operating systems. These techniques require processor and hardware support for virtualization. [21]

There are two types of hypervisors in hardware virtualization [2]:

- Type-1, also called native or bare metal hypervisors: These hypervisors run directly on the host hardware and control the hardware and manage guest operating systems.
- Type-2, also called hosted hypervisors: These hypervisors run on a host operating system and abstract guest operating systems on top of the host operating system. A guest operating system runs as a process on the host OS.

Another type of virtualization is para-virtualization. Para-virtualization provides hardware abstraction and requires special drivers. The drivers are linked through the underlying hypervisor and access the hardware through hypercalls. This type of virtualization requires modifications to the guest operating system and offers the guest operating system higher performance and the ability to communicate directly with the hypervisor. [21]

Containers

A container is a resource-isolated process that runs an application and its dependencies [2].

A container requires only the host operating system to provide basic services. The creation of a container and the act of running an application as a process in it is called containerization. A container is a single instantiation of a container image. A container image is a set of files that define the package of a container. To run an application in a container, all dependencies, such as libraries, binaries, middleware, and software components, must be included in the container image. [21]

Container adoption has increased in recent years. Containers provide a way to design applications using microservice architecture. The microservice architecture empowers companies to accelerate their development and strategies to scale their applications. The success of Docker, a container engine, has also popularized the concept of containers. [26]

Containers do not depend on special features, such as a hypervisor. Instead of that, the application runs as a binary inside a container and reuses the host kernel. A container runtime, such as containerd, can be used to create containers in an optimized manner by omitting some features. All of these technologies provide a way to standardize packaging software, improve the portability of software, simplify the maintainability of software, and enable the application to run in low-resource environments. [26]

Summary

Container-based abstractions are particularly attractive for some edge computing applications. They offer a very lightweight and portable method of building and deploying applications to edge servers and devices. Since the container approach does not use a guest operating system, it is naturally leaner and more efficient than traditional virtualization. This may be critical for resource-constrained edge devices. A container is also very portable and can be deployed in any environment and on nearly any host OS. [21]

Although a Type-2 hypervisor may appear similar to a container design, it is not analogous to containers. A Type-2 hypervisor still has high overhead and performance impact. When hypervisor and runtime services are taken into account, the amount of memory and processing required is much greater than that required for lightweight containers. [21]

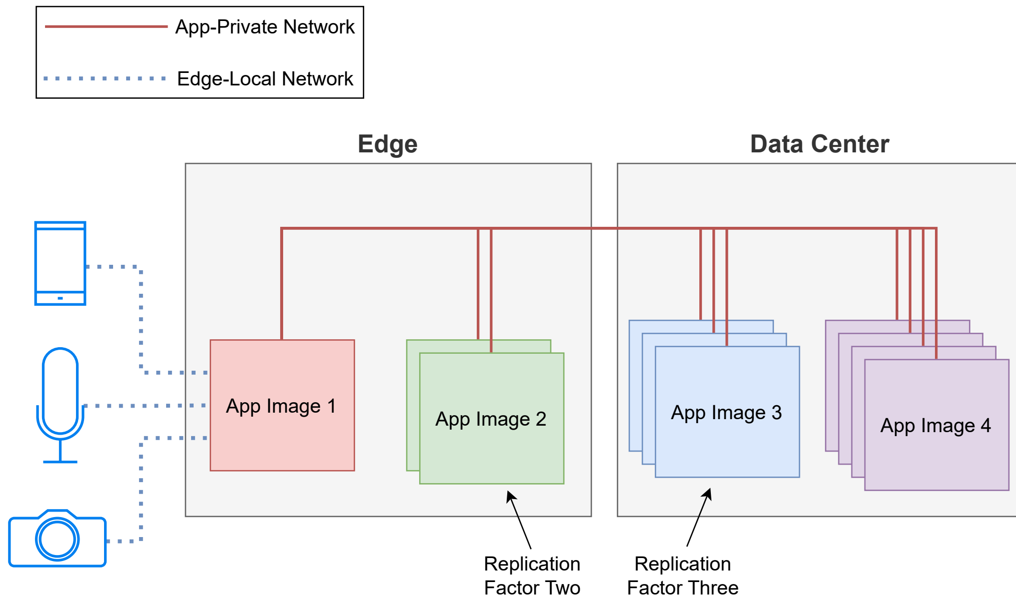


Figure 2.5: An anatomy of an edge application adapted from [10].

In practice, containers and virtual machines can be deployed together to provide additional layers of isolation and security for selected services by deploying containers on top of a virtual machine. Virtual machines and containers have benefits and downsides, and thus, one is not objectively better than the other. The choice of a correct service run-time environment depends on the requirements of an application. [2]

Both virtual machines and containers may be used in an edge cloud environment. However, lightweight containers fit edge computing applications very well, as resource requirements are usually limited, such as storage size and response time [2].

2.3.3 Edge Applications

Another important concept introduced by Hyunseok Chang *et al.* [10] is an edge application, referenced as an edge app. An edge app provides a service in the Edge Cloud. It is a bundled set of IaaS images that are designed to launch and work together in unison in the cloud and at the edge to provide a service. An edge app is composed of two types of virtual networks and two sets of compute instances. One set of compute instances runs in the cloud, and the other set runs in the edge. In each set, there can be different types of images, and each type of image can be deployed on multiple compute instances. A virtual network called the app-private network is instantiated for each edge app on startup to interconnect all instances belonging to the app. The communication of the edge app with end users and resources in the edge network is provided by the edge-local network, which bridges edge app instances running in the edge nodes to the local edge network. Figure 2.5 shows a sample layout of the edge application consisting of four images. Image 1 and Image 2 are used to realize the edge components with a replication factor of one and two, respectively, while Image 3 and Image 4 realize the data center components with a replication factor of three and four, respectively. All components are connected to the app-private network, while Image 1 is also connected to the edge-local network. [10]

Chapter 3

Demonstration Monitoring Application for Edge Cloud

This chapter introduces a demonstration application for an edge cloud environment. The goal of this chapter is to introduce a suitable application of an edge cloud environment, which highlights the benefits of the edge and the cloud working in unison.

As discussed in the previous chapter, an edge application is able to utilize the benefits of the cloud and the benefits of the edge to provide a valuable service that would otherwise not be possible using only one of the environments. This chapter designs such an application and subsequently describes its implementation details.

3.1 Design of the Application

To properly demonstrate the benefits of the edge cloud, a monitoring application was designed and implemented. The design of the high-level architecture is available in Figure 3.1. At the edge are located sensor applications that output raw monitoring data. These data are high-throughput, unfiltered, and unprocessed, thus containing unnecessary information and sensitive information. These sensors represent the data transmitted from the edge devices at the edge. An edge server collects all of this data for processing at the edge near its source. The edge server has larger processing and storage capabilities and, therefore, is capable of aggregating, filtering, and processing data. It is able to provide the processed data locally at the edge for real-time actions. The edge server also transmits the processed data to the cloud. A cloud server is then able to provide global decisions and utilize its massive processing and storage capabilities. Finally, the edge server provides all the processed data for additional context when performing real-time actions at the edge.

Apache Kafka

The demonstration application utilizes Apache Kafka[1] technology. To better understand the overall design, this section explains Apache Kafka and its core concepts.

Apache Kafka, also commonly called Kafka, is an event streaming platform. Kafka enables its users to publish, store, and process streams of events. It enables users to store the streams durably and reliably. It provides tools to process streams of events as they occur or retrospectively. These functionalities are delivered in a manner that is distributed,

[1] https://kafka.apache.org/

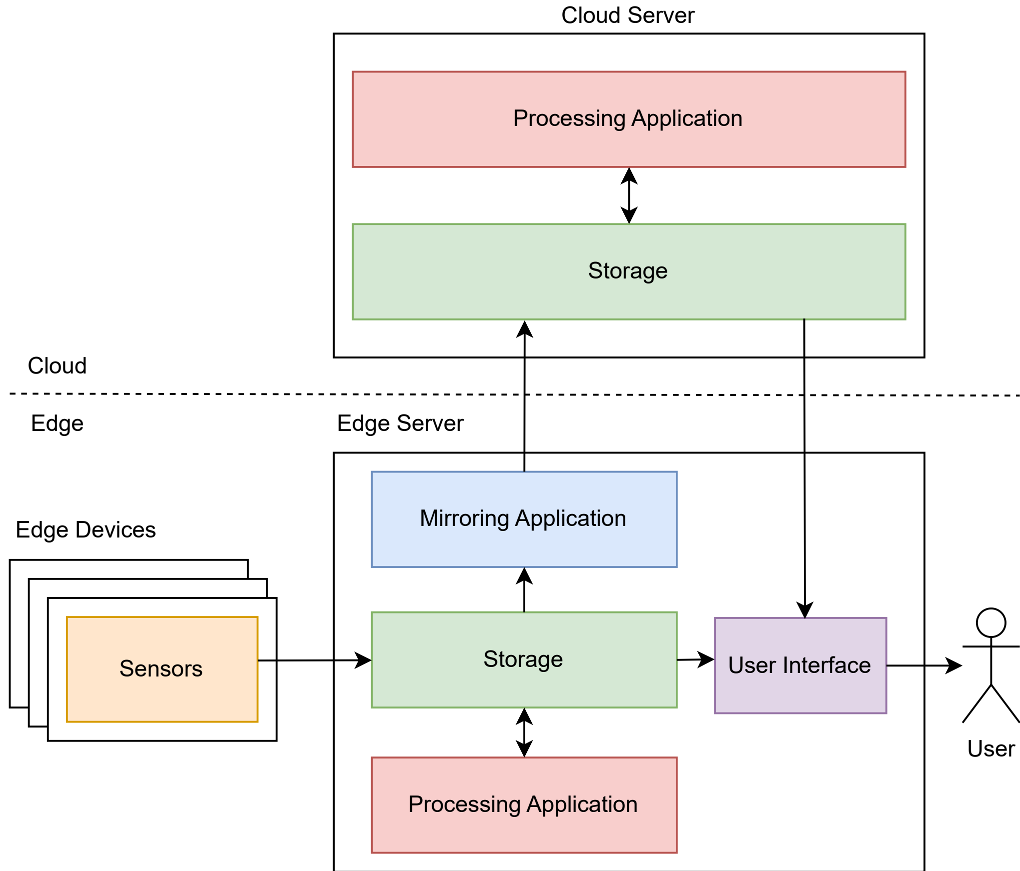


Figure 3.1: A high-level design of the overall architecture.

scalable, fault-tolerant, and secure. Kafka can be used, for example, to process payments, monitor shipments, and continuously capture and analyze telemetry in real time. [16]

Kafka is a distributed system consisting of servers and clients. Kafka runs as a cluster of one or more servers. Some of these servers, called brokers, form the storage layer. Other servers can continuously import and export data as event streams to integrate Kafka with existing systems. Clients enable developers to write distributed applications that can read, write, and process streams of events in parallel, at scale, and in a fault-tolerant manner. Clients can be categorized into two groups. Producers publish events, and consumers subscribe to groups of topics and subsequently consume these events. An event has a key, value, timestamp, and optional metadata headers. Partitioned topics are used to organize events and store them in a durable manner. [16]

From a Kafka perspective, the proposed demonstration application consists of two Kafka clusters. This is demonstrated in Figure 3.2. Sensors are simple Kafka publishers that create events that contain the monitoring data in a raw format. These data are stored at the edge on a local Kafka broker. A Kafka Streams application then processes this data and publishes the processed events in a new topic. This new topic is then mirrored into a different Kafka cluster, where further processing will be performed, and the newly created data is published into a new topic. Finally, all topics are consumed by a Kafka UI application that is able to display the events. By creating two Kafka clusters, the data is isolated, and computations are made in applicable places.

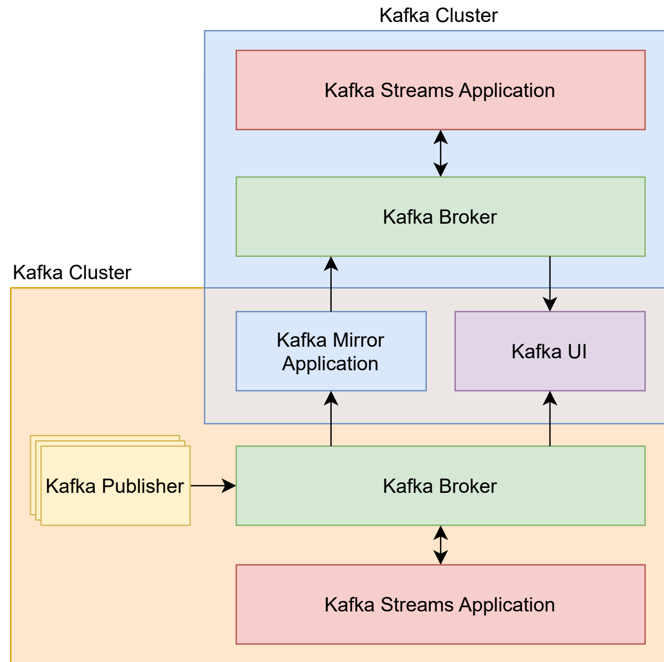


Figure 3.2: The application architecture represented as two Kafka clusters.

Computing at the Edge

Sensor publishers are designed to generate events at a rapid rate. This is to represent abstract unconfigurable sensors that generate high-throughput unprocessed data that contains unnecessary information. Using edge computing, generated events are aggregated over a specified duration, filtered from unimportant and sensitive information, and further processed to reduce their size. These processed events are then published to a new topic. The new topic can then be mirrored to the cloud for further processing. By this, the application can separate certain data. For example, the cloud may not be considered secure, and by filtering unimportant sensitive information, the application solves this problem. In addition, by reducing the size of events that are mirrored in the cloud, the network is less impacted by the transmitted events. Additionally, in the case the edge is disconnected from the cloud, local processing continues to work. This ensures gradual degradation of the service. Local consumers at the edge, such as the Kafka UI application, can still consume the new events generated at the edge to provide actions in real time.

Computing in the Cloud

The Kafka cluster in the cloud represents a Kafka cluster with larger processing and storage capabilities. On a mirrored topic from the edge, another processing is performed, and the events can be stored more durably and for longer than the edge is capable. As the edge filters sensitive information, in case a cloud is compromised, no sensitive information is stolen. The new topic that contains the processed events by the cloud can be consequently consumed by the edge server. From a cloud perspective, its strengths are utilized as much

as possible. Its strong processing and storage capabilities can be utilized, while the cloud's disadvantages, such as latency, are solved by the edge server.

3.2 Implementation of the Application

The demonstration application consists of multiple microservices. The microservices are containerized applications. This simplifies deployment and improves maintainability, especially since the application can be deployed on various machines. The benefits of using containerized applications are discussed in more depth in Section 2.3.2. Sensors and Kafka Streams applications were implemented using Kafka APIs, while the rest of the services are simply deployed using available container images from public image repositories.

The services implemented for this thesis, sensor application, and Kafka Streams applications, were developed using the Java programming language. Kafka currently provides its core APIs only for the programming languages Java and Scala. As Java has great performance and is widely used, its documentation and community guides allow for faster development that is less prone to errors. For these reasons, Java was chosen as the primary language for the applications.

Publishing Sensor Data

The sensor application utilizes the Kafka Producer[2] API. The configurable application connects to a Kafka cluster upon execution. The application generates events that contain a value in JSON format. This is to provide the events in a human-readable format without the need for further processing. The application generates the monitoring data using the `com.sun.management.OperatingSystemMXBean` interface from the `jdk.management` module. The interface provides information about the operating environment in which the Java virtual machine is running. These monitoring data are then encapsulated in an event, which is published in the Kafka cluster located at the edge. An example of a monitoring event is available in Listing 1. The values provided are expressed in bytes, nanoseconds, or percentages with high precision. These data are purposely published unprocessed and unfiltered to simulate generic unconfigurable sensors. The sensor application is then replicated to represent multiple sources of monitoring data.

Processing Events Using Kafka Streams Applications

The Kafka Streams application uses the Kafka Streams[3] library. The client library provides functions to process continuously updating data sets using operations such as transformations, aggregations, and windowed joins [17]. The Kafka Streams application located at the edge is thus able to process the events published by the sensors. An example of a processed monitoring event is available in Listing 2. The application aggregates events by their key that identifies specific sensors. This results in a lower frequency of these events in a new topic. These new events contain data in seconds, megabytes, and lower precision.

These processed events are then mirrored to the Kafka cluster in the cloud, where an additional Kafka Streams application further processes the data. It aggregates the events across all the sensors. This results in new events. An example is available in Listing 3.

[2] https://kafka.apache.org/documentation.html#producerapi
[3] https://kafka.apache.org/documentation/streams/

```json
{
    "name": "Linux",
    "arch": "amd64",
    "version": "5.15.133.1-microsoft-standard-WSL2",
    "availableProcessors": 8,
    "processCpuTime": 11950000000,
    "processCpuLoad": 0,
    "cpuLoad": 0.047058823529411764,
    "totalMemorySize": 12480131072,
    "freeMemorySize": 173137920,
    "totalSwapSpaceSize": 3221225472,
    "freeSwapSpaceSize": 3197726720,
    "committedVirtualMemorySize": 6647255040,
    "systemLoadAverage": 0.14
}
```

Listing 1: Example value of an event published at the edge by a sensor application.

```json
{
    "processCpuTime": 7.333,
    "processCpuLoad": 0.012,
    "cpuLoad": 0.06,
    "committedVirtualMemorySize": 6579,
    "freeSwapSpaceSize": 3148,
    "totalSwapSpaceSize": 3221,
    "freeMemorySize": 259,
    "totalMemorySize": 12480
}
```

Listing 2: Example value of an event published at the edge by a Kafka Streams application.

```json
{
    "processCpuTime": 15.454,
    "processCpuLoad": 0.004,
    "cpuLoad": 0.026,
    "freeSwapSpaceSizePercentage": 0.975,
    "freeMemorySizePercentage": 0.051,
    "status": "All is well"
}
```

Listing 3: Example value of an event published in the cloud by a Kafka Streams application.

Deployment of the Demonstration Application

The sensor and Kafka Streams applications mentioned in previous sections are technically various modes of a single application to simplify development. This application is then containerized using a written Dockerfile. Kafka broker servers are deployed using a widely used `bitnami/kafka`[4] container image of Apache Kafka that is maintained by VMware. The Kafka Mirroring application is deployed using the `bitnami/kafka` container image as well. Finally, the Kafka UI application is deployed using the `provectuslabs/kafka-ui`[5] container image from Provectus. As the overall demonstration application consists of multiple services, it can be easily deployed locally using the `docker-compose`[6] command to ensure that the overall application functions as expected.

[4] https://hub.docker.com/r/bitnami/kafka
[5] https://hub.docker.com/r/provectuslabs/kafka-ui
[6] https://docs.docker.com/compose/

Chapter 4

Managing Container Services in Edge Cloud

Software applications can rapidly evolve into highly complex constructs. The same applies to microservice architectures. One use case can lead to the creation of multiple microservices that communicate with other microservices, and moments later, a whole ecosystem is born. Containerization simplifies deployment and maintainability. However, it does not inherently solve other important aspects, such as granular decision making on deployment, service discovery, and configuration management. To reliably deploy, operate, manage, and monitor an edge cloud application consisting of multiple microservices deployed on various machines, an automated tool should be utilized. However, a wheel does not need to be reinvented. There are several technologies that solve various issues of container service orchestration. This chapter focuses on explaining some of the widely used technologies that are already being used in production.

4.1 Kubernetes

Kubernetes[1], originally developed by Google, is an open-source container orchestration system to manage containerized workloads and services. Kubernetes was able to gather more than 100,000 GitHub stars and has become a de facto container orchestration standard. Kubernetes enables developers to manage complex software architectures across multiple virtual or physical machines. Kubernetes provides features such as [35]:

- Service discovery and load balancing.
- Storage orchestration.
- Automated roll-outs and roll-backs.
- Secret and configuration management.
- Horizontal scaling.
- Extensible API.

[1] https://kubernetes.io

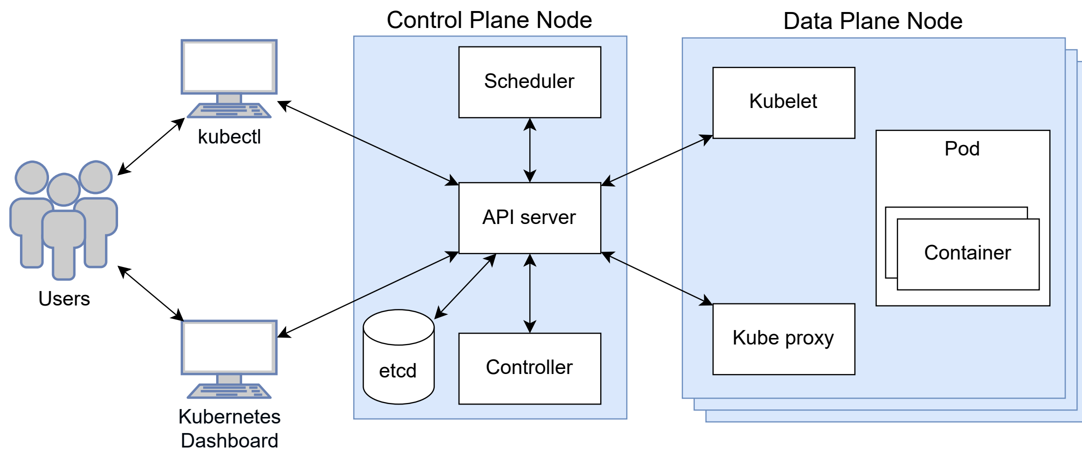


Figure 4.1: A diagram of the Kubernetes architecture adapted from [13].

Kubernetes is a powerful and complex system that is already being used by entities such as the US Department of Defense to enable development, security and operations on F-16s and battleships [9], OpenAI to launch and scale up experiments [7], BlackRock to accelerate rolling software into production [4], and CERN to process petabytes of data more efficiently [5]. Other notable companies are, for example, IBM [6] and Spotify [8].

A Kubernetes cluster consists of machines referred to as nodes that run containerized applications. A Kubernetes cluster consists of two planes that consist of a subset of nodes. The control plane and the data plane. The control plane makes global decisions about the cluster and reacts to events in the cluster. User applications are deployed on the data plane nodes. To achieve high availability and fault tolerance, control plane components can be run on multiple machines. Some of the most important control plane components are the API server that exposes the Kubernetes API, the backing store for all cluster data, etcd, the Kubernetes scheduler that selects nodes for workload, and the controller manager that manages controllers responsible for watching and responding to events in the cluster. [36]

The important components of a node are kubelet, kube-proxy, and the chosen container runtime. Kubelet's main responsibility is to make sure that the desired containers are running and healthy. Kube-proxy is a network proxy that maintains network rules on nodes to allow network communication to containers from within or outside the cluster. Finally, the container runtime is responsible for the life cycle of containers. [36]

The Kubernetes API server enables users to manage Kubernetes API objects. Some important Kubernetes objects worth mentioning for the thesis are namespaces, deployments, and pods. Namespaces can be used to isolate resources to ensure isolation between projects, environments, or teams. A pod represents a set of running containers, and deployments are used to manage replicated applications [36]. A simplified diagram of a Kubernetes architecture is available in Figure 4.1.

A Kubernetes cluster can also be easily extended. New infrastructure components can be created, and the Kubernetes API itself can be greatly extended. This introduces concepts such as infrastructure as code. For example, the goal of the HyperShift[2] project is to host the control planes of different Kubernetes clusters within a single central cluster called the

[2] https://github.com/openshift/hypershift

management cluster. An already powerful tool with the extensibility of its API and a strong community makes Kubernetes a great choice for numerous use cases. However, due to the rich Kubernetes functionality and its complex architecture, it may not be ideal for resource-constrained environments. When creating a Kubernetes cluster using the kubeadm tool, its minimum system requirements start at 2 GiB of RAM per machine and 2 CPUs per control plane machine [37]. This may not be ideal for edge environments, as they tend to have fewer processing and storage capabilities, and deployment of a Kubernetes cluster at such an edge may not be possible.

4.2 Lightweight Kubernetes Distributions

As Kubernetes, also called K8s, has started to be more and more used in edge environments, the problem of its complex architecture, installation, and system requirements has become apparent. This has allowed lightweight Kubernetes distributions to grow and become popular. This section briefly introduces some of the most popular K8s distributions. A brief summary of the distributions mentioned is available in Table 4.1.

4.2.1 K0s

An open source Kubernetes distribution, k0s[3], contains all the required features and dependencies to build a Kubernetes cluster in minutes. The k0s is distributed as a single binary with zero host dependencies. The distribution offers a number of key features, such as multiple installation methods and low resource utilization requirements. [14]

The minimum system requirements for k0s are a 1 vCPU and 1 GB of memory. As an open-source project, it has approximately 5,000 GitHub stars, making it one of the more popular Kubernetes distributions. [18]

4.2.2 MicroK8s

An open source system, MicroK8s[4], provides the functionality of the core Kubernetes components with a smaller resource utilization, scalable from a single node to highly available clusters. The system includes a plethora of extensions that provide additional features. [23]

MicroK8s is distributed as a snap package and requires as little as 540 MB of memory. The project ranks as one of the more popular systems based on Kubernetes, with its approximately 8,000 GitHub stars. [18]

4.2.3 K3s

An open source project, K3s[5], provides a lightweight Kubernetes distribution. K3s includes the required dependencies for a simple installation and provides additional installable services. It is distributed either as a single binary or as a minimal container image. [15]

As an open-source project, it was able to gather more than 25,000 GitHub stars, making it one of the most popular lightweight Kubernetes distributions. The minimum system requirements of K3s are 1 CPU and 512 MB of memory. [18]

[3] https://k0sproject.io/
[4] https://microk8s.io/
[5] https://k3s.io/

	k0s	**MicroK8s**	**K3s**
Key Developer	Mirantis	Canonical	Rancher / SuSE
CNCF Certified	Yes	Yes	Yes
GitHub Stars	~5,000	~8,000	~25,000
Operating System	Linux	Ubuntu, Linux, Windows, MacOS	Linux
CPU Architecture	x86-64, ARM64, ARMv7	x86, ARM64, s390x, Power9	x86, ARM64, ARMhf
Deployment	Single Binary	Snap Package	Single Binary
Container Runtime	containerd, custom	containerd, kata	containerd, docker, custom
Container Network Interface	Kube-Router, Calico, custom	Calico, Flannel	Flannel, custom
Control Plane Datastore	etcd, SQLite, PostgreSQL, custom	dqlite, custom	SQLite, PostgreSQL, MySQL, MariaDB, etcd
Minimum CPU Requirement	1 vCPU	N/A	1 CPU
Minimum RAM Requirement	510 MB	540 MB	512 MB

Table 4.1: Summary of lightweight Kubernetes distributions adapted from [18].

4.3 KubeEdge

KubeEdge[6], introduced by Ying Xiong *et al.* [41], takes a slightly different approach using Kubernetes technology. Lightweight Kubernetes distributions are suitable for deployment at the resource-constrained edge. However, as more edges are added to an architecture and thus multiple clusters are deployed at various locations, a multicluster architecture is born, which may need its own tooling, such as multicluster management solutions and a service mesh across the clusters. The goal of KubeEdge is seamless cloud-to-edge coordination.

KubeEdge is an open source project, built on top of Kubernetes, that extends native containerized application orchestration and device management to hosts at the edge. KubeEdge provides core infrastructure support for networking, application deployment, and metadata synchronization between the cloud and the edge. It also allows developers to write custom logic and enable communication for devices at the edge. [19]

It is worth highlighting that KubeEdge extends a Kubernetes cluster and is not a Kubernetes distribution. That is, KubeEdge extends an existing Kubernetes cluster with edge cloud capabilities. A diagram of its architecture is available in Figure 4.2. The architecture consists of the following components [19]:

- EdgeController: A controller that manages edge nodes and pods metadata.
- DeviceController: A controller that manages devices so that the device metadata and status can be synchronized between the edge and the cloud.
- CloudHub: A server that monitors changes in the cloud. It caches and sends messages to the EdgeHub component.
- EdgeHub: A client interacting with the cloud regarding the edge. This involves synchronizing updates of resources from the cloud to the edge and relaying changes in the status of hosts and devices at the edge to the cloud.

[6] https://kubeedge.io/

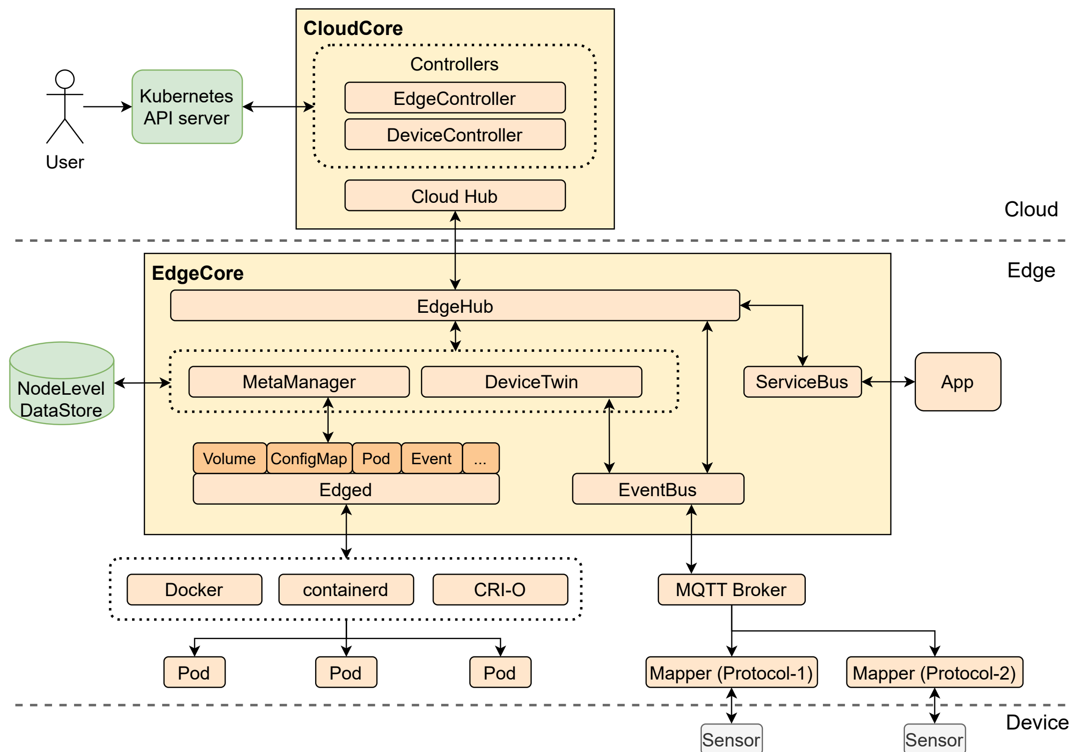


Figure 4.2: The KubeEdge architecture adapted from [19].

- MetaManager: A message processor between Edged and EdgeHub. It retrieves and stores metadata from and to a database.
- Edged: An agent running on edge nodes that manages containerized applications.
- DeviceTwin: A component responsible for storing the status of a device and synchronizing the status of the device with the cloud. It also provides query interfaces.
- EventBus: An MQTT client interacting with MQTT servers, offering publish and subscribe capabilities to other components.
- ServiceBus: An HTTP client offering HTTP client capabilities to components in the cloud to reach HTTP servers running at the edge.

4.4 Nomad

A popular alternative to Kubernetes is Nomad[7]. Nomad aims to be a simple workload orchestrator, as Kubernetes tends to be perceived as a complex system that has a steep learning curve. Nomad strives for simplicity and composition.

Kubernetes and Nomad have overlapping functionalities in terms of application deployment and management, but the technologies exhibit distinct differences in certain aspects.

[7] https://www.nomadproject.io/

Kubernetes is designed to provide the needed features for running Linux containerized applications, encompassing cluster management, scheduling, service discovery, monitoring, managing secrets, and more. Nomad, on the other hand, aims to focus solely on cluster management and scheduling. Nomad is designed to have a small scope and utilize tools such as Consul[8] for service discovery, and Vault[9] for secret management. [27]

The following are some of the key benefits of using Nomad instead of Kubernetes [27]:

- Simplicity: The Kubernetes architecture consists of multiple interoperating services. Nomad is architecturally simpler as it is provided as a single binary for both servers and clients. Nomad does not require external services for coordination or storage. It combines a resource manager and a scheduler into a single system.
- Support for a greater range of types of workload: Kubernetes is specifically designed for containers, while Nomad is a more general solution. Nomad supports virtualized, containerized, and standalone applications.
- Simple deployment: Installation of a production Kubernetes cluster is a complex and time-consuming task, which may lead to inconsistencies in capabilities and configuration. Nomad claims to be a lightweight binary that can be deployed locally, in production, at the edge, or in the cloud in a simple and consistent manner.
- Scalability: Kubernetes supports clusters up to 5,000 nodes and 300,000 containers [38]. These numbers are especially important when dealing with an edge cloud environment that encompasses multiple edge zones. Nomad has been proven to support up to 10,000 nodes and 2 million containers.

4.5 Summary

This chapter's goal was to explore various technologies for deploying, operating, and managing in an edge cloud environment. It presented complex and powerful Kubernetes and its lightweight distributions, ideal for the edge. It explored the KubeEdge that enables cluster administrators to extend a Kubernetes cluster to the edge in a seamless way with additional capabilities. Finally, it also described the Nomad system, a popular alternative to Kubernetes. However, the best container orchestration system simply does not exist. These tools exist simply to provide a platform for their workload. The decision to select a particular technology ultimately depends on the characteristics of the workload, the required infrastructure features, and finding a balance between simplicity and complexity.

[8] https://www.consul.io/

[9] https://www.vaultproject.io/

Chapter 5

Analysis of Edge Cloud Cluster Deployment

Given that a user has written their application consisting of containerized microservices and has read about existing container management technologies, only the last step remains. The actual deployment of the services using a chosen technology. The step may seem straightforward; however, it consists of provisioning infrastructure, configuring machines, deploying the management technology, and finally, using the technology to deploy the services. The previous chapter explored the management technologies but did not focus on the deployment of the technologies, which may play a significant role when choosing a technology. To even experiment with Kubernetes, K3s, or Nomad, a cluster administrator must first deploy the technologies. The purpose of this chapter is to analyze existing methods for edge cloud cluster deployments and describe their process from a high level. Using the knowledge, the chapter then describes the current problems for edge cloud cluster administrators using existing tools and proposes a solution to address the mentioned challenges.

5.1 Existing Deployments Tools

The existing deployment tools may be categorized into three groups: manual, automated, and managed. Each group provides a different set of benefits and downsides.

5.1.1 Manual Deployments

Manual deployment tools provide the most flexibility upon cluster deployment. This is achieved by manually installing technology on a given machine. Although this approach provides the most flexibility, it also requires the most technical knowledge and manual effort. A manual process commonly consists of three main stages.

- Preparation of machines
- Creation of the initial control plane node
- Addition of the remaining nodes to the cluster

The complexity of the stages differs greatly between the individual technologies. For example, when installing a classic Kubernetes cluster using the kubeadm[1] deployment tool, the process may be broken down into the following steps.

1. Access a machine
2. Modify the machine's networking rules
3. Install Kubernetes installation tools
4. Install and configure a container runtime
5. Initialize the machine to be a control plane node
6. Install a container network interface plugin
7. Export administrator credentials
8. Export a token to join additional nodes to the control plane node
9. Repeat steps one through four for all remaining nodes
10. Using the acquired token, join the remaining nodes to the initial control plane node

This process may become cumbersome when a cluster consists of multiple nodes or multiple clusters are required. This hands-on process demystifies cluster creation, is beneficial for learning fundamental cluster components, and provides the most flexible deployment. However, this process is not scalable, is prone to manual errors, and requires a system administrator's technical knowledge. This is not an issue, as the purpose of the kubeadm tool is to bootstrap a Kubernetes cluster and nothing more. It is expected that a higher-level, more tailored tooling is built on top of the kubeadm tool [39]. Automated tooling comes in to provide a simpler automated cluster deployment, some even using kubeadm underneath.

Other technologies provide similar tools, such as kubeadm. However, the tools vary in the provided functionalities and ease of use. For example, the K3s and MicroK8s projects try to provide even more value by simplifying the creation of clusters by requiring minimal prerequisite steps on hosts. The K3s project even promotes this feature with the known phrase „batteries included" with respect to its cluster creation and needed dependencies [15]. The projects may configure the machines and install a container runtime and a container network interface plugin by themselves. This greatly simplifies the deployment. These tools are also able to install additional helpful services, something that kubeadm does not strive to do. In combination with community projects such as k3sup[2], an administrator does not need to access a node manually and install K3s. The administrator may designate a node using the k3sup utility from a local machine to transform the accessible machine into a K3s node, further simplifying the cluster creation process. An example of such a simple two-node cluster creation is available in Listing 4.

However, technologies such as KubeEdge may require a more complex setup. As explained in Section 4.3, KubeEdge extends a Kubernetes cluster. That is, a cluster already installed is needed. KubeEdge cloud capabilities can be installed using the Helm[3] package

[1] https://kubernetes.io/docs/reference/setup-tools/kubeadm/
[2] https://github.com/alexellis/k3sup
[3] https://helm.sh/

```
$ k3sup install --ip $IP_SERVER
$ k3sup join --ip $IP_AGENT --server-ip $IP_SERVER
```

Listing 4: An example of creating a two-node K3s cluster using the k3sup command.

```
node1 ansible_host=10.33.0.11
node2 ansible_host=10.34.0.12
node3 ansible_host=10.35.0.13

[kube_control_plane]
node1

[etcd]
node1

[kube_node]
node2
node3
```

Listing 5: An example of an inventory file for a three-node cluster using Kubespray.

manager for Kubernetes, and new edge nodes are connected to the cluster using the keadm tool, a KubeEdge tool similar to the kubeadm tool. An additional network solution for edge nodes may also be needed. In such cases, EdgeMesh[4] is a popular solution due to its simple addition to a KubeEdge cluster using Helm. However, additional tools mean additional configurations and moving parts, increasing the overall complexity.

Manual deployment tools provide a flexible method to create clusters. Automated deployment tools solve the problem of complexity, scalability, and automatization.

5.1.2 Automated Deployments

To further delegate the task of cluster creation, multiple tools are available. Given that an administrator has an existing infrastructure and wants to delegate the responsibility of creating a Kubernetes cluster to a tool, the Kubespray[5] project is widely popular.

Using Kubespray, a cluster administrator can simply define an inventory file of hosts and their roles, such as the control plane, and Kubespray takes the responsibility of setting up the cluster. Minimal knowledge of the IT automation engine Ansible[6] is required, as the project uses the technology underneath to configure the cluster. For example, by creating a simple inventory file such as one provided in Listing 5 and running a single command, a three-node cluster is created.

[4] https://edgemesh.netlify.app/
[5] https://github.com/kubernetes-sigs/kubespray
[6] https://github.com/ansible/ansible

Kubernetes Operations[7] (kOps) further automates cluster creation by enabling a cluster administrator to also provision the needed infrastructure using its capabilities. As such, a cluster administrator only provides cloud credentials and cluster configuration, and a cluster is created from scratch using only a single command.

Another rather newer project is the Cluster API[8]. Using a declarative Kubernetes API, a Kubernetes cluster administrator can simply define a cluster using a Kubernetes resource. This means that an initial Kubernetes cluster is needed to utilize the Cluster API to create new clusters. However, such an API simplifies and automates an entire cluster life cycle. Provisioning of infrastructure, cluster creation, scaling, upgrading, and destruction of Kubernetes-conformant clusters using a declarative API. Cluster API aims to define common cluster operations, provide a default implementation, and provide the ability to swap out implementations, which is an important aspect. Using this model, the Cluster API can be extended to support any infrastructure provider or bootstrap provider. [32]

As of Cluster API version 1.18, some of the available bootstrap providers are Kubeadm, MicroK8s, Talos, K3s, and K0s [33]. As such, the Cluster API may be used to create clusters of some of the Kubernetes technologies mentioned in Chapter 4. However, technologies such as Nomad, which are not based on Kubernetes, are not supported by the Cluster API due to their inherently different architecture.

Automated deployments enable administrators to easily create Kubernetes-conformant clusters. Some of the technologies support infrastructure provisioning, and some even support the entire life cycle of a cluster. These tools are commonly created for more popular and widely used technologies, as their development and maintenance require active contributions. An important benefit of this approach is that the infrastructure and the cluster itself are still managed by the cluster administrator; the infrastructure and clusters are created and managed by the administrator. These clusters are commonly referred to as self-managed clusters, as the administrator self-manages the clusters. Another method of creating clusters further simplifies the deployment and maintenance by reducing the deployment to even a single click and delegating maintenance to experts; this can be achieved by clusters managed by a provider.

5.1.3 Managed Clusters

As described in Chapter 2.1, cloud computing provides a wide range of service models. From infrastructure as a service to platform and software as a service. Another method of using cloud edge technologies is managed clusters, where, for example, managed production-ready Kubernetes clusters are provided as a service to customers. The following are some of the most popular Kubernetes service offerings provided by the largest cloud providers.

- Amazon Elastic Kubernetes Service[9]
- Google Kubernetes Engine[10]
- Azure Kubernetes Service[11]

[7] https://github.com/kubernetes/kops
[8] https://github.com/kubernetes-sigs/cluster-api
[9] https://aws.amazon.com/eks/
[10] https://cloud.google.com/kubernetes-engine
[11] https://azure.microsoft.com/en-us/products/kubernetes-service

Using a managed cluster, an administrator obtains a control plane in which the management, availability, and scalability aspects are managed by the service provider. An administrator can then simply attach the data plane nodes. Using such a managed service significantly reduces engineering efforts regarding the life cycle of the cluster, as the deployment and management of a cluster require significant technical efforts. Providers also provide a more seamless integration of their additional services to such clusters. [28]

However, simple deployment and maintenance are tied to the dependency on the service provider. Aspects such as provider-imposed restrictions and trust in provider computing infrastructure may play a role in decision-making regarding this deployment approach [28].

5.2 Challenges of Edge Cloud Cluster Deployment

The number of existing edge cloud technologies can become overwhelming, and each edge cloud environment is unique, as are its requirements. Thus, hands-on experimentation with technology is necessary in order to test deployment, maintenance, and usage and to analyze performance impact on machines and applications, among others.

Section 5.1 has explored the various deployment methods of such clusters. The deployment is not a straightforward exercise, and this fact can discourage cluster administrators from experimenting with different technologies.

The simplest deployment and management using managed clusters tend to be supported by providers only for the most popular technologies, such as classic Kubernetes. Other technologies, such as Nomad or K3s, do not receive the same attention. Automated deployment simplifies cluster creation; however, projects such as Kubespray and kOps tend to focus on an individual technology or a subset of technologies. Therefore, a cluster administrator who wishes to experiment and evaluate individual technologies is forced to try a number of different deployment technologies whose technical requirements and functionalities vary greatly. Using manual deployment tools requires the most engineering efforts, using automated tools requires the use of a number of different technologies, and managed clusters simply exist for the most desired technologies.

A cluster administrator may simply choose a technology and adapt. However, such clusters may end up running for years and years with limited acceptable downtime. Deployment of such clusters is also only the beginning of a cluster's life cycle, as maintenance, including updates, is a critical part of cluster administration. Thus, the choice should be carefully considered by the acting parties.

Therefore, the tooling for creating such clusters becomes greatly valuable. Easily deploying and managing clusters lowers the entry barrier for cluster administrators to fully experiment with respective technologies before fully committing. As can be understood from the existing deployment tools explored in Section 5.1, tools for deploying such technologies are complex, require the expertise of large communities, and most of the time focus on a subset of technologies. Implementing a universal automated tooling capable of deploying every mentioned edge cloud technology on a similar level of configurability and supportability of existing tools using a unified interface would be a daunting task, at least.

However, a tool for the deployment of such clusters using a unified interface with limited capacity would bring significant value, as it would allow administrators to easily deploy a cluster of chosen technology. Even though some administrators may understandably prefer to use an already established tooling supported by a large community of active contributors for production clusters, a tooling that lowers the entry barrier by allowing administrators to quickly and easily create edge cloud clusters of different technologies for further exper-

imentation and evaluation could help them to make a more informed decision regarding the choice of the management technology. A more informed decision means that the administrator is less likely to migrate the running applications to a cluster using a different technology in the future, which would mean the integration of a different deployment tooling, new training of cluster administrators, and migration of all applications. By lowering the bar, administrators are more likely to experiment with the technologies and thus make a more informed decision. A tool to create edge cloud clusters using a unified interface would enable cluster administrators to easily experiment with, utilize, and evaluate edge cloud technologies.

5.3 Addressing the Challenges

To address the mentioned challenges, the thesis proposes implementing tooling for edge cloud cluster deployment, usage, and monitoring using a unified interface. A tooling to enable cluster administrators, developers, and students to easily deploy and monitor edge cloud services. A tooling to:

- Provision required IT resources
- Deploy edge cloud technology
- Deploy and monitor services in the cluster
- Monitor metrics of the cluster

The thesis also proposes to leverage the tooling for a comparison of edge cloud technologies; a comparison of the difficulty of deployment, usage, and performance impact to provide additional value for cluster administrators in determining the appropriate technology. Manual deployment methods, which were discussed in Section 5.1, will be utilized in the tooling to provide comparable results, since there are manual deployment tools for every technology as opposed to automated deployments and managed clusters.

To achieve these goals, a project needs to be defined, designed, implemented, and results evaluated. The following sections are intended to outline the project functionalities and provide a high-level design by focusing on the needed areas and components.

5.3.1 Technical Requirements

The project, due to its complexity and the number of components, must comply with the following technical requirements.

- The option of provisioning infrastructure using a cloud provider must be provided for repeatability and possible local hardware constraints.
- A tool will be implemented to deploy a specified edge cloud management technology. Cluster deployment must be automated to achieve the repeatability needed for development, testing, and performance comparison.
- Tools to deploy and monitor services in created clusters will be implemented.
- Automated end-to-end testing must be implemented as the project will utilize a number of different services and different configurations.

- The project will be published as an open-source project, and thus, continuous integration is paramount.
- To compare the performance impact of technologies, persistent storage must be used to collect metrics from a number of clusters over a period of time.

5.3.2 Core Areas of the Project

The overall project can be broken down into three main areas. Cluster deployment, application deployment, and testing and monitoring. These areas are independent of each other and may be broken down further. The areas may also be implemented independently of each other, improving development. A simplified schema of the project is available in Figure 5.1, which highlights the primary areas together with their respective internal components.

Cluster Deployment

The cluster deployment part of the project aims to provide tools to easily deploy clusters that utilize one of the edge cloud technologies. The tools will be able to provision the infrastructure based on the configuration provided by a user. Afterward, the specified edge cloud technology is to be installed on the cluster.

Application Deployment and Monitoring

Given an existing cluster, application deployment and monitoring tools will provide users with a way to deploy user-specified applications on specified nodes and monitor them.

Testing and Monitoring

The final area of the project focuses on the testing of the project and the collection of cluster metrics. The demonstration application designed and implemented in Chapter 3 will be deployed in the created cluster using the implemented tools to verify the overall functionality of the project. The application will be tested to verify that its functionalities work as expected. This part is crucial to ensure that all the preceding steps have been successfully executed for every edge cloud technology. Cluster metrics must be collected and exported to persistent storage. The metrics will be analyzed during the comparison of the edge cloud technologies.

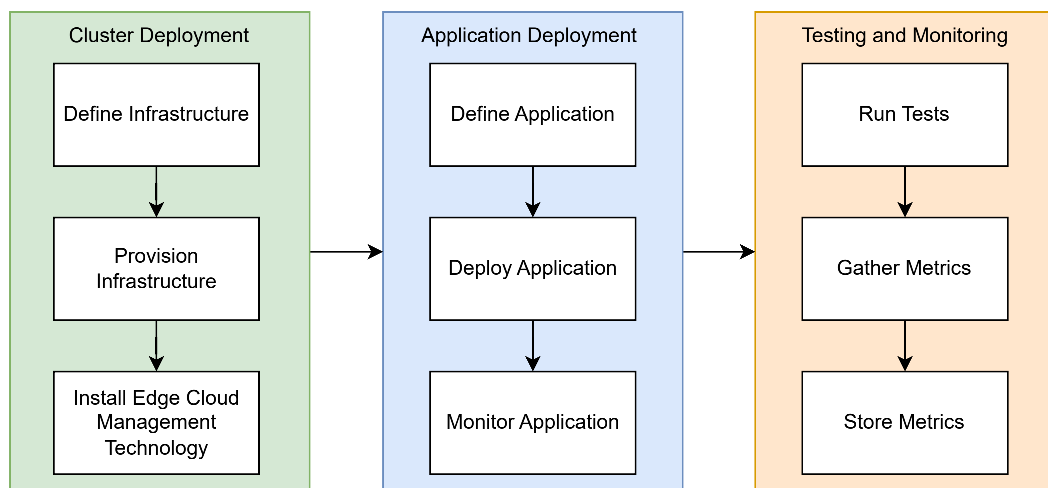


Figure 5.1: Schema of the project for deployment and monitoring of edge cloud services.

Chapter 6

Design of Edge Cloud Services Deployment Project

This chapter aims to design the project for the deployment of edge cloud services, which was proposed in Section 5.3 to solve the challenges of edge cloud services deployment. The section has outlined the main functionalities, requirements, and areas. This chapter aims to define the outlines in a clearer and more actionable way by providing a design for the overall project and its components, which can then be implemented. The chapter is broken down into the main areas of the project, where each section aims to provide a solution for the respective area of problems.

6.1 Cluster Deployment

The cluster deployment area of the project focuses on the provisioning of IT resources, their configuration, and the deployment of edge cloud management technology. These problems can be categorized into two independent areas, resulting in two components.

6.1.1 Provisioning of Infrastructure

Every edge cloud environment is unique, as is its infrastructure. It is outside the project's scope to support every possible infrastructure permutation. The goal is to provide a unified interface for a configurable edge cloud cluster deployment with limited scope. Based on the input configuration specified by a user, the IT resources needed will be provisioned by a cloud provider. Four types of nodes will be supported.

- **Control plane**: A control plane node hosts components that manage the state of the cluster. For example, the Kubernetes API server.
- **Cloud**: These nodes will represent nodes that will host the user workload application in the cloud. For example, the cloud services of the demonstration monitoring application, which is described in Chapter 3.
- **Edge**: These nodes will represent nodes that will host the user workload application at the edge. For example, the sensors and edge services of the demonstration monitoring application, which is described in Chapter 3.

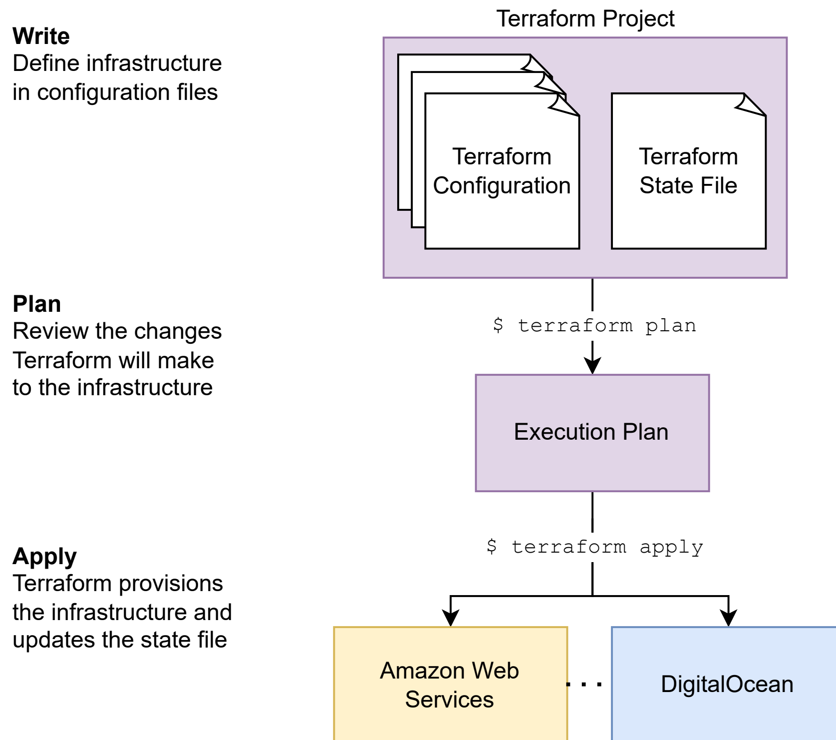


Figure 6.1: Visualized Terraform stages adapted from [40].

- **Infrastructure**: Infrastructure nodes will be used to host important infrastructure applications that should not interfere with the control plane but should not be affected by workload applications. For example, monitoring and security solutions.

To achieve this, a Terraform project will be created to provision the desired infrastructure based on a user configuration, specifying the desired nodes and their types.

Terraform

A tool for building, modifying, and versioning cloud and on-premise resources, Terraform utilizes the infrastructure as code concept. Using human-readable configuration files, which can be versioned, reused, and shared, administrators can provision and manage their infrastructure throughout its life cycle, from low-level resources such as computing, storage, and networking resources to high-level components such as DNS. [40]

Terraform manages resources through the application programming interface (API) of a platform. Terraform providers enable users to work with any platform with an accessible API. There are thousands of written Terraform providers to manage many different types of resources and services, including resources from providers such as Amazon Web Services, Azure, Google Cloud Platform, and DigitalOcean. [40]

The core Terraform workflow consists of the following three stages, which are visualized in Figure 6.1. The write stage allows users to define the needed resources, which may

be across multiple cloud providers and services. In the planning stage, Terraform creates an execution plan describing the resources it plans to create, modify, or destroy based on existing resources and configuration. The final stage, apply, executes the plan. [40]

Terraform Project

By specifying input variables of the Terraform project that will represent nodes and their types, the project will be able to provision the nodes and any additional infrastructure using a cloud provider. After all IT resources are provisioned, a Terraform state file, which represents the provisioned infrastructure, will be available to a user.

6.1.2 Installation of Edge Cloud Technology

Given an existing infrastructure, installation of the edge cloud management technology will be performed. Using manual deployment tools often requires a prior configuration of nodes and the following execution of a respective deployment tool on the nodes. To achieve this in a consistent, repeatable, and automated manner, an Ansible project will be created.

Ansible

Automation technology to automate virtually any task, using Ansible, developers may manage system configuration, deploy software, perform updates, eliminate repetition, simplify workflows, and much more. Ansible uses human-readable scripts called playbooks to automate tasks on a local or remote system. A user defines the desired state of a system, and Ansible is responsible for ensuring that the desired state is met. [30]

The core concepts of Ansible relevant to the thesis are visualized in Figure 6.2. The components are described as follows [31]:

- Control node: The machine from which the Ansible command line interface tools are run. It may be any machine that meets the software requirements as a control node.
- Managed nodes: The target devices that are managed by a control node. Ansible does not even have to be installed on managed nodes. Managed nodes are also known simply as hosts.
- Inventory: A list of the managed nodes, which may contain additional information for each node, such as an IP address of the node, an assigned group, and custom variables later utilized when running Ansible.
- Playbooks: A playbook is written in the YAML format, which makes playbooks easy to read and write. A playbook consists of units called plays that map managed nodes to tasks. A play consists of variables, roles, and tasks. A role is reusable Ansible content. A task is an action to be performed on a managed node.

Ansible Project

By specifying an inventory file and a playbook representing edge cloud technology, the Ansible project will configure the nodes and install the needed software to create the desired cluster. A user's machine will represent the control node, and managed nodes will be any machines specified by the user or the machines provisioned by the Terraform project. Due

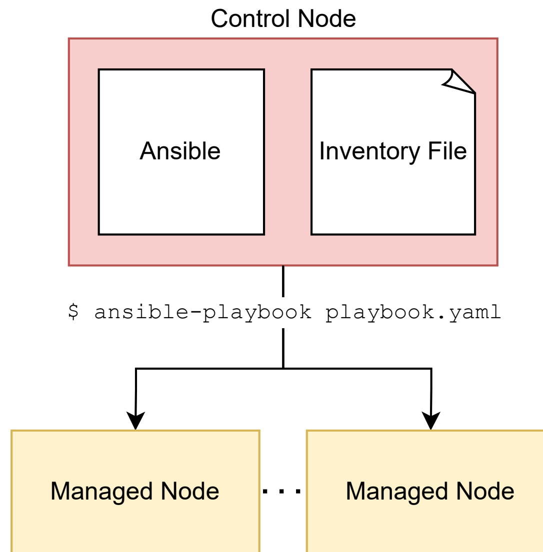


Figure 6.2: Visualization of main Ansible components adapted from [29].

to the scope constraints of the thesis, the project will focus on Kubernetes technologies, the technologies being classic Kubernetes, K3s, MicroK8s, and KubeEdge. These technologies were chosen because they represent the major different types of clusters: classic Kubernetes distribution, lightweight distributions, and the extension of a Kubernetes cluster.

By specifying an inventory file, a chosen playbook, and any additional input variables, the project will be able to create the desired cluster. Afterward, a file that enables the user to access a cluster using the Kubernetes API server will be made available to the user.

6.2 Services Deployment and Monitoring

For the deployment and monitoring of services, a command-line interface (CLI) application will be implemented. A core component of the edge cloud technologies that will be explored is the Kubernetes API server. A CLI application will be written to communicate with the API server to deploy and monitor services. Kubernetes already provides a much more extensive CLI application, called kubectl[1] to communicate with the API server; however, one of the goals of the thesis is also to analyze the usage of the management tooling. This includes the usage of the API server as writing tooling that is able to communicate with the API server if often required for further automation.

For deployment, a user will be able to provide Kubernetes manifest files that describe the desired state of Kubernetes objects that are to be applied in the cluster, and the CLI application will communicate the specifications to the API server.

To monitor the services, multiple solutions could be considered. The thesis takes inspiration from the kubectl tool, as it provides a simple, straightforward solution that is

[1] https://kubernetes.io/docs/reference/kubectl/

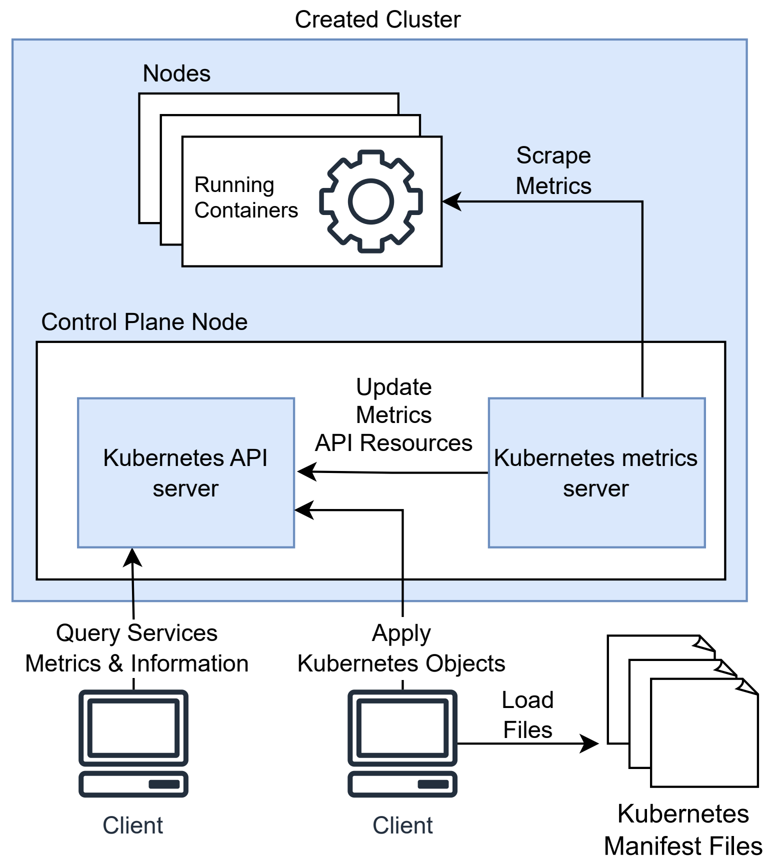


Figure 6.3: An overview of deployment and monitoring of services.

commonly used. The CLI application will utilize the Kubernetes metrics API to collect metrics regarding services and nodes. The API is supported by the Kubernetes metrics server[2], which must be installed. The CLI application will fetch the Kubernetes API server with respect to the metrics API to receive information regarding the resource utilization of the running services and nodes. The relationship between the servers and the CLI application is visualized in Figure 6.3.

6.3 Cluster Metrics Collection

To compare the performance impact of technologies on services and nodes, metrics must first be collected and stored. The metrics API utilized in the CLI for service deployment and monitoring could be considered; however, the API provides resource utilization at a current point in time. Thus, additional processing would have to be done. The Prometheus[3] monitoring solution will be used to collect metrics over a period of time.

[2] https://github.com/kubernetes-sigs/metrics-server
[3] https://prometheus.io/

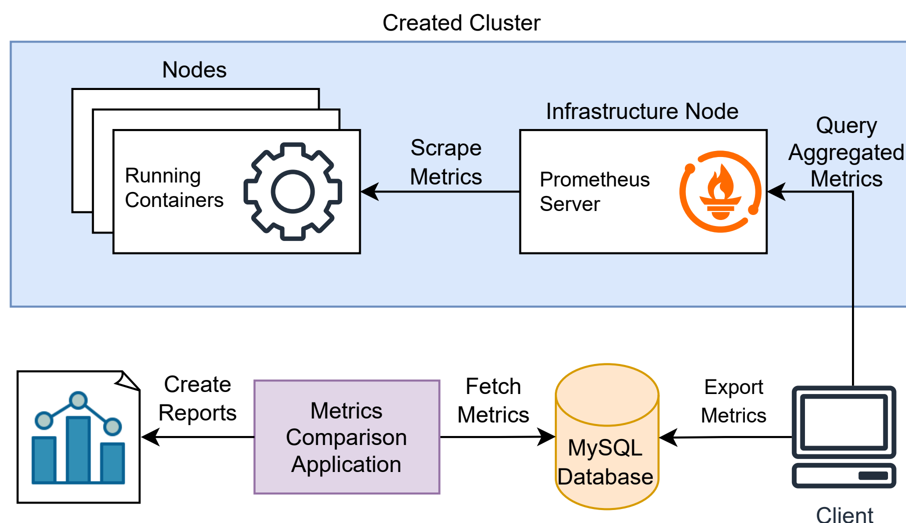


Figure 6.4: An overview of cluster metrics collection and visualization.

The Prometheus technology was chosen for the collection of metrics because of its simple deployment and configuration. In order not to interfere with the control plane or workload services while collecting metrics, the Prometheus server will be deployed on an infrastructure node. The Prometheus server collects metrics that might be subsequently queried using PromQL (Prometheus Query Language). The metrics will need to be stored in persistent storage outside the created clusters to ensure the data can be compared later. There are multiple solutions for long-term storage of Prometheus metrics, such as Thanos[4]. To limit the complexity of the project and its scope, only aggregated metrics regarding CPU and memory utilization of running containers and nodes will be queried and exported to long-term storage. The metrics will be exported to a MySQL[5] database. Afterward, a written application will be able to import and analyze the data for a comparison of the stored metrics. The relationship between servers and clients is visualized in Figure 6.4.

6.4 Development and Testing

The project consists of multiple technologies, such as Terraform and Ansible, to create various cloud edge clusters. The project creates a CLI tool to deploy and monitor services. Additional services also need to be deployed for certain use cases. The Kubernetes metrics server is needed to utilize the developed application for monitoring, and the Prometheus server is needed to collect metrics information for comparison of technologies. Thus, extensive testing is needed to validate that the project is functioning correctly. Extensive automated testing is needed to validate that the entire project functions correctly using all of the cloud edge technologies, and after any change to the code base. The purpose of this section is to outline the needed testing and its automation using continuous integration and continuous testing.

[4] https://thanos.io/
[5] https://www.mysql.com/

6.4.1 End-to-End Testing

End-to-end testing verifies the integration of an overall system by validating the entire breadth of a system, including downstream systems. End-to-end, also referenced as E2E, tests may even run for several hours and tend to require more maintenance. The purpose of these tests is to determine whether all components are properly integrated from end to end. This can result in a few tests that can activate all components. [24]

As the project consists of multiple components, it is necessary to verify that the project works correctly from start to finish and that all components are integrated correctly. This means that users can provision the desired infrastructure, create a specified cluster, deploy and monitor services using developed tools, and collect cluster metrics.

To further ensure that a created cluster provides all the required functionalities, the demonstration monitoring application, which was developed in Chapter 3, will be deployed and its functionality verified. The application consists of multiple services that are to be deployed across several nodes and communicate with each other for the application to function correctly. The application will be deployed and queried to verify its functionality.

The following steps represent the core steps of a single end-to-end test run that will be implemented. Additional logic is omitted for simplicity purposes. Test runs will be performed for all supported cloud edge technologies. The steps are visualized in Figure 6.5.

- **Provisioning of infrastructure**: The Terraform project successfully provisions a configured infrastructure.
- **Installation of an edge cloud technology**: The Ansible project successfully configures all the provisioned machines and installs a specified management technology.
- **Installation of additional services**: Additional services to verify functionalities are deployed, such as the Kubernetes metrics server and the Prometheus server.
- **Deployment of the demonstration application**: The demonstration application is successfully deployed on the specified nodes using the developed CLI application.
- **Verification of the application**: The application services communicate successfully with each other to create expected messages.
- **Monitoring of the application**: The demonstration application can be monitored using the developed CLI application.
- **Collection of metrics**: The Prometheus server provides collected metrics.
- **Export of metrics**: The collected metrics can be exported to an external database.
- **Saving of cluster information**: Cluster information, such as API resources and logs, is saved for debugging purposes.
- **Saving results**: The results of a test run are saved and reported.

After development has finished, this run can also be used later to collect performance metrics of the technologies over a number of runs to provide statistically comparable results.

To ensure that bugs in development are detected early and to ensure that committed code does not introduce regressions, E2E testing must be performed on every code change. To achieve that, continuous integration and continuous testing are needed.

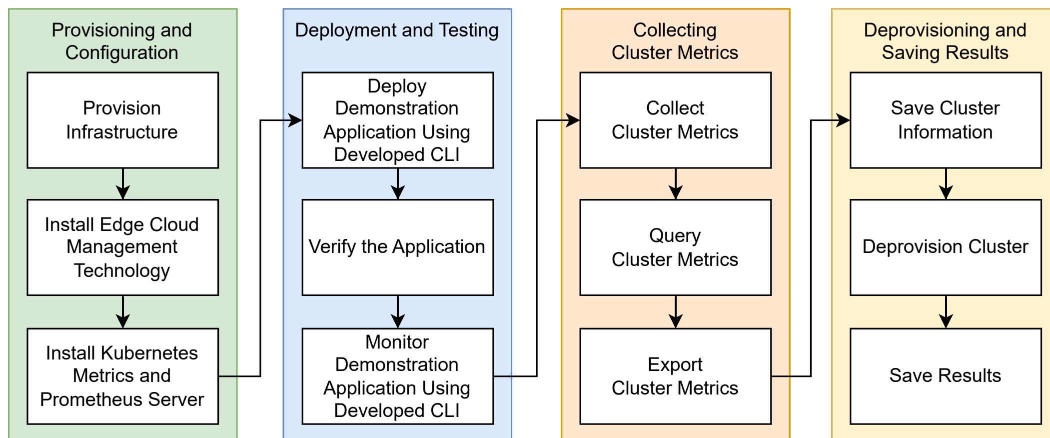


Figure 6.5: A high-level overview of a single test run.

6.4.2 Continuous Integration and Continuous Testing

Continuous integration (CI) is the process of continuously integrating code changes. CI is necessary when working in teams to ensure that changes from several developers are continuously integrated. Integrating changes at the end of a development cycle may become considerably time-consuming, as issues regarding integration are found much later, when the context of relevant changes may already be forgotten, and the faulty changes are already intertwined in the code base. [25]

It is essential to continuously receive feedback to regulate the quality of the code base throughout the development cycle. Early detected issues are immediately addressed. This ensures that problems do not go unnoticed for long periods of time, preventing them from escalating and increasing the effort required to resolve them. [25]

Continuous testing (CT) is the process of continuously validating application quality using manual and automated tests after every change. This further provides additional feedback on the quality of a change as early as possible. CT validates code changes holistically, including its functional and cross-functional aspects, as opposed to CI, which often contains only micro-level tests, such as unit tests. CT is significantly leveraged by CI to perform extensive automated testing against every change before its integration. [25]

However, CI and CT require the following components [25]:

- A version control system to track changes and to hold the application code base.
- Automated tests to validate the application.
- A CI server to automatically execute the automated tests against the latest changes of the application code.
- An infrastructure to host the CI server.

To ensure continuous testing throughout the development of the project, CI and CT will be utilized. This will ensure that end-to-end tests are run on every change to continuously validate changes and detect issues early. The project will utilize the GitHub Actions platform to enable CI and CT for the project, as it takes care of every relevant component needed, such as a version control system, CI servers, and their infrastructure.

Continuous Integration and Continuous Testing Using GitHub Actions

GitHub Actions[6] is a continuous integration and continuous delivery platform that automates the build, test, and deployment pipeline. A GitHub Actions workflow can be configured to be triggered upon certain events in a repository, such as a pull request being opened. A workflow is run on a freshly provisioned virtual machine and contains one or more jobs that can run in parallel or sequentially. A job may run inside a virtual machine or a container. A job consists of steps that define the logic that will be executed. A workflow is represented by a YAML file within the repository. For example, workflows can be defined to build and test pull request changes and deploy applications. [34]

The designed project will utilize the GitHub Actions platform to automate code integration, testing, and delivery. As it is a GitHub[7] feature, its incorporation into a repository hosted on GitHub is simplified. Thus, the project code base will be developed and maintained in a GitHub repository. GitHub Actions workflows will be written to publish container images, check code formatting and compilation, and test the project from end to end. These actions will be automatically performed for changes to the repository. End-to-end testing will therefore be able to run automatically on every change to ensure continuous testing. An architecture diagram of services used during a CI run using GitHub Actions is available in Figure 6.6, where a change is published to a GitHub repository, and a GitHub Actions workflow performs end-to-end testing, publishes container images, and exports metrics, among others.

6.5 Summary

The project for the deployment of edge cloud services consists of a number of components and technologies. From provisioning infrastructure using Terraform, configuring machines using Ansible, and deploying and monitoring services, to collecting metrics and analyzing them for comparison. To ensure that the project is validated throughout the development, it will be continuously integrated and continuously tested. The chapter has described how the proposed solution will be implemented. The following chapters will focus on the actual implementation and subsequent comparison of explored technologies.

[6] https://github.com/features/actions
[7] https://github.com/

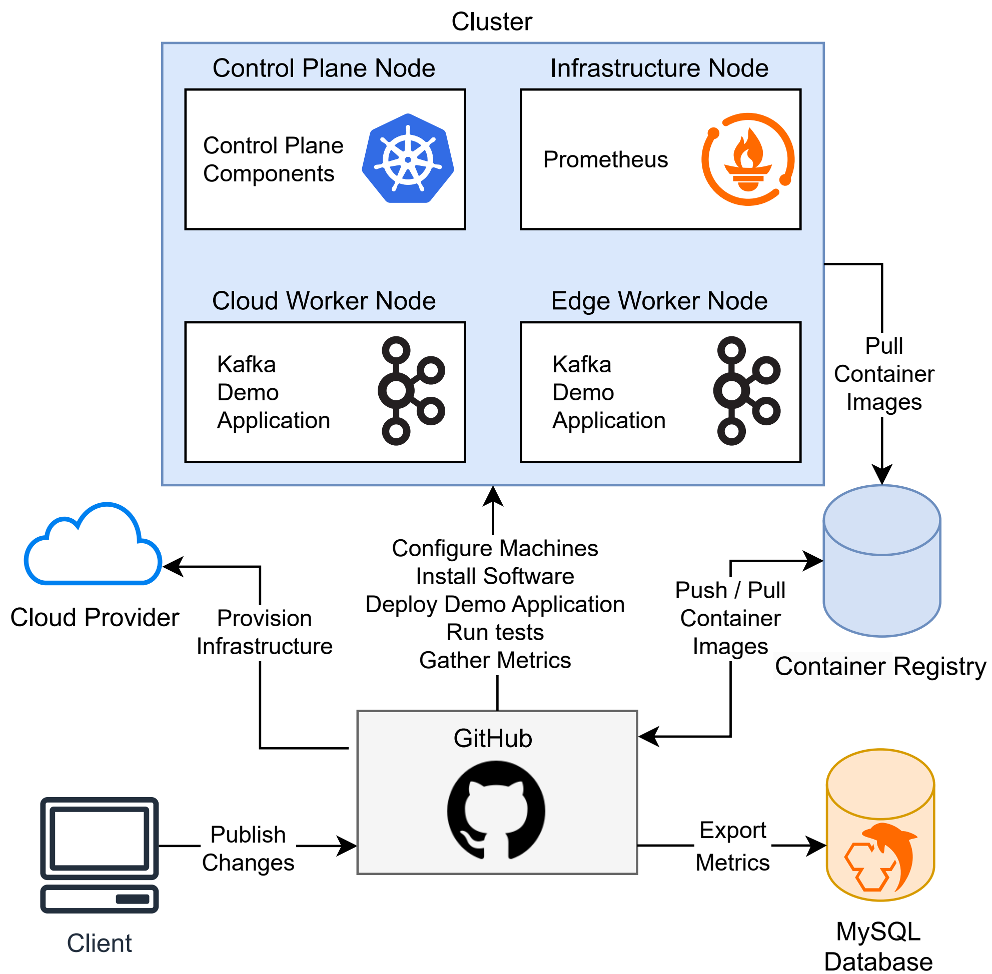


Figure 6.6: A simplified architecture of the overall project upon continuous integration.

Chapter 7

Implementation of Edge Cloud Services Deployment Project

The purpose of this chapter is to describe how the individual components of the edge cloud services deployment project were implemented. Notable logic, libraries, and services are described for each component.

7.1 Provisioning of Resources

A Terraform project was implemented to provision the fundamental IT resources needed for the creation of a cluster using the DigitalOcean[1] cloud provider. DigitalOcean was selected for its affordability and comprehensive documentation. A user can provide input variables to define the desired nodes of a cluster. A default configuration of a four-node cluster, which consists of a control plane, cloud, edge, and infrastructure node, is available.

The `digitalocean`[2] Terraform provider is used to interact with the DigitalOcean cloud provider to create and partially configure the necessary infrastructure.

Depending on the user configuration of the nodes, virtual machines, referred to as droplets by DigitalOcean, are created. The user may specify the desired region, size, and operating system of the nodes. A DigitalOcean load balancer is created for control plane nodes as well, in case a user desires a highly available cluster. This is done by using the resources of the Terraform provider and configuring them adequately.

Access to provisioned machines is permitted using an SSH connection and by providing a valid certificate, which is specified during provisioning. A firewall is also configurable to ensure that the machines can communicate with each other and the Internet without being accessible by unknown hosts. It is possible to configure an external allowed IP address that may communicate with the machines. The cloud provider creates a virtual private cloud (VPC) by default. Thus, a private network is available for the machines to use. A diagram of a possible cluster configuration is available in Figure 7.1.

Another significantly used Terraform provider is the `ansible`[3] provider, which is used together with the `cloud.terraform`[4] Ansible collection to seamlessly integrate the Terraform and Ansible project. These libraries enable Ansible to interpret a Terraform state

[1] https://www.digitalocean.com/
[2] https://registry.terraform.io/providers/digitalocean/digitalocean
[3] https://registry.terraform.io/providers/ansible/ansible
[4] https://github.com/ansible-collections/cloud.terraform

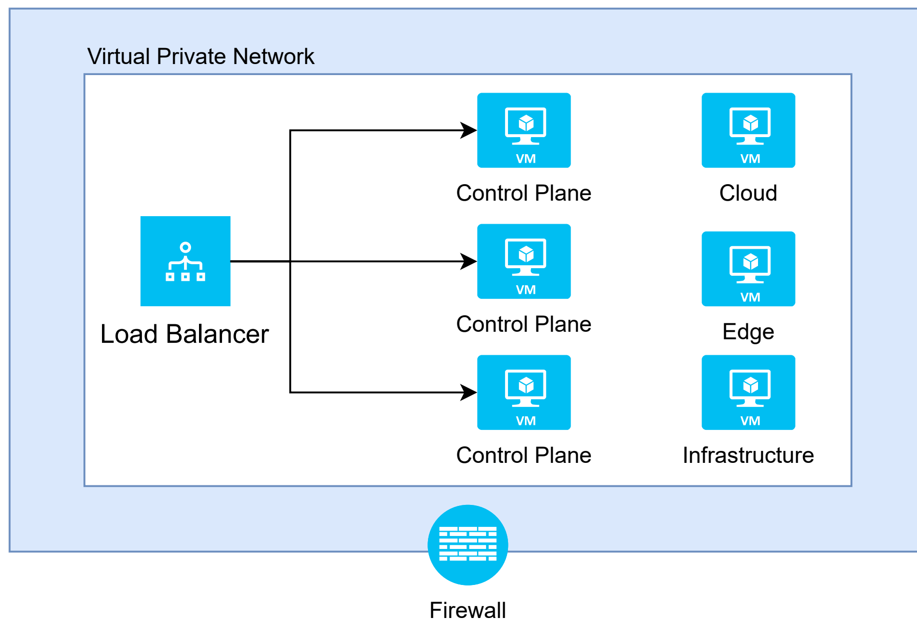


Figure 7.1: An example of provisioned DigitalOcean resources by the project.

file. A state file contains information on provisioned resources. The resources of the `ansible` Terraform provider are used to group virtual machines by user configuration and to provide additional information about the nodes and the cluster. This information gets saved to the state file upon Terraform execution. As such, using the mentioned Ansible collection, the collection is able to read a Terraform state as if it were a normal inventory file. This also means that the Terraform project is not inherently needed for the Ansible project to function. A user may manually create an inventory file and define the hosts and required information. However, for automation purposes, the project utilizes the cloud provider to provision new machines quickly and in a repeatable manner. The Ansible project then configures the machines to deploy a specified Kubernetes cluster using written playbooks. A simple overview of the relationship between the components is represented in Figure 7.2.

7.2 Installation of Management Technologies

An Ansible project was written to configure the machines and install the desired edge cloud technology using a manual deployment tool. The Ansible project configures machines specified in an inventory file to provide a specified Kubernetes cluster using written playbooks that utilize Ansible plays, tasks, roles, and benefits from collections provided by the Ansible community. The project consists of playbooks and a number of roles, which contain plays designed to be imported. The supported technologies are represented by high-level playbooks whose structure consists of three main stages that import additional playbooks. The stages are start-up, installation, and post-installation.

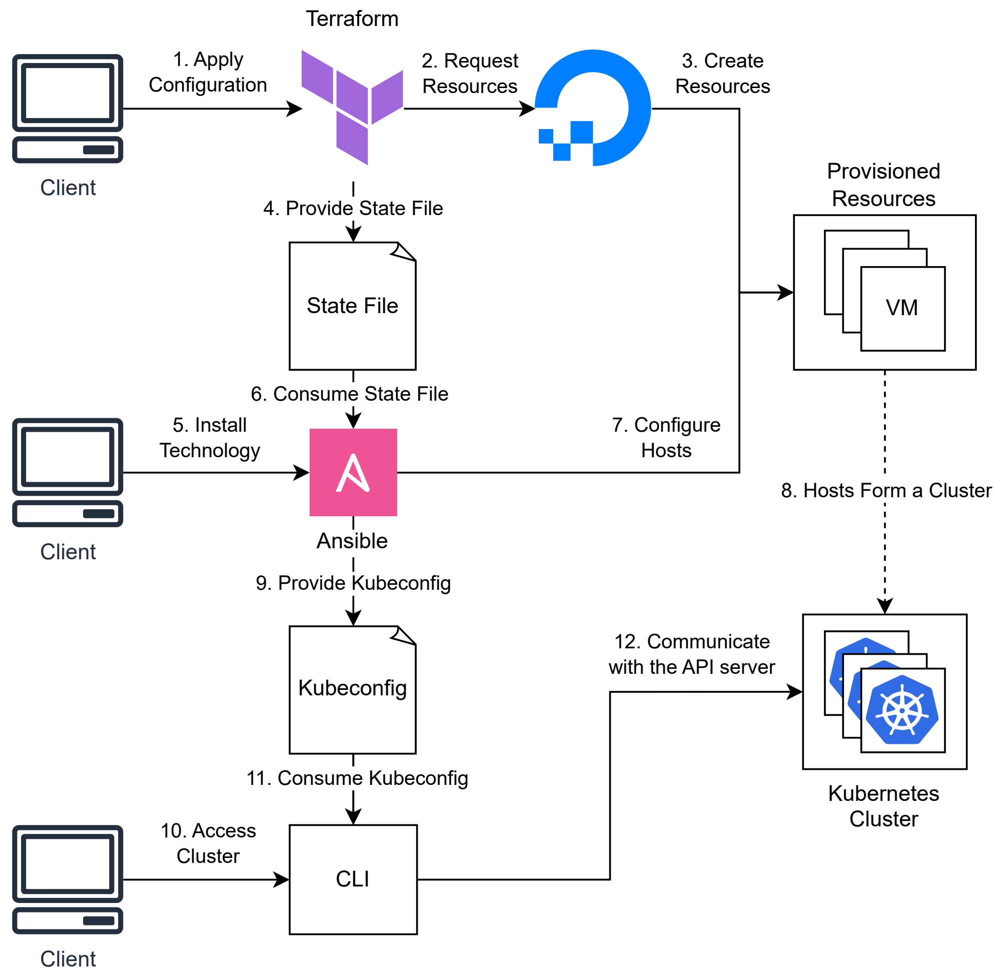


Figure 7.2: A diagram of the relationships between the components of the project.

Start-up

The start-up stage contains logic, which is agnostic to the technology that will be installed, and is related to all machines on start-up. Currently, it consists of logic to wait until all machines are ready to receive an SSH connection. This is important for automating the execution of the project, as the machines may not be ready when executing the Ansible project. This is achieved by the built-in Ansible module `ansible.builtin.wait_for_connection`, which provides a simple method of blocking until all machines are ready.

Installation

The installation stage adequately configures all machines according to the chosen technology. The stage prepares the machines for installation, installs the desired technology, forms a cluster, and provides an access file to the cluster. The installation stage for individual technologies is described in more detail in the following subsections. Kubernetes

version 1.29 is used in all installations. The containerd[5] container runtime is used in all installations, and the Flannel[6] container network interface plugin is used if not specified otherwise. Ubuntu 24.04 was chosen as the operating system for the hosts. As of the time of writing, the support for high-availability clusters is not fully implemented for all technologies, and thus its description is omitted.

Notable Ansible collections that were used are `cloud.terraform`[7] to read the Terraform state file and `kubernetes.core`[8] for Kubernetes usage. For general tasks, the `ansible.builtin`[9] and `community.general`[10] collections are utilized.

Post-Installation

The post-installation stage contains logic that is agnostic to the technology that will be installed and is related to all machines after installing the management technology. The logic installs a Prometheus server on an infrastructure node and some of its related services using Helm and a written Helm values file, which contains relevant configuration values, such as defining the Prometheus server to be installed on an infrastructure node. This is achieved using the `kubernetes.core` Ansible collection.

7.2.1 Installation of K3s

K3s clusters are installed using the k3sup utility. K3s is one of the most popular lightweight Kubernetes technologies. The project has surpassed other projects in GitHub stars multiple times. This results in community projects related to the technology, such as k3sup[11]. An important aspect when choosing technology is that larger communities result in more contributors and related projects, which can simplify and streamline their usage.

K3s, by default, is able to configure a host, install a container runtime, a container network interface plugin, and various additional services, such as the Kubernetes metrics server. Normally, to form a K3s cluster, a cluster administrator needs to access each individual node, download a K3s installation shell script, and execute it appropriately to form a cluster. The k3sup utility automates this process. Using the CLI application, which can be executed from a local machine, an administrator can form a K3s cluster without manually accessing the nodes.

A K3s cluster deployment using Ansible thus automates the k3sup usage given a received inventory file. The installation can be broken down into a few notable parts. The installation of the Kubernetes metrics server is omitted as it is already installed by default.

1. Creation of the initial control plane node using `k3sup install`
2. Addition of data plane nodes to form a cluster using `k3sup join`
3. Labeling and tainting of nodes using `kubectl`

As such, the installation of a K3s cluster in combination with a tool such as k3sup results in a simple deployment of a fully functional and lightweight Kubernetes cluster.

[5] https://containerd.io/
[6] https://github.com/flannel-io/flannel
[7] https://github.com/ansible-collections/cloud.terraform
[8] https://github.com/ansible-collections/kubernetes.core
[9] https://github.com/ansible/ansible
[10] https://github.com/ansible-collections/community.general
[11] https://github.com/alexellis/k3sup

7.2.2 Installation of MicroK8s

MicroK8s clusters are installed using the `microk8s` snap package. Installation of such clusters is still simplified, as the package configures a host and installs the container runtime and the container network interface plugin. However, the process, which was automated using k3sup for K3s clusters, had to be automated for MicroK8s clusters. The installation may be broken down into the following important steps.

1. Creation of individual MicroK8s nodes
   (a) Installation of `microk8s` on each machine using `snap`
   (b) Switching from default to the Flannel CNI plugin using `microk8s disable`
2. Designation of the control plane node of the desired cluster
   (a) Creation of a token by running `microk8s add-node` on the control plane node
   (b) Export of administrator credentials
3. Addition of remaining nodes using `microk8s join` on each of the nodes
4. Labeling and tainting of nodes using using `kubectl`
5. Enabling the cluster DNS and Kubernetes metrics server using `microk8s enable` on the control plane node

However, additional logic was also required for automating purposes. For example, wait for nodes to be ready and wait until nodes have joined the desired cluster. MicroK8s uses by default the Calico[12] CNI plugin. To maintain the same core components across technologies, additional steps had to be taken for this purpose. The cluster creation is thus still straightforward, as only the installation of one snap package is needed across the nodes. Similarly to K3s, MicroK8s supports a number of additional services by default as well. Further steps had to be taken due to the nature of the manual installation and for automating purposes.

7.2.3 Installation of Kubernetes

Classic Kubernetes clusters are installed using the `kubeadm` tool. This installation starts to showcase the benefits of a simpler deployment using technologies such as K3s and MicroK8s. The installation can be broken down into the following important steps.

1. Preparation of each machine
   (a) Configuration of networking rules
   (b) Installation of Kubernetes tools `kubelet`, `kubeadm`, and `kubectl`
   (c) Installation and configuration of a container runtime
2. Initialization of a single-node cluster using the designated control plane node
   (a) Initialization of a control plane node using `kubeadm init`

[12] https://github.com/projectcalico/calico

(b) Installation of a CNI plugin using `kubectl`
(c) Export of administrator credentials
(d) Generation of a token for additional nodes using `kubeadm token create`

3. Addition of the remaining nodes using `kubeadm join` on each of the nodes
4. Labeling and tainting of nodes using `kubectl`
5. Installation of the Kubernetes metrics server using `helm`

Creating a classic Kubernetes cluster using `kubeadm` may require more steps than installation using `k3sup` or `microk8s`. However, the approach is very similar to that of a MicroK8s cluster deployment using `microk8s` with the addition of installing and configuring all relevant dependencies and additional services.

7.2.4 Installation of KubeEdge

KubeEdge clusters are installed using `keadm` and `helm`. KubeEdge extends a Kubernetes cluster. Thus, all steps described for a classic Kubernetes installation are required for all nodes except the steps required for edge nodes, which are handled by the KubeEdge installation. The KubeEdge version 1.19 is installed due to its compatibility with the used Kubernetes version.

Instead of using Flannel to enable networking between containers, the combination of reference CNI plugins[13] and the networking solution EdgeMesh[14] is used for the connectivity of edge nodes. This approach is documented and supported by KubeEdge. Using other networking solutions is possible; however, it requires additional manual steps to be executed and is less documented. As such, the mentioned approach was chosen for the implementation. EdgeMesh was chosen to support DNS on edge nodes. The installation can be broken down into the following steps.

1. Creation of a cloud Kubernetes cluster using the steps described in Section 7.2.3
2. Installation of KubeEdge cloud capabilities using `keadm init`
3. Obtainment of the token for edge nodes using `keadm gettoken`
4. Configuration and addition of edge nodes to the cluster
   (a) Installation and configuration of a container runtime
   (b) Installation and configuration of reference CNI plugins
   (c) Installation of `keadm`
   (d) Addition of the node to the cluster using `keadm join`
   (e) Configuration of the node to enable querying of container logs
   (f) Configuration of the node to support DNS and EdgeMesh
5. Installation of EdgeMesh using `helm`

[13]https://github.com/containernetworking/plugins
[14]https://edgemesh.netlify.app/

The installation of KubeEdge clusters requires the most steps and configurations to create a functional Kubernetes cluster extended to the edge. Additional steps must be taken to support basic functionality, such as `kubectl logs` to view container logs or to enable DNS for containers on the edge. However, KubeEdge outweighs these steps with its rich functionalities for edge cloud environments.

7.3 Deployment and Monitoring of Services

The central component of the explored edge cloud technologies is the Kubernetes API server. A command line interface (CLI) application was written to communicate with the API server to deploy and monitor services.

A CLI application was written in the Go[15] programming language to deploy and monitor services in Kubernetes clusters. The Go programming language was chosen as a majority of Kubernetes libraries are written in Go, since Kubernetes itself is primarily written using the language. The Cobra[16] CLI interface library was utilized to create the CLI tool. Notable Go modules used to communicate with the API server and decode Kubernetes manifest files are `k8s.io/client-go`[17], `k8s.io/apimachinery`[18], and `k8s.io/metrics`[19]. The dependencies of the application are tracked to ensure a successful compilation in the future. Two main subcommands were implemented. The `apply` subcommand to deploy services and the `top` subcommand to monitor services.

The `apply` subcommand searches a specified directory for Kubernetes manifest files. It reads discovered files and decodes their content into individual generic objects. To be able to parse manifest files that contain any API objects to provide more value, the objects are handled as generic objects. This is achieved using the `unstructured`[20] package. The `Unstructured` data type provided by the package allows the application of unstructured objects to a cluster using a dynamic client as opposed to an API-versioned client. Additional communication with the API server is required to receive an object's resource mapping; however, this results in a subcommand that is able to apply any supported API object.

The `top` subcommand communicates with the API server to gather information about the existing nodes and containers. The information is subsequently shown in a defined and structured format. Two Kubernetes API groups are utilized. The core API group is used to fetch information about existing pods, containers, and nodes. Information such as pod placement on a node, status of containers, and nodes. The metrics API group is used to fetch information about the resource utilization of individual containers and nodes. The gathered information is formatted and shown. An example of an output from the `top` subcommand is available in Figure 7.3.

Leveraging the comprehensive Kubernetes Go libraries makes it easier to perform the deployment and monitoring of services. This is not surprising since Kubernetes is widely recognized for its extensive API capabilities and active community.

[15] https://go.dev/
[16] https://github.com/spf13/cobra
[17] https://pkg.go.dev/k8s.io/client-go
[18] https://pkg.go.dev/k8s.io/apimachinery
[19] https://pkg.go.dev/k8s.io/metrics
[20] https://pkg.go.dev/k8s.io/apimachinery/pkg/apis/meta/v1/unstructured

```
$ ./bin/cloud-edge top --namespace kafka-app --kubeconfig playbooks/kubeconfig
NAMESPACE   POD                                  CONTAINER      STATUS      CPU(cores)   MEMORY(bytes)   NODE
kafka-app   demo-app-cloud-59c48df48c-f5nfz      app-cloud      Running     26m          100Mi           stage-cluster-cloud-d64a99bc
kafka-app   demo-app-edge-79bcc6f546-5gsbq       app-edge       Running     22m          130Mi           stage-cluster-edge-ecd6fad4
kafka-app   demo-app-sensor-55bf485dd8-7fkth     app-sensor     Running     26m          106Mi           stage-cluster-edge-ecd6fad4
kafka-app   demo-app-sensor-55bf485dd8-mvjrn     app-sensor     Running     28m          110Mi           stage-cluster-edge-ecd6fad4
kafka-app   e2e-9bdt4                            e2e            Completed   0m           0Mi             stage-cluster-cloud-d64a99bc
kafka-app   kafka-cloud-7985f5b6d-lrcq5          kafka-cloud    Running     102m         390Mi           stage-cluster-cloud-d64a99bc
kafka-app   kafka-edge-7fdb69fd47-b2j66          kafka-edge     Running     70m          405Mi           stage-cluster-edge-ecd6fad4
kafka-app   kafka-mirror-9f94d4dd9-77t54         kafka-mirror   Running     24m          441Mi           stage-cluster-edge-ecd6fad4
kafka-app   kafka-ui-5bfbfd49c-2ktqq             kafka-ui       Running     8m           240Mi           stage-cluster-cloud-d64a99bc

NODE                                 READY   CPU(cores)   MEMORY(bytes)
stage-cluster-cloud-d64a99bc         True    277m         1980Mi
stage-cluster-control-plane-9f50ec4d True    119m         1001Mi
stage-cluster-edge-ecd6fad4          True    413m         3226Mi
stage-cluster-infra-cdcb9ee0         True    37m          961Mi
```

Figure 7.3: An example of monitoring services using the implemented CLI application.

7.4 Development and Testing

The goals of continuous integration and continuous testing were achieved by implementing the GitHub Actions workflows to run the following procedures on relevant changes.

- Run end-to-end testing for a matrix of configurations
- Publish a container image of a control node's dependencies
- Verify compilation of the CLI application source code and its formatting
- Validate the Terraform project and verify its formatting

End-to-end testing validates the overall project. A container image of a control node's dependencies is built using a defined Dockerfile and published to the Docker Hub[21] container registry to ensure tracking of dependencies and repeatability. The container image is used by the control node that executes the end-to-end testing. Additional workflows were created to improve the quality of the project by requiring correct syntax and appropriate formatting and validating their usage.

The credentials, such as those for the cloud provider and private container registry, and various configuration values are saved in the GitHub repository secrets and variables. As such, secret values are treated as sensitive data and are encrypted. Variables allow for configuring values for the repository instead of hard-coding values in the source code.

The end-to-end testing workflow provides the most value to the project. It uses the project functionalities from start to finish and verifies the usability of a cluster by deploying the demonstration Kafka application.

7.4.1 Deploying the Kafka Demonstration Application

To deploy the Kafka Demonstration application, the application was defined using Kubernetes manifest files, and a container image of the implemented application was built and published to the Docker Hub container registry. The application is represented by Kubernetes objects, such as deployments, services, and a configuration map. Additional objects, such as a job and a secret, were used to run testing and to contain credentials to the container registry. To ensure that pods defined by the deployments are deployed on desired nodes, the workload is defined to be scheduled on adequately labeled nodes, which

[21] https://hub.docker.com/

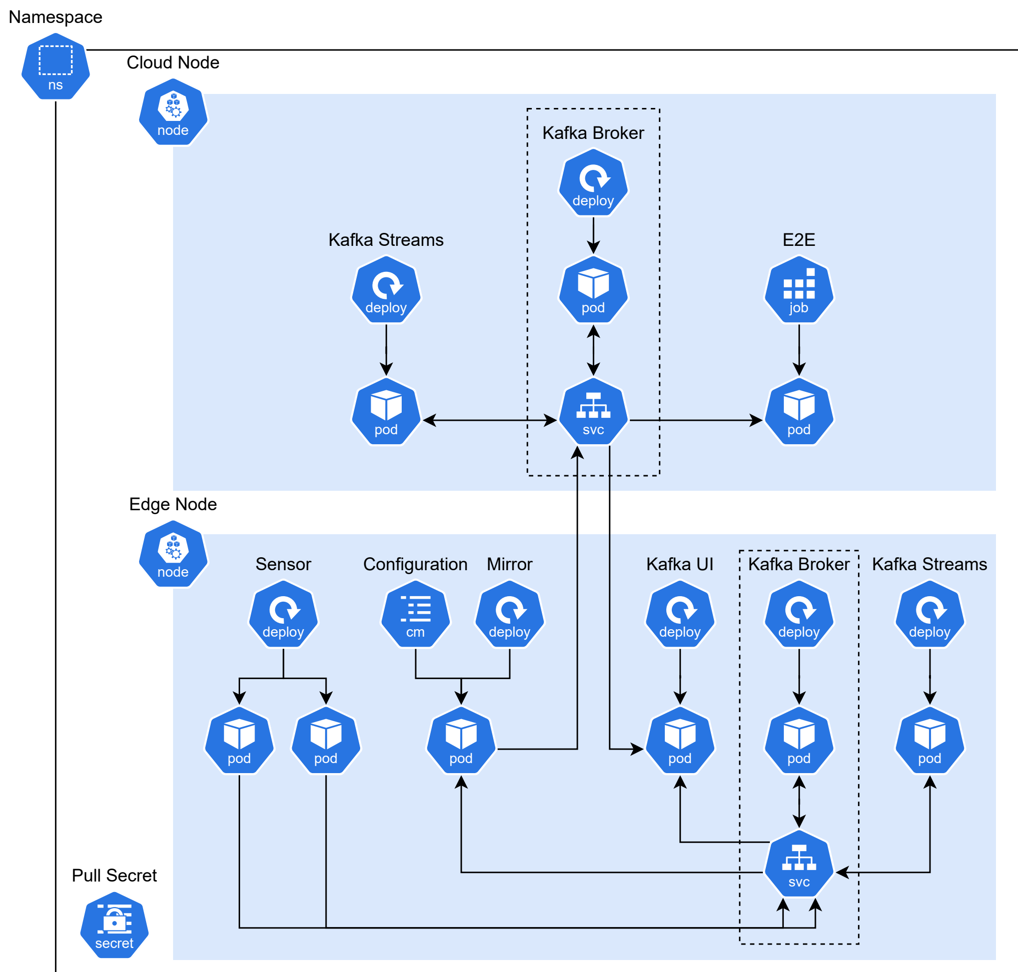


Figure 7.4: A diagram of the demonstration application using Kubernetes API objects.

are labeled and tainted during installation to ensure only the desired workload is scheduled onto the nodes. The overall application is written to be deployed within a single Kubernetes namespace. Kubernetes deployments contain information for deploying the containers of the demonstration application, such as the container image, the number of replicas, and the placements of the pods. Kafka brokers are exposed using Kubernetes services. A configuration map contains a configuration for the mirroring application, which is mounted in the respective container. A Kubernetes job was written to deploy the testing container in the cluster. Finally, a pull secret was used during development to access a private container registry to pull the demonstration container image. The diagram of the application as a Kubernetes application using the relevant resources is available in Figure 7.4.

7.4.2 End-to-End Testing

The end-to-end testing workflow is the most important of the implemented workflows. Its configuration matrix consists of four technologies and two cluster topologies. In total, eight unique configurations. Each configuration is executed via an individual job. A default

cluster topology of a single control plane, cloud, edge, and infrastructure node, and a larger cluster topology consists of a single control plane node and double cloud, edge, and infrastructure nodes. Each technology is deployed in both cluster topologies. This results in 44 provisioned virtual machines for a workflow run. For each configuration, a job is created that consists of the following steps.

1. Cluster creation using Terraform and Ansible projects
   (a) Provision resources using `terraform apply`
   (b) Configure hosts using `ansible-playbook`
2. Verifying the demonstration Kafka application deployment and monitoring
   (a) Apply a secret resource, which contains container registry credentials
   (b) Compile the CLI application
   (c) Apply the demonstration application manifests, including the E2E job, using the compiled CLI application
   (d) Monitor the deployed workload using the CLI application
   (e) Wait for the E2E job to successfully complete
3. Metrics collection and export (for default cluster topology)
   (a) Wait for the Prometheus server to collect a sufficient number of metrics
   (b) Execute scripts for metrics querying and export
4. Saving debugging information
   (a) Save the cluster information using `kubectl cluster-info dump`
   (b) Upload the information
5. Deprovisioning of Resources

The application functionality is verified by fetching the cloud Kafka broker for the existence of a sufficient number of topics generated by the cloud Kafka Streams application. Sensor applications generate topics at the edge that are processed and mirrored to the cloud. The Kafka Streams cloud application processes these topics and generates new ones, which are stored using the Kafka broker in the cloud. Verifying processed topics in the cloud ensures that all respective services function from start to finish, including sensors, processing applications, and Kafka brokers. A failure in a single producer or consumer would result in an overall failure of the application that could be observable by missing processed topics in the cloud.

Additional logic is present, such as failing the overall workflow run on at least a single failed job and deprovisioning resources on any outcome of a run. GitHub Actions is also capable of sending alerts given a failed workflow run, which is welcomed, given that a single workflow run of the project may run up to an hour in extreme cases. However, end-to-end testing is helpful in detecting regressions immediately and, as such, is worth the development, maintenance, and cloud costs. An example of an overview of a change that passes the CI is available in Figure 7.5.

Figure 7.5: A successful CI run overview upon publishing a change.

7.5 Cluster Metrics Comparison

Shell scripts were written to query the Prometheus server and export aggregated metrics to a MySQL database. Subsequently, a Python application was written to import the database data and analyze them.

7.5.1 Querying Collected Metrics

A shell script was written to query the Prometheus server. Queries were written in Prometheus query language (PromQL) and used to query the Prometheus server for aggregated metrics regarding the resource utilization of deployed containers and nodes.

Using relevant metrics, PromQL functions, and aggregation operators, queries were written to query information regarding average resource utilization over a period of time for containers and nodes.

In order not to expose the Prometheus server on a public IP, the script queries the server using Kubernetes jobs. The script deploys Kubernetes jobs in the cluster to query the Prometheus server from within the cluster. Upon successful completion, the output of a job is provided as input to an additional shell script whose purpose is to store the data.

7.5.2 Storage of Queried Metrics

The purpose of the additional shell script is to filter and insert the aggregated metrics into a MySQL database, including meta-information. A MySQL database managed by DigitalOcean was used to store the data. The schema of the table used to store the data is available in Figure 7.6, where a source property represents the source of a metric, such as an edge node. Technology corresponds to the source's cluster technology, such as KubeEdge. The metric type specifies the type of recorded value, such as node memory utilization. The data in the database can be subsequently imported and subjected to analysis.

Metrics
<PRIMARY KEY> ID INT
Source VARCHAR
Technology VARCHAR
Value DOUBLE
Recorded DATETIME
MetricType VARCHAR

Figure 7.6: A schema of the table used for storing the collected metrics.

7.5.3 Analysis Tool for Comparison of Stored Metrics

A Python application was written to import, analyze, and visualize data from the MySQL database. The application utilizes a number of notable Python modules. The application establishes a connection to the database using the SQLAlchemy[22] module. The module is a Python SQL toolkit and an object relational mapper. The database data are loaded using the pandas[23] module. The module is also used in combination with the numpy[24] module to manipulate the data. The pandas module also forms table summaries of the respective technologies, such as summaries of median values, standard deviations, and median absolute deviations. The data is visualized using the Matplotlib[25] and seaborn[26] modules to generate box plot graphs of the collected data.

Statistical tests, such as one-way analysis of variance and the Shapiro-Wilk test, are performed using the SciPy[27] module in combination with post hoc tests, which are done using the scikit-posthocs[28] module. Pairing annotations in box plot graphs, signifying the corresponding p-values of post hoc tests for pairwise comparisons between groups, are made using the statannotations[29] module.

The dependencies of the application are tracked using the Pipenv[30] Python virtualenv management tool. Deterministic builds are ensured due to the tool's generated `Pipfile` and `Pipfile.lock` files that track needed packages.

---

[22] https://www.sqlalchemy.org
[23] https://pandas.pydata.org
[24] https://numpy.org/
[25] https://matplotlib.org
[26] https://seaborn.pydata.org
[27] https://scipy.org/
[28] https://github.com/maximtrp/scikit-posthocs
[29] https://github.com/trevismd/statannotations
[30] https://pipenv.pypa.io/

7.6 Summary

Several technologies and services were used to implement the project for the deployment and monitoring of edge cloud services. A Terraform project was implemented to provision IT resources, which can be later configured to form a K3s, MicroK8s, classic Kubernetes, or KubeEdge cluster. A CLI application was written to communicate with the Kubernetes API server to deploy and monitor services. GitHub Actions workflows were written to, among other things, test the project workflow from start to finish, and scripts were written to query and export metrics that can later be analyzed by a written Python application.

A number of services were utilized to achieve this. The DigitalOcean cloud provider was used to provision resources and a database. Docker Hub, a container registry, was used to store container images. Finally, GitHub Actions are used to define and host continuous integration and continuous testing.

The project for the deployment and monitoring of edge cloud services and the Kafka demonstration application were published as open-source projects on the GitHub platform under the GNU General Public License v3.0. The project for the deployment and monitoring of edge cloud services was published in the DavidHurta/edge-cloud[31] repository. The Kafka application was published in the DavidHurta/demo-edge-cloud-app[32] repository.

[31] https://github.com/DavidHurta/edge-cloud
[32] https://github.com/DavidHurta/demo-edge-cloud-app

Chapter 8

Comparison of Explored Edge Cloud Technologies

The purpose of this chapter is to compare the explored edge cloud management technologies. The deployment difficulty, usage, and performance of nodes and services are compared. Kubernetes, K3s, MicroK8s, and KubeEdge technologies are compared as used in the implemented project for the deployment and monitoring of edge cloud services, and through the collected metrics.

8.1 Deployment and Usage

Manual deployment tools were used to create K3s, MicroK8s, classic Kubernetes, and KubeEdge clusters to provide a comparable comparison, as the deployment experience may differ significantly depending on the type of tool used. The project has utilized Ansible to configure hosts to form a cluster of a desired technology. Using written Ansible playbooks and roles, concrete, repeatable, and necessary steps can be viewed. The most notable steps used to deploy each technology are described in Chapter 7.2.

Technologies such as K3s and MicroK8s can be categorized as more straightforward to deploy among the technologies. These projects provide additional value, in addition to being lightweight distributions. Minimal prerequisite steps are needed, and optional services are installable. It is worth mentioning that K3s can be installed using basic common commands such as `curl` and `sh`. On the other hand, MicroK8s requires the usage of the `snap` package manager. In addition, the popularity of K3s has given space for projects such as the utilized K3sup, which further simplifies the cluster deployment.

A classic Kubernetes cluster is created using the `kubeadm` tool. Additional prerequisite steps are needed, and additional services need to be deployed separately. There is a clear distinction in difficulty between the explored lightweight distributions and the classic Kubernetes cluster using `kubeadm`. Hosts need manual configuration, installation of the container runtime, and the container network interface (CNI) plugin. Also, the needed dependencies must be compatible with each other's versions. This approach is beneficial for extended configuration, customization, and a deeper understanding of the Kubernetes architecture and its core components.

KubeEdge extends an already running Kubernetes cluster. An existing classic Kubernetes cluster was extended in the project with KubeEdge cloud capabilities using KubeEdge tooling. Edge nodes are configured and subsequently joined using the tooling. Additional

configuration must have been performed to ensure DNS functionality on edge nodes and the ability of a cluster administrator to view logs of containers deployed on edge nodes. The Flannel CNI plugin was omitted on edge nodes due to encountered difficulties, and reference CNI plugins were used. As such, KubeEdge requires even further configuration on top of a classic Kubernetes cluster deployment using multiple tools, and several additional steps have been performed to enable the cluster's capabilities to support the Kafka demonstration application.

A cluster administrator must make a compromise when choosing a technology for their edge cloud environment based on the deployment difficulty between simplicity and the range of customization and provided functionalities. K3s and MicroK8s offer excellent tools to deploy clusters quickly without the need to install dependencies manually and with the possibility of easily installing additional services. The deployment of a classic Kubernetes cluster requires additional steps; however, these steps enable the administrator to configure a cluster in more detail. KubeEdge extends an existing cluster with additional capabilities that may outweigh the complexity of its deployment, depending on the requirements.

The usage of the technologies is primarily the same due to the main component of the technologies, the Kubernetes API server. This fact highlights one of the main benefits of using Kubernetes technologies as a cluster administrator. It is easier to migrate to another cloud provider or infrastructure using Kubernetes, as opposed to migrating from one cloud provider's specific services to a different cloud provider. Using the Kubernetes API server, the usage is abstracted by an extensive and expandable API. A developer using a Kubernetes API server may not even need to know the details of the underlying machines, their provider, or the exact technology to achieve their goals.

8.2 Performance Impact on Services and Nodes

The resource utilization of nodes and containers of the demonstration application between technologies is analyzed. During data collection, a cluster of the project's default topology was used, which consists of a single control plane, cloud, edge, and infrastructure node. Thirty runs were executed for each edge cloud technology. The average resource utilization over ten minutes for the CPU and memory of the containers and nodes was collected to be compared. Nodes of 2 CPUs and 4 GB of RAM were used. Identical steps were executed using each technology. Steps have been made to ensure identical components across technologies, such as the container runtime, and several default services were disabled across technologies; however, default settings were left in some cases.

The resource utilization of node types and specific containers between technologies is compared using statistical tests. This is done to check whether a technology has a statistically significant impact on the resource utilization of a container or a specific node type. A level of significance of 0.05 was used in all tests.

One-way analysis of variance (ANOVA) followed by Tukey's honestly significant difference (HSD) test is done when the respective assumptions are verified. The assumption of independence is fulfilled through the nature of the sampling, as each observation was drawn independently without any interdependence between groups. The assumptions of normality and homogeneity of the variances are ensured through additional tests. The Shapiro-Wilk test is used to test whether the data were drawn from a normal distribution. The Levene test is used to test whether samples are from populations with equal variances.

In the case that the assumptions are not met, the Kruskal-Wallis H-test followed by Dunn's test is executed. Bonferroni correction is used to adjust the p-values in Dunn's test.

Nodes CPU Usage [%] – Median

	k3s	microk8s	kubernetes	kubeedge
control-plane	6.921	7.746	8.917	10.125
cloud	4.992	6.092	6.200	5.954
edge	9.438	10.313	10.179	11.146
infra	2.008	2.354	2.446	2.879

Figure 8.1: A summary of the median CPU usage across nodes and technologies. Colored gradient indicates the range from minimal to maximal values on a given row.

Nodes Memory Usage [%] – Median

	k3s	microk8s	kubernetes	kubeedge
control-plane	17.228	15.704	19.607	24.447
cloud	31.294	31.271	31.132	31.784
edge	41.609	41.890	41.097	42.300
infra	14.128	14.374	12.706	13.912

Figure 8.2: A summary of the median memory usage across nodes and technologies. Colored gradient indicates the range from minimal to maximal values on a given row.

Summaries of median resource usage are provided for a general overview of technologies. Annotated box plot graphs that contain the p-values of respective pairwise tests are provided. Summaries of standard deviations and median absolute deviations were omitted as no substantial information could have been observed.

8.2.1 Performance Impact on Nodes

Comparisons of explored technologies and their impact on the CPU and memory usage of nodes were made using statistical measurements, visualization of data, and statistical tests.

Comparison of CPU Usage

A summary of the median CPU usage between nodes and technologies is available in Figure 8.1. The lowest median CPU usage between all technologies is achieved using K3s. The significantly lower CPU usage across all nodes using K3s suggests potentially the most suitable candidate for low-resource constraint environments. A trend for a higher CPU usage across all nodes can be observed using KubeEdge. Although KubeEdge is specifically designed for edge cloud environments, as mentioned, it extends a Kubernetes cluster with additional capabilities and thus workload. Although KubeEdge edge nodes are created using a KubeEdge command, additional steps have been taken to ensure that the demonstration application is functional, which might have increased CPU usage to some degree. However, KubeEdge showcases higher CPU utilization across all nodes. MicroK8s and Kubernetes CPU usage seem to be very similar, with the exception being the control plane node, which showcases distinct differences in CPU utilization between all the technologies. The difference between the minimal and maximal median values for the control plane type is 3.2 %, while the respective average difference for the remaining types is 1.26 %.

A distribution of CPU usage between nodes and technologies, with the corresponding p-values from the post hoc tests for pairwise comparisons, is available in Figure 8.3. A statistically significant difference was discovered between K3s and the remaining technologies between all types of nodes. Due to the nature of the distributions, medians, and statis-

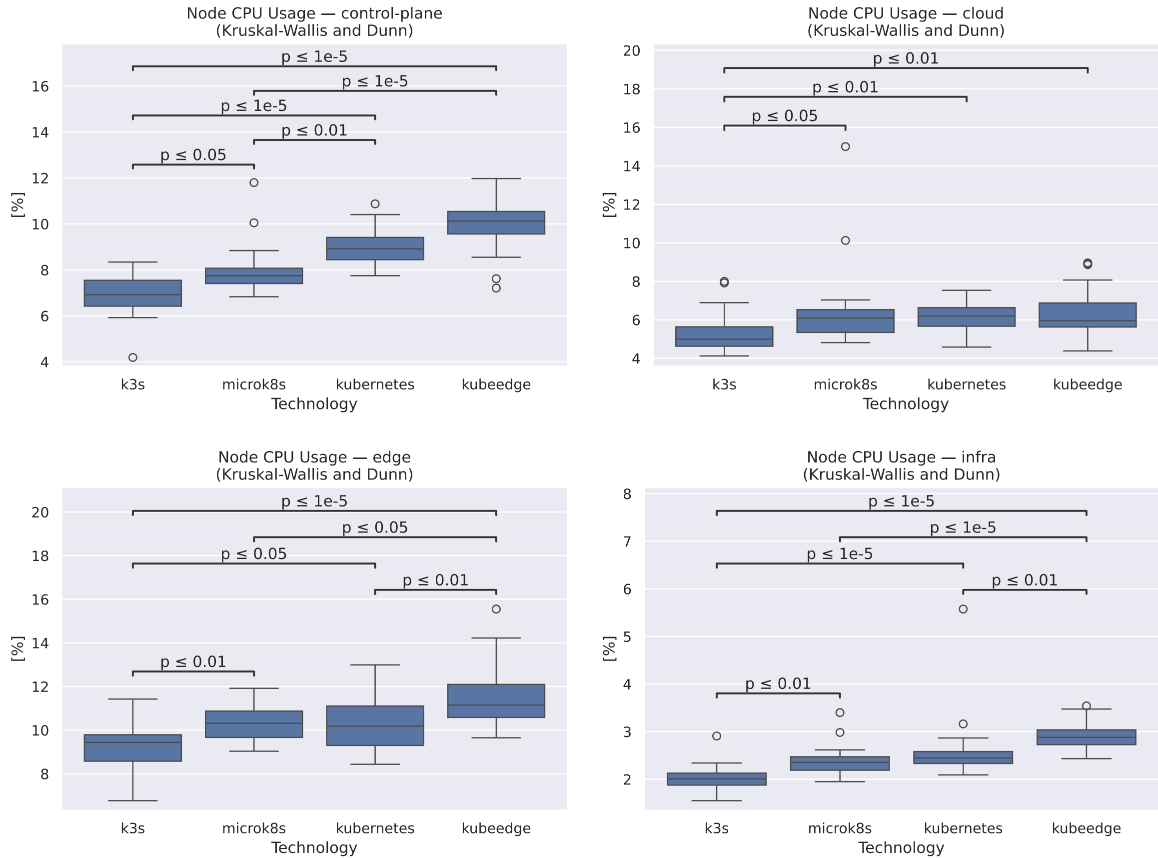


Figure 8.3: A distribution of CPU usage between nodes and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. A pairing annotation signifies a statistically significant difference using a post hoc test and the respective p-value. A missing pairing represents that no statistically significant difference was found.

tically significant differences, it can be strongly suggested that K3s has the lowest overall CPU utilization of nodes among the technologies.

Although not between all types of nodes, the statistical tests suggest a statistically significant difference between KubeEdge and the remaining technologies. Considering the distributions of CPU usage of KubeEdge nodes, an indication of a higher CPU usage can be observed for KubeEdge nodes compared to the remaining technologies. There are no statistically significant differences between the MicroK8s and Kubernetes nodes, except for the control plane node, which shows statistically significant differences between all technologies except for Kubernetes and KubeEdge.

Comparison of Memory Usage

A summary of the median memory usage between nodes and technologies is available in Figure 8.2. Noticeable distinctions between the usage of control plane nodes can be observed, with MicroK8s having the lowest memory consumption and KubeEdge having the highest. A tendency for a lower memory consumption on the remaining nodes can be observed using Kubernetes, and a tendency for an overall higher memory consumption can be seen on the KubeEdge nodes. As was the case for CPU usage, technology has a higher impact on

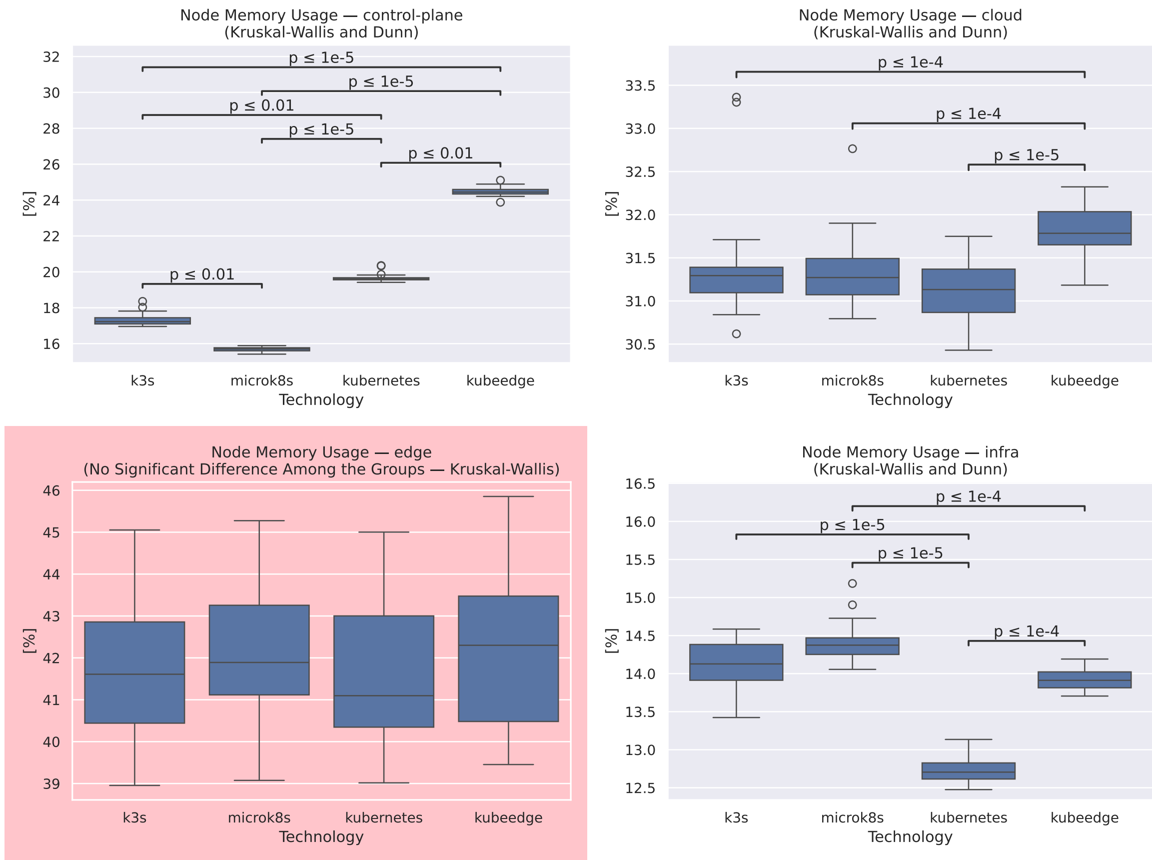


Figure 8.4: A distribution of memory usage between nodes and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. A pairing annotation signifies a statistically significant difference using a post hoc test and the respective p-value. A missing pairing represents that no statistically significant difference was found.

control plane nodes than on data plane nodes. The difference between the minimal and maximal median values for the control plane type is 8.74 %, while the respective average difference for the remaining types is 1.17 %.

A distribution of memory usage between nodes and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons, is available in Figure 8.4. Statistically significant differences can be observed regarding memory consumption between all control plane nodes. MicroK8s has the lowest utilization, while KubeEdge has the highest. There is also a statistically significant difference between the cloud nodes, with KubeEdge having a higher memory utilization. Although the edge node is the most strained node, no statistically significant difference was calculated using the Kruskal-Wallis test. The infrastructure node shows a significant difference in Kubernetes technology.

Apparent distinctions in memory usage on control plane nodes between technologies are observed. Although Kubernetes was initially observed to have the lowest median memory consumption on non-control plane nodes, there is insufficient data to support this claim for the majority of the nodes. KubeEdge is observed to have a higher memory utilization; however, this is observed only on the control plane and cloud nodes.

Containers CPU Usage [millicores] – Median

	k3s	microk8s	kubernetes	kubeedge
app-cloud	15.763	15.899	14.958	14.259
kafka-cloud	42.915	42.740	43.754	40.275
app-edge	26.267	26.550	25.625	25.094
app-sensor	26.008	27.269	26.132	24.861
kafka-edge	66.177	69.595	67.539	67.123
kafka-ui	3.852	3.876	3.797	3.658
kafka-mirror	25.549	27.334	26.583	25.370

Figure 8.5: A summary of the median CPU usage between containers and technologies. Colored gradient indicates the range from minimal to maximal values on a given row.

Containers Memory Usage [MB] – Median

	k3s	microk8s	kubernetes	kubeedge
app-cloud	132.980	136.441	115.546	116.115
kafka-cloud	414.455	417.089	393.681	399.214
app-edge	129.739	128.509	126.688	127.367
app-sensor	126.706	127.584	127.486	122.126
kafka-edge	405.189	407.973	405.546	408.484
kafka-ui	260.741	260.168	260.048	258.475
kafka-mirror	428.130	419.927	399.936	416.816

Figure 8.6: A summary of the median memory usage between containers and technologies. Colored gradient indicates the range from minimal to maximal values on a given row.

8.2.2 Performance Impact on Services

Comparisons of explored technologies and their impact on the CPU and memory usage of containers of the demonstration application were made using statistical measurements, data visualization, and statistical tests.

Comparison of CPU Usage

A summary of the median CPU usage between containers and technologies is available in Figure 8.5. A lower CPU utilization trend for containers deployed in KubeEdge clusters is observable, while a higher CPU utilization trend is observable for containers deployed in MicroK8s clusters. However, no considerable differences are observed. The average difference between minimal and maximal median values for containers is merely 2.1 millicores. This is expected to some extent as the underlying container runtime, containerd, is the same across the explored technologies.

A distribution of CPU usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons, is available in Figure 8.7. The Kafka UI container is omitted from the box plot graphs, as no statistically significant differences among the technologies were found; the same applies to the following memory utilization analysis. The container itself is not heavily utilized during a metrics collection. Statistically significant differences in CPU utilization are found between KubeEdge and the remaining technologies. Based on distributions and statistical tests, KubeEdge has one of the lowest CPU utilization impacts on containers. Although MicroK8s suggested a worse CPU impact on containers, there is insufficient data to support this claim.

Comparison of Memory Usage

A summary of the median memory usage between containers and technologies is available in Figure 8.6. Lower memory utilization can be observed for containers in Kubernetes and KubeEdge clusters. On the other hand, higher utilization is observable in the K3s and MicroK8s clusters. Considerable differences in minimal and maximal values are observed in containers deployed in the cloud and the mirroring container. The average difference between minimal and maximal median values for containers deployed in the cloud is 22.12 MB, while the value for containers deployed on the edge is 8.45 MB, which is considerably skewed by the mirroring container memory usage.

A distribution of memory usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons, is available in Figure 8.8. Containers deployed on the cloud nodes show a statistically significant difference that supports the initial claim that Kubernetes and KubeEdge containers have a lower memory consumption for containers, while K3s and MicroK8s have higher utilization. The same cannot be said for containers that are deployed on the edge. For example, the sensor container showcases a statistically significant difference that supports lower KubeEdge memory consumption. However, there are insufficient data to suggest more significant differences regarding the impact of performance due to technology.

8.3 Summary

The difficulty of deployment may play an important role in choosing a technology. There are clear distinctions in the explored technologies. From lightweight technologies that install the necessary dependencies and provide a simple installation of additional services to manually configuring and installing a range of needed dependencies and services.

A few notable observations were noted based on the collected data and analysis. There are clear distinctions in control plane node CPU and memory utilization based on the chosen technology. This may especially play an important role in decision making for edge environments due to their constraints with respect to resources and to single-node clusters where control plane components are deployed on the same node as workload containers.

Based on collected metrics, statistical tests, and observed distributions, more statistical differences and more apparent differences in distributions were found between nodes and technologies, rather than between containers and technologies. This is expected to some extent, as the underlying components affecting the containers are mostly the same, such as the container runtime and the container network interface plugin. However, trends between technologies were observed and, to some degree, demonstrated by statistical tests. More research is needed to provide a more definitive conclusion.

As such, with the analysis complete, the importance of observed performance impact of technologies and the deployment difficulty may inherently depend on a cluster's specific environment and requirements. Every cluster is unique, as are its requirements. The analysis may help cluster administrators make a more informed decision.

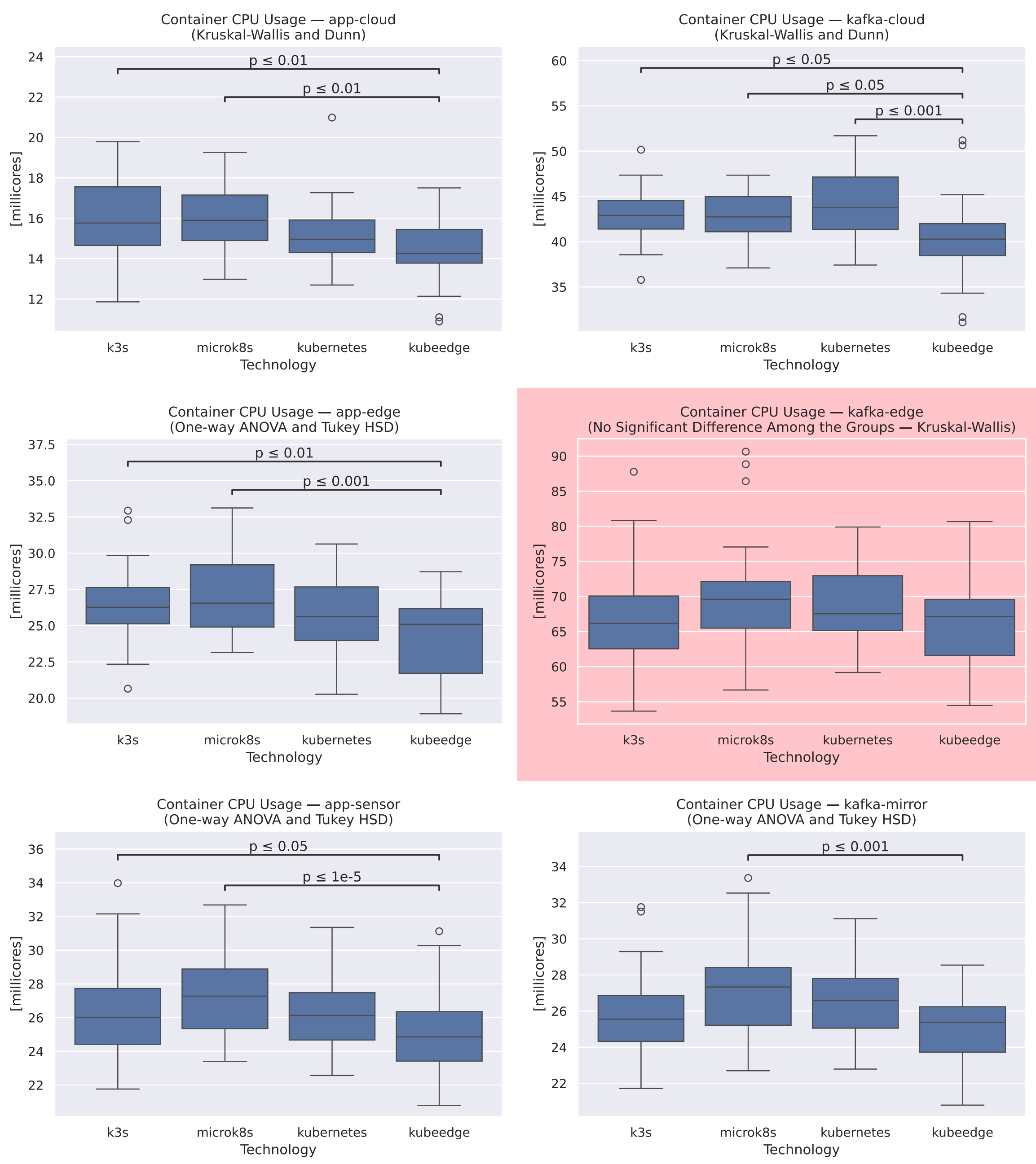


Figure 8.7: A distribution of CPU usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. A pairing annotation signifies a statistically significant difference using a post hoc test and the respective p-value. A missing pairing represents that no statistically significant difference was found.

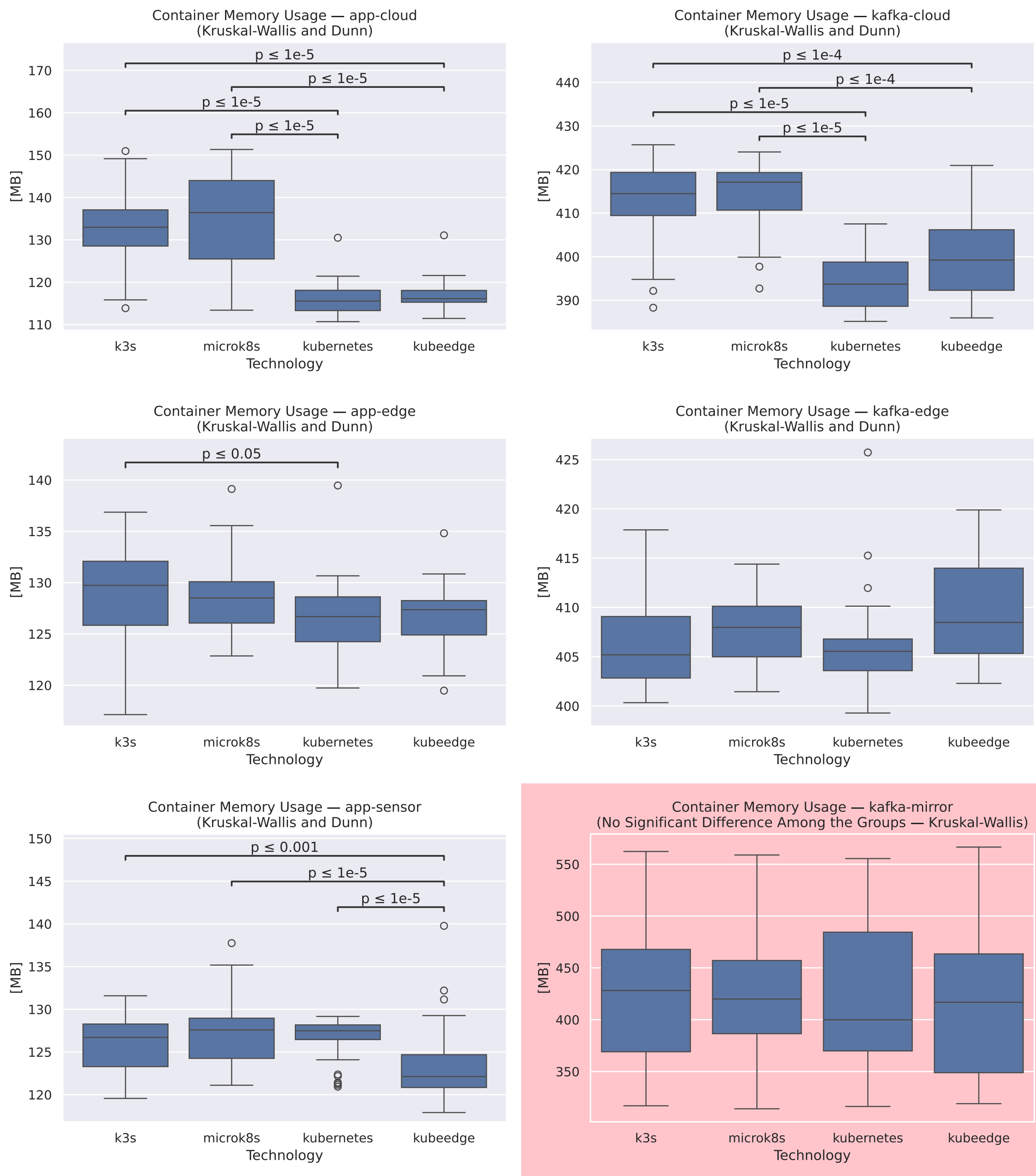


Figure 8.8: A distribution of memory usage between containers and technologies, with the corresponding p-values from post hoc tests for pairwise comparisons. A pairing annotation signifies a statistically significant difference using a post hoc test and the respective p-value. A missing pairing represents that no statistically significant difference was found.

Chapter 9

Conclusion

The goal of the thesis was to explore edge cloud environments and technologies for managing services in such environments to assist edge cloud cluster administrators in making a more informed decision when determining a technology for their clusters. The thesis explored the paradigms of cloud computing and edge computing, as well as their unification, which aims to utilize the strengths of both worlds, edge cloud computing, and the suitable devices and runtime environments of the unified paradigm. A demonstration edge cloud application was designed and implemented using Apache Kafka to highlight the benefits of edge cloud computing. The thesis explored existing technologies for deploying, operating, and managing services in the edge cloud, and their deployment tools were analyzed.

A project was designed and implemented to deploy and monitor edge cloud services to help cluster administrators easily explore the technologies. The project utilized a number of modern technologies, such as Terraform and Ansible, to provision resources and form K3s, MicroK8s, Kubernetes, and KubeEdge clusters. A CLI application was implemented to deploy and monitor services in such clusters. The project is being continuously integrated and tested using GitHub Actions to execute end-to-end testing to ensure the quality of the overall project. The deployment, usage, and performance impact of the technologies were compared. Significant differences between the explored technologies were identified using data visualization, statistical tests, and measurements of collected metrics. The project and the demonstration application were published as open-source projects in the DavidHurta/edge-cloud[1] and DavidHurta/demo-edge-cloud-app[2] GitHub repositories.

The thesis has multiple areas for further development and research, as it consists of multiple individual components. The Terraform and Ansible projects can be made more configurable to support more complex cluster topologies. For deploying and monitoring the services, an operator in a cluster using the Kubernetes API server could manage and observe services based on the user's specifications, even potentially across multiple clusters. Finally, future work could be expanded by exploring fundamentally different technologies.

I have truly enjoyed working on the thesis due to its complexity and the required learning of all of the explored aspects. Technologies such as containerization, Kubernetes, Terraform, Ansible, and Apache Kafka are being used daily in today's world, and having a closer look was delightful. It is truly remarkable to see what can be done using today's technologies and knowledge provided by universities, communities, and individuals.

[1] https://github.com/DavidHurta/edge-cloud
[2] https://github.com/DavidHurta/demo-edge-cloud-app

Bibliography

[1] Buyya, R.; Vecchiola, C. and Selvi, S. *Mastering Cloud Computing: Foundations and Applications Programming* online. 1st ed. 225 Wyman Street, Waltham, MA 02451, USA: Morgan Kaufmann, 2013. ISBN 978-0-12-411454-8. Available at: https://doi.org/10.1016/C2012-0-06719-1. [cit. 2024-09-24].

[2] Cao, J.; Zhang, Q. and Shi, W. *Edge Computing: A Primer* online. 1st ed. Cham, Switzerland: Springer Cham, november 2018. SpringerBriefs in Computer Science. ISBN 978-3-030-02083-5. Available at: https://doi.org/10.1007/978-3-030-02083-5. [cit. 2024-10-05].

[3] Cao, K.; Hu, S.; Shi, Y.; Colombo, A. W.; Karnouskos, S. et al. A Survey on Edge and Edge-Cloud Computing Assisted Cyber-Physical Systems. *IEEE Transactions on Industrial Informatics* online. 1st ed., 2021, vol. 17, no. 11, p. 7806–7819. ISSN 1551-3203. Available at: https://doi.org/10.1109/TII.2021.3073066. [cit. 2024-10-20].

[4] Case Study: BlackRock. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 8. august 2017. Available at: https://www.cncf.io/case-studies/blackrock/. [cit. 2025-05-08].

[5] Case Study: CERN. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 2. july 2019. Available at: https://www.cncf.io/case-studies/cern/. [cit. 2025-05-08].

[6] Case Study: IBM. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 16. may 2024. Available at: https://www.cncf.io/case-studies/ibmwatsonxassistant/. [cit. 2025-05-08].

[7] Case Study: OpenAI. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 25. june 2018. Available at: https://www.cncf.io/case-studies/openai/. [cit. 2025-05-08].

[8] Case Study: Spotify. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 27. june 2019. Available at: https://www.cncf.io/case-studies/spotify/. [cit. 2025-05-08].

[9] Case Study: U.S. Department of Defense. *Cloud Native Computing Foundation* online. Cloud Native Computing Foundation, 5. may 2020. Available at: https://www.cncf.io/case-studies/dod/. [cit. 2025-05-08].

[10] Chang, H.; Hari, A.; Mukherjee, S. and Lakshman, T. V. Bringing the cloud to the edge. In: Institute of Electrical and Electronics Engineers. *2014 IEEE Conference*

on Computer Communications Workshops (INFOCOM WKSHPS) online. 2014, p. 346–351. ISBN 978-1-4799-3088-3. Available at: https://doi.org/10.1109/INFCOMW.2014.6849256. [cit. 2024-10-25].

[11] Erl, T.; Puttini, R. and Mahmood, Z. *Cloud Computing: Concepts, Technology & Architecture*. 1st ed. Pearson, may 2013. ISBN 978-0-13-338752-0.

[12] Hwang, K.; Fox, G. C. and Dongarra, J. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things* online. 1st ed. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., october 2011. ISBN 978-0-12-385880-1. Available at: https://dl.acm.org/doi/10.5555/2060077. [cit. 2024-09-22].

[13] Islam Shamim, M. S.; Ahamed Bhuiyan, F. and Rahman, A. XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In: Institute of Electrical and Electronics Engineers. *2020 IEEE Secure Development (SecDev)* online. September 2020, p. 58–64. ISBN 978-1-7281-8388-6. Available at: https://doi.org/10.1109/SecDev45635.2020.00025. [cit. 2024-12-02].

[14] k0s - The Zero Friction Kubernetes. *K0s Documentation* online. Mirantis Inc. Available at: https://docs.k0sproject.io/v1.28.5+k0s.0/. [cit. 2025-05-08].

[15] K3s Project Authors. K3s - Lightweight Kubernetes. *K3s Docs* online. Revised 2025-05-05. Available at: https://docs.k3s.io/. [cit. 2025-05-08].

[16] Kafka 3.6 Documentation. *Apache Kafka* online. Apache Software Foundation. Available at: https://kafka.apache.org/36/documentation.html. [cit. 2025-05-08].

[17] Kafka Streams: Core Concepts. *Apache Kafka* online. Apache Software Foundation. Available at: https://kafka.apache.org/36/documentation/streams/core-concepts. [cit. 2025-05-08].

[18] Koziolek, H. and Eskandani, N. Lightweight Kubernetes Distributions: A Performance Comparison of MicroK8s, k3s, k0s, and Microshift. In: Association for Computing Machinery and Standard Performance Evaluation Corporation. *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering* online. New York, NY, USA: Association for Computing Machinery, 2023, p. 17–29. ICPE '23. ISBN 9798400700682. Available at: https://doi.org/10.1145/3578244.3583737. [cit. 2024-12-10].

[19] KubeEdge Project Authors. Why KubeEdge. *KubeEdge* online. Revised 2023-08-07. Available at: https://kubeedge.io/docs/. [cit. 2025-05-08].

[20] Kumari, K. A.; Sadasivam, G. S.; Dharani, D. and Niranjanamurthy, M. *Edge Computing: Fundamentals, Advances and Applications* online. 1st ed. Boca Raton, Florida, USA: CRC Press, 2022. ISBN 978-1-003-23094-6. Available at: https://doi.org/10.1201/9781003230946. [cit. 2024-10-08].

[21] Lea, P. *IoT and Edge Computing for Architects* online. 2nd ed. Birmingham, United Kingdom: Packt Publishing Ltd., march 2020. ISBN 978-1-83921-480-6. Available at: https://www.oreilly.com/library/view/iot-and-edge/9781839214806/. [cit. 2024-10-15].

[22] MELL, P. and GRANCE, T. *The NIST Definition of Cloud Computing* online. Gaithersburg, Maryland: National Institute of Standards and Technology, september 2011. Available at: https://doi.org/10.6028/NIST.SP.800-145. [cit. 2024-09-20].

[23] MicroK8s Documentation - Home. *MicroK8s* online. Canonical Ltd. Revised January 2024. Available at: https://microk8s.io/docs. [cit. 2025-05-08].

[24] MOHAN, G. Automated Functional Testing. In: MOHAN, G., ed. *Full Stack Testing* online. 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., June 2022, p. 49–95. ISBN 978-1-09810-813-7. Available at: https://www.oreilly.com/library/view/full-stack-testing/9781098108120/. [cit. 2025-02-02].

[25] MOHAN, G. Continuous Testing. In: MOHAN, G., ed. *Full Stack Testing* online. 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., June 2022, p. 97–120. ISBN 978-1-09810-813-7. Available at: https://www.oreilly.com/library/view/full-stack-testing/9781098108120/. [cit. 2025-02-02].

[26] MÉNDEZ, S. *Edge Computing Systems with Kubernetes* online. 1st ed. Birmingham, United Kingdom: Packt Publishing Ltd., october 2022. ISBN 978-1-80056-859-4. Available at: https://www.oreilly.com/library/view/edge-computing-systems/9781800568594/. [cit. 2024-10-10].

[27] Nomad versus Kubernetes. *HashiCorp Developer: Nomad* online. HashiCorp, Inc. Available at: https://developer.hashicorp.com/nomad/docs/v1.9.x/nomad-vs-kubernetes. [cit. 2025-05-09].

[28] ROSSO, J.; LANDER, R.; BRAND, A. and HARRIS, J. Deployment Models: Managed Service Versus Roll Your Own. In: ROSSO, J.; LANDER, R.; BRAND, A. and HARRIS, J., ed. *Production Kubernetes: Building Successful Application Platforms* online. 1st ed. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., March 2021, chap. 2, p. 24–26. ISBN 978-1-492-09230-8. Available at: https://www.oreilly.com/library/view/production-kubernetes/9781492092292/. [cit. 2025-02-08].

[29] THE ANSIBLE PROJECT CONTRIBUTORS. Getting started with Ansible. *Ansible Community Documentation* online. Revised 2025-05-05. Available at: https://docs.ansible.com/ansible/latest/getting_started/index.html. [cit. 2025-05-09].

[30] THE ANSIBLE PROJECT CONTRIBUTORS. Introduction to Ansible. *Ansible Community Documentation* online. Revised 2025-05-05. Available at: https://docs.ansible.com/ansible/latest/getting_started/introduction.html. [cit. 2025-05-09].

[31] THE ANSIBLE PROJECT CONTRIBUTORS. Ansible concepts. *Ansible Community Documentation* online. Revised 2025-05-05. Available at: https://docs.ansible.com/ansible/latest/getting_started/basic_concepts.html. [cit. 2025-05-09].

[32] The Cluster API Project Authors. Kubernetes Cluster API. *The Cluster API Book* online. Available at: https://release-1-8.cluster-api.sigs.k8s.io/. [cit. 2025-03-28].

[33] The Cluster API Project Authors. Provider Implementations. *The Cluster API Book* online. Available at: https://release-1-8.cluster-api.sigs.k8s.io/reference/providers. [cit. 2025-03-28].

[34] The GitHub Docs Project Contributors. Understanding GitHub Actions. *GitHub Docs* online. GitHub, Inc. Available at: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions. [cit. 2025-3-16].

[35] The Kubernetes Authors. Overview. *Kubernetes* online. Revised 2024-09-11. Available at: https://kubernetes.io/docs/concepts/overview/. [cit. 2025-05-09].

[36] The Kubernetes Authors. Cluster Architecture. *Kubernetes* online. Revised 2024-10-20. Available at: https://kubernetes.io/docs/concepts/architecture/. [cit. 2025-05-09].

[37] The Kubernetes Authors. Creating a cluster with kubeadm. *Kubernetes* online. Revised 2024-10-16. Available at: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/. [cit. 2025-05-09].

[38] The Kubernetes Authors. Considerations for large clusters. *Kubernetes* online. Revised 2024-04-26. Available at: https://kubernetes.io/docs/setup/best-practices/cluster-large/. [cit. 2025-05-09].

[39] The Kubernetes Authors. Kubeadm. *Kubernetes* online. Revised 2023-07-12. Available at: https://kubernetes.io/docs/reference/setup-tools/kubeadm/. [cit. 2025-05-09].

[40] What is Terraform? *HashiCorp Developer: Terraform* online. HashiCorp, Inc. Available at: https://developer.hashicorp.com/terraform/intro/v1.9.x. [cit. 2025-1-25].

[41] Xiong, Y.; Sun, Y.; Xing, L. and Huang, Y. Extend Cloud to Edge with KubeEdge. In: Institute of Electrical and Electronics Engineers and Association for Computing Machinery. *2018 IEEE/ACM Symposium on Edge Computing (SEC)* online. 2018, p. 373–377. ISBN 978-1-5386-9445-9. Available at: https://doi.org/10.1109/SEC.2018.00048. [cit. 2024-10-01].