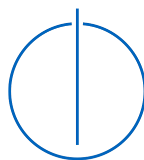
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Orchestrating Serverless Functions on Edge-Cloud Infrastructure based on KubeEdge

Max Eisner



DEPARTMENT OF INFORMATICS

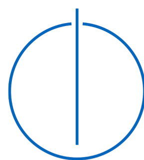
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Orchestrating Serverless Functions on Edge-Cloud Infrastructure based on KubeEdge

Orchestrierung von serverlosen Funktionen auf Edge-Cloud-Infrastrukturen basierend auf KubeEdge

Author:	Max Eisner
Supervisor:	Prof. Dr. Michael Gerndt
Advisors:	M.Sc. Anshul Jindal, Gert Glatz
Submission Date:	15.04.2022



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

Munich, 15.04.2022 Max Eisner

Abstract

Serverless computing, mostly powered by Function-as-a-Service (FaaS), has been a growing field for several years now. In addition, the field of *Edge computing* has seen simultaneous growth as well, especially in sectors such as mobile computing, Internet of Things and Industry 4.0. Both approaches come with various advantages, although they are sometimes quite contrary to each other.

Now, a combination of the two paradigms for FaaS might be feasible to achieve to exploit their respective advantages. Yet, implementations for heterogeneous edge-cloud FaaS-environments have not yet been attempted, although the groundwork has been laid by edge-only FaaS-approaches such as *tinyFaaS* [1] or *Lambda@Edge* [2].

In this thesis, an architecture for heterogeneous edge-cloud infrastructures based on technologies such as KubeEdge has been designed and implemented, as well as an automation to reproducibly deploy the infrastructure and all its components.

As a result, the infrastructure enables running a serverless computing framework such as OpenFaaS on a Kubernetes cluster consisting of both cloud and edge nodes whilst leveraging knowledge of, among other factors, physical node placement to optimize FaaS workload scheduling between cloud and edge nodes.

Serverless computing, meist auf der Grundlage von Function-as-a-Service (FaaS), ist bereits seit einigen Jahren ein stetig wachsender und weiterentwickelter Bereich. Parallel dazu hat sich auch das *Edge-Computing* entwickelt, insbesondere in Bereichen wie Mobile Computing, Internet of Things und Industrie 4.0. Beide Ansätze bringen zahlreiche Vorteile mit sich, auch wenn sie manchmal recht konträr zueinander sind.

Nun könnte eine Kombination der beiden Paradigmen für FaaS realisierbar sein, um ihre jeweiligen Vorteile zu nutzen. Bisher wurden jedoch noch keine Implementierungen für heterogene Edge-Cloud-FaaS-Umgebungen erforscht, obwohl der Grundstein durch reine Edge-FaaS-Ansätze wie *tinyFaaS* [1] oder *Lambda@Edge* [2] gelegt wurde.

In dieser Thesis wurde eine Architektur für heterogene Edge-Cloud-Infrastrukturen auf Basis von Technologien wie KubeEdge entworfen und implementiert, sowie eine Automatisierung zur reproduzierbaren Bereitstellung der Infrastruktur und aller ihrer Komponenten.

Als Ergebnis ermöglicht die Infrastruktur die Ausführung eines serverlosen Computing-Frameworks wie OpenFaaS auf einem Kubernetes-Cluster, das sowohl aus Cloud- als auch aus Edge-Knoten besteht, während unter anderem Wissen über die Platzierung der physischen Knoten genutzt werden kann, um die Verteilung der FaaS-Workloads zwischen Cloud- und Edge-Knoten zu optimieren.

Contents

Abstract iii

Acronyms vii

Glossary ix

1. Introduction 1
1.1. Motivation . . . 1
1.2. Problem Statement and Objectives . . . 2
1.3. Contributions . . . 3
1.4. Outline . . . 3

2. Background 5
2.1. Serverless Functions / Function-as-a-Service (FaaS) . . . 5
2.2. Used Technologies . . . 5
2.2.1. Kubernetes . . . 5
2.2.2. KubeEdge . . . 6
2.2.3. OpenFaaS . . . 6
2.2.4. Terraform . . . 6
2.2.5. Ansible . . . 7
2.2.6. KubeSphere . . . 7
2.3. Taxonomy . . . 7

3. Related Work 9
3.1. tinyFaaS . . . 9
3.2. K3s . . . 9
3.3. MicroK8s . . . 10
3.4. OpenYurt . . . 10

4. Implementation 11
4.1. Overall Architecture . . . 11
4.2. Infrastructure Setup . . . 11
4.2.1. Hosting Environment and Hardware . . . 11
4.2.2. Kubernetes . . . 13
4.2.3. KubeEdge . . . 14
4.2.4. OpenFaaS . . . 14
4.3. Automated and Reproducible Infrastructure Deployment . . . 15

4.4. Scheduling of Functions 16
4.5. Challenges, Caveats and Failed Attempts 18
4.5.1. KubeSphere Setup 18
4.5.2. KubeSphere-KubeEdge Integration 20
4.5.3. KubeEdge EdgeMesh / Cross-Cloud-Edge Networking 22
4.5.4. Cluster Monitoring 22
4.5.5. NodePort Routing from Cloud to Edge 23
4.5.6. OpenFaaS Portal 24
4.5.7. Readiness / Liveness Probes on Edge 25

5. Evaluation **27**
5.1. Method 27
5.2. Scenarios and Objectives 28
5.3. Examined Functions 30
5.4. Results 31
5.4.1. `nodeinfo` (CPU) 31
5.4.2. `gzip-compression` (Memory & CPU) 33
5.4.3. `dd` (Disk I/O) 35
5.4.4. `shasum` (Network I/O & CPU) 37

6. Conclusion and Future Work **41**

List of Figures **43**

List of Tables **45**

List of Code Snippets **47**

Bibliography **49**

A. Appendix **53**

Acronyms

AWS Amazon Web Services. 1, 5, 6

Azure Microsoft Azure. 1, 5, 6

CNCF Cloud Native Computing Foundation. 5–7, 9, 10, 22

CNI Container Network Interface. 8–10, 19–21

CRI Container Runtime Interface. 19

FaaS Function-as-a-Service. iii, ix, 1–3, 5, 9–11, 18, 27, 39, 41, 42

GCP Google Cloud Platform. 1, 5, 6

HPA Horizontal Pod Autoscaler. 8, 13, 23, 27, 28, 31–33, 47

IoT Internet of Things. iii, ix, 1, 2, 9, 10, 41

K8s Kubernetes. iii, ix, 2, 3, 5–11, 13–16, 18–25, 27, 32, 41, 42

VU Virtual User. 27, 31, 33–40

Glossary

Amazon Web Services [3] Cloud computing provider owned by Amazon. vii, ix, 1

Ansible [4] Open-Source tool for automation of configuration and administration of nodes via SSH by declaring repeatable tasks in YAML, see also Section 2.2.5. 3, 7, 16

Google Cloud Platform [5] Cloud computing provider owned by Google. vii, ix, 1

KubeEdge [6] Kubernetes extended infrastructure for edge & Internet of Things workloads, see also Section 2.2.2. iii, 2, 3, 6, 7, 9–11, 13, 14, 18–24, 39, 41, 47

Kubernetes [7] Open-Source system for automating deployment, scaling, and management of containerized applications, see also Section 2.2.1. iii, vii, ix, 2, 5, 6, 10, 13, 23

KubeSphere [8] Distributed operating system for cloud-native application management, using Kubernetes as its kernel, see also Section 2.2.6. 3, 7, 18–23

Microsoft Azure [9] Cloud computing provider owned by Microsoft. vii, ix, 1

OpenFaaS [10] Open-Source Function-as-a-Service framework, see also Section 2.2.3. iii, 3, 5, 6, 11, 14–16, 18, 23, 24, 27, 30, 41, 42, 47

OpenStack [11] Open-Source project providing an architecture stack for cloud computing. ix, 3, 6, 11, 13, 16

Terraform [12] Open-Source tool developed by HashiCorp to declaratively manage infrastructure as code, supporting numerous cloud providers, including Amazon Web Services, Google Cloud Platform, Microsoft Azure, and OpenStack, see also Section 2.2.4. 3, 6, 16

1. Introduction

1.1. Motivation

Serverless computing, mostly powered by Function-as-a-Service (FaaS), has been a growing field for several years now. All relevant cloud providers offer services to run FaaS workloads, such as Amazon Web Services (AWS) *Lambda*, Google Cloud Platform (GCP) *Functions* or Microsoft Azure (Azure) *Functions*. Together with the usage of a serverless computing platform comes a plethora of advantages. To name a few, as advertised by the aforementioned cloud providers [13, 14, 15]:

- no need for provisioning or managing infrastructure
- simple and automatic up- and downscaling depending on the request rate to respond to different request scales
- save cost for infrastructure by not needing to provision for peak request load
- integrated monitoring, logging and debugging capabilities
- no vendor-lock-in by using Open-Source-FaaS-Framework

In addition, the field of *Edge computing*, sometimes also known as *Fog computing*, pioneered by Content Delivery Networks, has seen simultaneous growth as well, especially in sectors such as mobile computing, Internet of Things (IoT) and Industry 4.0. Edge computing can be seen as quite contrary to cloud computing, a paradigm of moving away from centralized computing power such as in datacenters of cloud providers and towards decentralized computing on nodes at the edge of the Internet, in physical proximity to mobile devices, sensors, robots, connected industrial machines etc. [16] This approach in turn also offers a multitude of advantages, mainly enabled by the physical proximity:

- low latency computation close to physical position, important for emerging applications such as self-driving cars, IoT, Augmented Reality and Virtual Reality [16, 17]
- reduce amount of data that is transmitted to and processed by the cloud, improving interactivity and user privacy [17]
- reduced energy and power consumption [18]

However, edge computing suffers from an important limitation: resources such as CPU power, RAM and disk availability are limited, contrary to the virtually endless computing power available to the cloud. Additionally, failover strategies are harder to establish given the physical nature of edge computing units.

Now, a combination of the two paradigms for FaaS might be feasible to achieve to exploit their respective advantages.

The following non-exhaustive list gives a glimpse of possible achievements realizable by implementing such a hybrid edge-cloud architecture for FaaS:

- Edge-Cloud-Continuum

  Achieve an Edge-Cloud-Continuum between nodes to schedule FaaS edge & cloud workloads intelligently and transparently on the same cluster containing both edge and cloud nodes. This removes the need to maintain two separate infrastructures, say, one Kubernetes (K8s) cluster consisting solely of cloud nodes and one cluster consisting solely of edge nodes.

- Latency

  Functions that work on data from other edge hardware, such as IoT devices, will operate with lower latency if also scheduled on edge nodes close to where the data is generated. In addition, functions that heavily communicate with or invoke each other are likely to be scheduled close to each other on the same hardware or hardware type (edge or cloud), potentially decreasing latency even further.

- Network bandwidth

  Given that communicating workloads are scheduled close to each other, less network bandwidth is required: data transfers to cloud nodes via the internet or local network might not even be necessary if workloads are operating on the same node. As a result, network capacities are freed up to be used by workloads which actually need to communicate with cloud nodes.

1.2. Problem Statement and Objectives

As outlined in Section 1.1, both edge and cloud computing approaches come with their own specific advantages. Now, a combination of the two paradigms for FaaS might be feasible to achieve to exploit their respective advantages, and the basis for this approach has been described as “Osmotic computing” in 2016 already [19]. Yet, implementations for heterogeneous edge-cloud FaaS-environments have not been attempted, although the groundwork has been laid by edge-only FaaS-approaches such as *tinyFaaS* [1] or *Lambda@Edge* [2] as well as by projects extending K8s to the edge, such as *KubeEdge*.

We want to build upon that research and want to elevate knowledge about e.g. which nodes run in cloud and which ones run in edge contexts, or physical node location—information held by a K8s distribution itself or by a hybrid cloud K8s platform such

as *KubeSphere*. To achieve this, an architecture for heterogeneous edge-cloud FaaS-infrastructures is to be designed and implemented using tools like the aforementioned KubeEdge and KubeSphere.

The main objective is to run a serverless framework such as OpenFaaS on a K8s cluster consisting of both cloud and edge nodes. At the same time, we want to leverage K8s' knowledge of, among other factors, physical node placement to influence and optimize scheduling of FaaS workloads between cloud and edge nodes.

Ideally, we want to implement some kind of tool to help deploy and configure the infrastructure and to help with management of deployed FaaS workloads in terms of scheduling configuration and cloud / edge placement.

After designing and implementing the architecture and tool successfully, we want to thoroughly test and evaluate our work.

1.3. Contributions

Several contributions as part of this thesis have been crucial to achieve the objectives listed in Section 1.2.

The most important contribution is the design and implementation of an architecture to run the OpenFaaS framework and corresponding functions on a hybrid edge-cloud-infrastructure based on KubeEdge, leveraging the power of edge and serverless computing with manageable overhead by moving management components to the cloud.

Additionally, an automation based on Terraform and Ansible has been developed to automatically deploy and configure the architecture and all its related components on an OpenStack-based cloud in a reproducible manner.

Furthermore, said infrastructure implementation has been tested, benchmarked and evaluated according to multiple criteria to assess its value and to evaluate whether the goals stated in Sections 1.1 and 1.2 have been reached.

1.4. Outline

Following this introductory chapter, we will first give an overview on the background and technologies upon which this thesis is based in Chapter 2. This also includes significant or essential terms used throughout the thesis and relating to the applied technologies and tools, which are fundamental for understanding certain concepts.

After shining a light on the technical background in Chapter 2, Chapter 3 contains an overview of research and technologies which relate to, or aim to achieve goals similar to what is planned to achieve in this thesis.

Chapter 4 is the first of two longer chapters, which are also the most important ones as they contain the majority of work done. It describes the designed architecture and the steps taken to implement said architecture in infrastructure, including hardware specifications as well as tool configurations. The chapter also contains an overview of

the aforementioned automation that allows for reproducible deployment, as well as details regarding scheduling configuration and parameters for function deployments involving edge nodes. The final part of the chapter addresses a number of encountered challenges during implementation as well as some caveats concerning the implemented solution.

After implementing our architecture, naturally we want to analyze and evaluate our approach. Immediately following the implementation details, Chapter 5 contains detailed information about the evaluation method as well as analyses of several benchmarks of the implemented infrastructure.

Finishing up, Chapter 6 concludes the thesis and summarizes our work. In addition, we highlight possible future work and research that may be performed based on the work of this thesis.

2. Background

This chapter provides information about several topics, technologies and concepts that our work is based on. It gives an overview of significant terms and concepts used throughout the thesis, as well as a summary of the technologies that Chapter 4 (Implementation) builds upon.

2.1. Serverless Functions / Function-as-a-Service (FaaS)

Serverless Functions, also called Function-as-a-Service (FaaS) workloads, are the core of the "Serverless" cloud computing execution model.

A function is a piece of code that is uploaded to, deployed, managed and invoked by a FaaS platform and runtime such as OpenFaaS. Functions can be written in any programming language, given the targeted FaaS platform provides a runtime environment for the one chosen.

The function itself must contain a handler function, which may receive a payload upon which the function can operate. Invocations may happen due to an HTTP(S) request targeted directly at the function by a user, or due to any other event tied to the function, such as triggers fired upon arbitrary changes in other components of a cloud infrastructure.

Function instances can be up- and downscaled depending on the current request load, resulting in efficient management of cloud resources and minimization of costs, especially when using an external cloud provider.

All relevant cloud providers offer services to run FaaS workloads, such as AWS Lambda, GCP Functions or Azure Functions.

2.2. Used Technologies

The following section briefly describes all major technologies that find employment throughout this thesis.

2.2.1. Kubernetes

Kubernetes (K8s) [7] is an Open-Source system for the automated deployment, scaling, and management of containerized applications, originally developed by Google and nowadays maintained by the Cloud Native Computing Foundation (CNCF). It provides a common API for large-scale application and container management across hundreds

of computing nodes, with resources such as Pods, Deployments or Services declared and managed as YAML manifests.

Besides the official K8s distribution, several certified and interoperable downstream distributions are available, such as *Rancher RKE*[1], *Red Hat OpenShift*[2] or *K3s* [20].

In addition, all relevant cloud providers offer some form of managed Kubernetes-as-a-Service: *EKS* (AWS), *GKE* (GCP) or *AKS* (Azure).

2.2.2. KubeEdge

KubeEdge [6] is a "*Kubernetes Native Edge Computing Framework*", an incubating project under the umbrella of the CNCF just like K8s. It enables the orchestration and management capabilities of K8s for hosts at the edge with limited resources.

To achieve this, KubeEdge delegates some responsibilities of the K8s *kubelet*, which normally runs on each node, from edge nodes to other cloud nodes in the cluster to a component called *CloudCore*. Remaining on the edge node is a component called *EdgeCore*, communicating with CloudCore and accommodating for edge-specific events such as a network failure to keep applications running.

2.2.3. OpenFaaS

OpenFaaS [10] is an Open-Source framework for creating, deploying and managing serverless functions, mainly targeting K8s for deployment. Functions can be created, managed and deployed via the `faas-cli` [21] or from a web interface. Deployed functions on a K8s cluster behave like any other native K8s resource and can be managed, altered and orchestrated as such.

2.2.4. Terraform

Terraform [12] is an Open-Source tool developed by HashiCorp to manage infrastructure as code in a declarative manner. It supports numerous cloud providers, including AWS, GCP, Azure, as well as OpenStack, the one used for the Implementation (see Section 4.2.1), by using so-called plugin *providers*.

Infrastructure is managed by declaring resources in the HashiCorp Configuration Language (HCL) or JSON syntax, and modules that are reusable across different projects can be implemented, too. Terraform can manipulate infrastructure in any given, deviating state, so that the result adheres to the desired, declared state after applying the tool. The desired state may very well be dynamically determined by processing input from variable files at runtime.

[1] `https://rancher.com/products/rke`
[2] `https://www.redhat.com/en/technologies/cloud-computing/openshift`

2.2.5. Ansible

Ansible [4] is an Open-Source tool developed by Red Hat used for the automation of configuration and administration of computing nodes.

Ansible connects to configurable nodes via SSH, has no other prerequisites, and is driven by declarative, repeatable tasks and roles in the YAML language. Tasks, roles, templates, and the use of variable files facilitate code reuse, thus configurations can be applied and adapted to multiple projects easily.

2.2.6. KubeSphere

KubeSphere [8] is "the Kubernetes platform tailored for hybrid multicloud", and calls itself an operating system for cloud-native application management, based on K8s and certified by CNCF. It features plug-in support for use cases such as DevOps and logging, and is open for third-party extensions to integrate into the platform.

Of particular interest for this thesis is the promised integration with KubeEdge, which should deliver easy setup, management and configuration of all components involved.

2.3. Taxonomy

Most of the following terms and technologies are also included in the *Glossary* of the thesis (see Page ix). However, as some terms are used or coined throughout the thesis and are most crucial to follow some concepts, the description in the glossary may not suffice. Therefore, the most important terms receive a more thorough explanation in this section.

Node

A *Node* refers to a computing unit. In our case, this term refers to the server VMs further described in Section 4.2.1. These servers are mapped one-to-one to *K8s Nodes* [22], being entities onto which K8s resources such as *Pods* can be scheduled for execution.

Pod

The term *Pod* [23] refers to the K8s concept. A Pod is a workload that consists of one or more containers, which are scheduled onto a node for execution. Pods are usually managed by *Deployments*.

Deployment

Deployments [24] are yet another K8s concept, used to declare the desired state of an application consisting of pods in declarative YAML manifests. After creating or altering a deployment, the K8s deployment controller takes actions to bring the actual state to the desired state, e.g. by adding or removing pods.

Horizontal Pod Autoscaler (HPA)

Horizontal Pod Autoscalers [25] are a K8s resource and are responsible for automatic up- and downscaling of pod-managing resources such as deployments to match the current need. Scaling decisions are based on continuously collected and evaluated metrics in order to fulfill configured target quotas such as average CPU or memory utilization across pods of a deployment.

Container Network Interface (CNI) Add-on

In order for pods to be able to communicate with each other, a *Pod network* must be deployed onto the K8s cluster. Pod networks are also called plugins, and must adhere to the Container Network Interface specification[3]. Popular examples for CNI plugins include Flannel[4] and Calico[5].

[3] `http://github.com/containernetworking/cni`
[4] `https://github.com/flannel-io/flannel`
[5] `https://github.com/projectcalico/calico`

3. Related Work

Related research exists that touches on some parts of, or aims to achieve goals similar to what is planned to achieve in this thesis. Furthermore, technologies that are similar to the KubeEdge project are continually developed in parallel.

This chapter aims to give a brief overview about both types of related work.

3.1. tinyFaaS

tinyFaaS [1], designed and implemented by Pfandzelter and Bermbach, promises to be a "lightweight FaaS platform for edge environments". The prototype is aimed at IoT applications "with a focus on performance in constrained environments".

This goal is achieved by implementing a custom FaaS platform that is optimized for resource-scarce edge nodes, and consists of a *Management service*, a *Reverse Proxy*, and several *Function Handlers* acting as the core for function execution environments. Currently, only functions written for NodeJS 10 are supported.

Additionally, the Docker container runtime is expected to be installed on the node *tinyFaaS* is to be run on. As Docker is only one flavor of containerization and container runtime, this may pose as a limitation on certain use-cases.

Unlike our work, *tinyFaaS* pursuits an edge-only FaaS approach and is not aimed at hybrid edge-cloud infrastructures. Additional edge-only FaaS approaches, proposals, and prototypes exist, but at this time, none investigates possibilities of running a FaaS platform on mixed edge-cloud clusters as is done in this thesis.

3.2. K3s

K3s [20] is a CNCF-certified K8s distribution "built for IoT & Edge computing", and calls itself "perfect for Edge".

K3s bundles all components required to successfully run a K8s cluster, such as container runtime, CNI plugin or Metrics Server [26] into one single binary. This binary promises a smaller distribution size, as well as a smaller memory footprint than upstream K8s, which is beneficial to resource-constricted edge nodes.

In contrast to KubeEdge, K3s does not use a specialized architecture to mimic kubelet behavior on edge nodes, and does not differentiate between cloud and edge nodes at all—nodes are solely classified as standard K8s control-plane or worker nodes.

3.3. MicroK8s

MicroK8s [27] is, similarly to K3s from Section 3.2, a CNCF-certified K8s distribution bundled in a single package. Calling itself "production Kubernetes for [...] Edge and IoT", it also includes "sensible defaults" for e.g. metrics as well as the capability to plug in replacements and add-ons easily.

What sets this distribution apart is the way of packaging itself—it is bundled as a single container image, which seems peculiar at first, but enables features such as simple K8s upgrades.

Nevertheless, *MicroK8s* is only suitable for single-node K8s clusters and is therefore most likely not applicable to most edge computing and FaaS use-cases.

3.4. OpenYurt

OpenYurt [28] is, similarly to KubeEdge, a project under the umbrella of the CNCF, although at the earlier *Sandbox* project stage. It is "an open platform that extends upstream Kubernetes to Edge", just like KubeEdge promises.

Similarly, *OpenYurt* employs an architecture of running a management component in the cloud, and a separate component on each edge node to accommodate for edge-specific events such as network failure. However, the component at edge does not mimic a kubelet like EdgeCore does—the edge node still runs all K8s components such as the kubelet, `kube-proxy` or CNI plugin, which removes the need for a networking component such as EdgeMesh.

This project should be closely followed and compared to the state of rivaling projects such as KubeEdge, given that it progresses further and graduates from the *Sandbox* project stage. It could prove itself as an alternative for the integration of edge nodes into a K8s cluster as a basis for running FaaS workloads in a hybrid edge-cloud environment.

4. Implementation

This chapter describes the architecture designed to achieve the objectives already mentioned in Chapter 1. It also includes information about all steps taken to implement the designed architecture in infrastructure, including hardware specifications as well as tool configurations.

A section provides an overview of the implemented automation that allows for reproducible deployments of said infrastructure. In addition, the subsequent section is devoted to detailing scheduling configuration and parameters for deployments involving edge nodes.

Lastly, we list a number of encountered challenges during implementation as well as some caveats concerning the implemented solution.

4.1. Overall Architecture

Figure 4.1 depicts the desired architecture to be implemented.

We start with hardware nodes, located both in the cloud and at edge, splitting our infrastructure in a cloud realm and an edge realm. Now, to form a K8s cluster with both types of nodes, we will setup KubeEdge. The *CloudCore* management component of KubeEdge runs on cloud nodes, its counterpart *EdgeCore* on every edge node that should participate in the cluster. The network connection between cloud and edge nodes is bridged by the KubeEdge component *EdgeMesh*, which lives inside the K8s cluster.

Furthermore, we deploy OpenFaaS inside the K8s cluster. Its management components such as the gateway will run in the cloud realm while FaaS workloads will be freely schedulable between both cloud and edge realm.

4.2. Infrastructure Setup

This section contains information and details about the hosting environment and hardware used to deploy the infrastructure, as well as details regarding the setup and configuration of employed technologies.

4.2.1. Hosting Environment and Hardware

The infrastructure hosting the architecture is provided by the OpenStack-based Leibniz-Rechenzentrum Compute Cloud [29]. For this thesis, VMs with specifications according to Table 4.1 have been used.

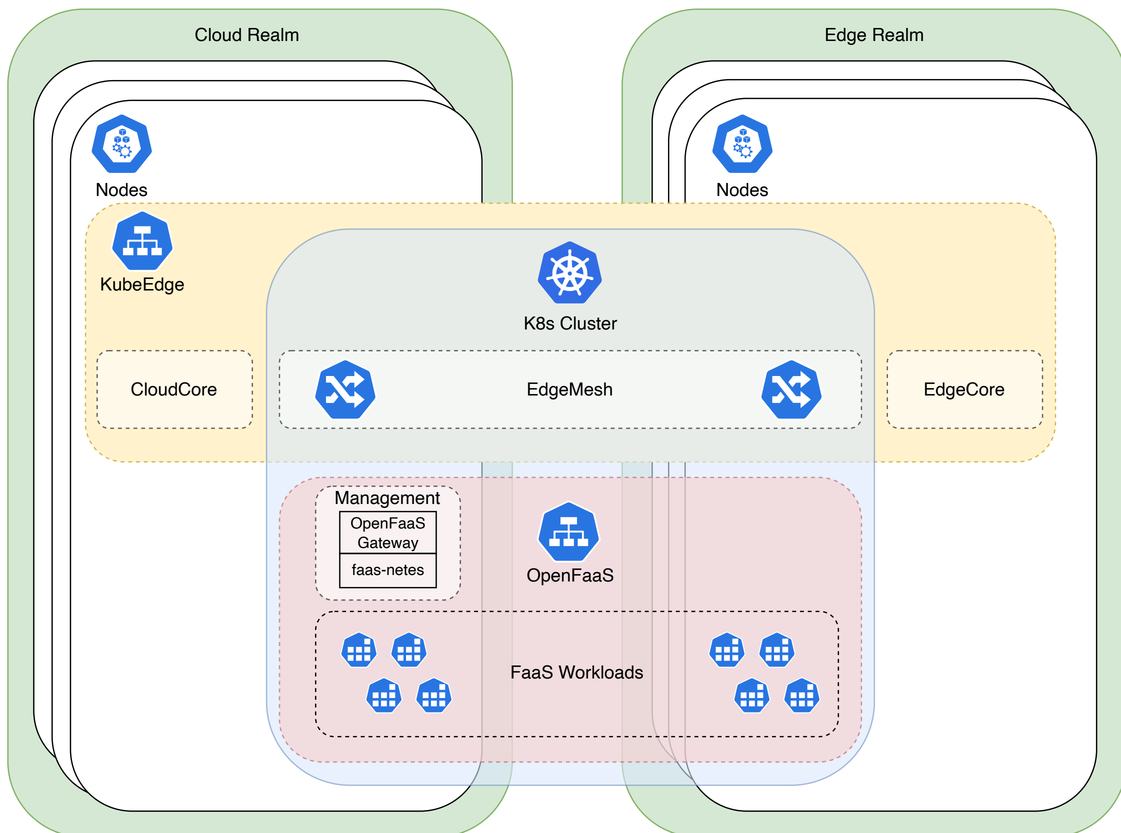


Figure 4.1.: Overall Architecture.

Table 4.1.: Hardware specifications.

VM Name & Alias		VM flavor	vCPU count	RAM	Disk size	OS
Cloud Node 1	(Main)	lrz.large	4	18 GB	20 GiB	Ubuntu 20.04
Cloud Node 2	(Worker)	lrz.medium	2	9 GB	20 GiB	Ubuntu 20.04
Edge Node 1	(Edge)	lrz.small	1	4.5 GB	20 GiB	Ubuntu 20.04

Table 4.2.: Port Requirements for inter-node communication.

Protocol	Port (Range)	Purpose
TCP	22	SSH
	53	DNS
	80	HTTP
	443	HTTPS
	6443	K8s API Server
	8443	K8s Metrics Server
	9090	Prometheus
	9100	Prometheus Node Exporter
	10000–10004	KubeEdge
	10250–10258	K8s kubelet
	10350	KubeEdge
	10550	KubeEdge EdgeMesh
	30000–32767	K8s NodePorts
UDP	53	DNS
	8285	Flannel
	8472	Flannel

The ports in Table 4.2 must be opened for incoming traffic in order to achieve successful inter-node connectivity and communication. In our case, these adjustments had to be made in the Security Group rules of our VMs in OpenStack.

4.2.2. Kubernetes

There are many ways to set up a K8s cluster, e.g. via the official `kubeadm`[1] binary provided by K8s, or via one of many third-party certified distributions. In our setup, the official upstream K8s distribution in `v1.23.4` is installed on all cloud nodes using the `kubeadm` installer. We also deploy the Kubernetes Metrics Server [26] for metrics collection to be used by HPAs. See also Section 4.5.4 for more context about Cluster Monitoring.

[1] `https://kubernetes.io/docs/setup/production-environment/tools/kubeadm`

4.2.3. KubeEdge

KubeEdge consists of two main components, also depicted in Fig. 4.1: the CloudCore component that runs in the cloud and performs management tasks of edge nodes, which in turn each run the EdgeCore component as an alternative to the classic K8s *kubelet*. Both components are installed in `v1.10.0` using the `keadm` [30] installer tool (`v1.9.1`) according to the KubeEdge installation guide [31], as seen in Code Snippet 4.1. CloudCore is installed on the K8s main node, EdgeCore is installed on each edge node that should be joined to the cluster.

```
# install CloudCore (main node)
$ keadm init --advertise-address=<PUBLIC IP OF MAIN NODE>
# get join token (main / CloudCore node)
$ keadm gettoken
# install EdgeCore (egde nodes)
$ keadm join --cloudcore-ipport=<PUBLIC IP OF MAIN NODE>:10000 \
  --token=<TOKEN>
```

Code Snippet 4.1: KubeEdge installation with `keadm`.

```
$ kubectl taint node <EDGE NODE NAME> "node-role.kubernetes.io/edge:
   NoSchedule"
```

Code Snippet 4.2: Tainting edge nodes with `NoSchedule` effect.

Each edge node is automatically labeled by the `keadm` installer with the `node-role.kubernetes.io/edge=` label to identify edge nodes. Additionally, we *taint* each edge node as seen in Code Snippet 4.2. This prevents workloads from being scheduled onto these nodes without explicitly specifying a toleration against the taint. We apply this because we want to select workloads that should be allowed to run at edge, and prevent everything else that might get deployed to the K8s cluster from accidentally being run at edge.

4.2.4. OpenFaaS

After setting up the K8s cluster according to Sections 4.2.2 and 4.2.3, OpenFaaS can be deployed to it. We follow the OpenFaaS deployment guide for K8s [32], and, to facilitate the installation process, use the `arkade` [33] installer tool.

Executing Code Snippet 4.3 will install all OpenFaaS components into the K8s cluster. The additional argument deactivates OpenFaaS Pro-specific features such as scaling down functions to zero.

All OpenFaaS management components will run in the cloud realm under the namespace `openfaas`, as depicted in Fig. 4.1, due to the *taint* on the edge nodes described in

Section 4.2.3.

With Code Snippet 4.4 we can follow the status of the deployment and wait for the OpenFaaS gateway to reach the K8s *Ready* state.

In the meantime, we can already retrieve the basic authentication credentials the deployment has created. Executing Code Snippet 4.5 will print out the username and password which can be used to access the OpenFaaS gateway to deploy and manage functions.

```
$ arkade install openfaas --set openfaasPRO=False
```

Code Snippet 4.3: OpenFaaS deployment with `arkade`.

```
$ kubectl rollout status deployment gateway --namespace openfaas
```

Code Snippet 4.4: OpenFaaS rollout status.

```
$ kubectl get secret basic-auth --namespace openfaas \
  -o jsonpath="{.data.basic-auth-user}" | base64 --decode
$ kubectl get secret basic-auth --namespace openfaas \
  -o jsonpath="{.data.basic-auth-password}" | base64 --decode
```

Code Snippet 4.5: OpenFaaS credential retrieval.

Now we just need to figure out at which IP address and port the gateway is reachable. Code Snippet 4.6 first prints the node name to which the gateway is scheduled, followed by the port where it is accessible. An alternative to access the gateway, e.g. if the node it is scheduled on is not publicly reachable, is to proxy the gateway service to `localhost` with `kubectl proxy` and accessing it from there.

To deploy and manage functions, we can either use the OpenFaaS portal via a Browser at the IP and port we retrieved with Code Snippet 4.6, or use the `faas-cli` [21]. Code Snippet 4.7 showcases how to login to the portal via the CLI and how to deploy a sample function, in this example the `figlet` function, which echoes back input in ASCII art when called at `http://<GATEWAY_IP>:<GATEWAY_PORT>/function/figlet`.

All OpenFaaS functions will be deployed inside the K8s cluster namespace `openfaas-fn`.

4.3. Automated and Reproducible Infrastructure Deployment

As all the provisioning, deployment and configuration described in Section 4.2 can be quite tedious and also error-prone if performed by hand repeatedly, especially when working and experimenting with throw-away clusters, we have developed an automation to handle these tasks [34].

```
$ kubectl get pods --selector app=gateway --namespace openfaas \
  -o jsonpath="{.items[0].spec.nodeName}"
$ kubectl get service gateway-external --namespace openfaas \
  -o jsonpath="{.spec.ports[0].nodePort}"
```

Code Snippet 4.6: OpenFaaS Gateway node and port retrieval.

```
$ export OPENFAAS_URL=<GATEWAY IP>:<GATEWAY PORT>
$ faas-cli login --username <USERNAME> --password <PASSWORD>
$ faas-cli store deploy figlet
```

Code Snippet 4.7: OpenFaaS login and function deployment via `faas-cli`.

The automation is based on the popular tools Terraform and Ansible, the former for provisioning of VMs for both cloud and edge, and the latter for deployment and configuration of the tools employed in our architecture design from Section 4.1 to behave as described in this thesis.

This results in the ability to reproducibly set up a cluster architecture like the one used in this thesis in minutes, the only prerequisite being access to an OpenStack-based cloud. Using a different cloud provider for deployment may also be possible, but requires adjustments to the Terraform providers and VM resource configurations.

4.4. Scheduling of Functions

To influence K8s scheduler decisions in the way of preferring edge nodes over cloud nodes and only scheduling to cloud nodes if resources at the edge are depleted, we make use of *K8s node affinity constraints* [35].

The configuration for *Deployments* shown in Code Snippet 4.8 has the following effect:

- Lines 10 to 13 make sure that the deployment's pods can also be scheduled on nodes tainted with the `node-role.kubernetes.io/edge:NoSchedule` taint. As all our edge nodes are tainted as such (see Section 4.2.3), to only allow selected workloads, we specify a toleration against said node taint for the respective deployment. Otherwise, the taint prevents workloads without toleration against it from being scheduled on the tainted node.

- Lines 14 to 22 make use of the aforementioned node affinity constraints, expressing a scheduling preference for nodes with the `node-role.kubernetes.io/edge` label. The `weight` value in Line 18 can be adjusted in the range of 0–100, specifying the weight of the preference with regard to other scheduling rules, effectively influencing the ratio of pods scheduled on cloud/edge nodes.

```yaml
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      [...]
      tolerations:
        - key: "node-role.kubernetes.io/edge"
          operator: Exists
          effect: NoSchedule
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            # variable [0-100] depending on preferred cloud-edge ratio
            - weight: 50
              preference:
                matchExpressions:
                  - key: "node-role.kubernetes.io/edge"
                    operator: Exists
```

Code Snippet 4.8: Deployment configuration for functions at edge.

By applying this configuration to existing deployments, such as deployments created by the `faas-cli` or OpenFaaS gateway when deploying a function, we achieve the desired scheduling effects for our FaaS workloads.

Given the case one is not working with deployments but rather ephemeral *Pods*, the significant key-value pairs in Lines 10 to 22 can also be applied to the PodSpec directly to only influence scheduling decisions for a specific Pod.

4.5. Challenges, Caveats and Failed Attempts

During implementation and experiments with infrastructure and different tools, multiple challenges presented themselves. This section lists and describes most of them in detail, including challenges that resulted in failed attempts, and additionally contains some caveats regarding the final implementation approach.

4.5.1. KubeSphere Setup

Early attempts implementing the infrastructure included an approach to deploy the K8s operating system KubeSphere as it promises out-of-the box support for KubeEdge nodes. Deployment of both K8s and KubeSphere was done using the KubeKey (`kk`) [36] installer tool with the commands shown in Code Snippet 4.9. We use an adapted cluster configuration file to enable KubeEdge integration, as shown in Code Snippet A.1.

```
$ kk create config \
  --with-kubernetes v1.21.5 \
  --with-kubesphere v3.2.1 \
  -f configuration.yaml
# adapt configuration.yaml file before executing next command
$ kk create cluster -f configuration.yaml
```

Code Snippet 4.9: Cluster creation with KubeKey.

However, as easy and seamless as installation and deployment of the desired tools seems, there were a lot of unanticipated issues and challenges both during setup and use, resulting in a setback of our implementation and evaluation. The following challenges occurred during setup of the KubeSphere environment and during early use:

1. Operating system versions for cluster nodes

   Usage of CentOS 7 and Ubuntu 18.04 LTS is recommended. Attempts at setting up a KubeSphere cluster with CentOS 8 for cloud nodes and Ubuntu 20.04 for edge nodes have been unsuccessful and unreliable, see also Challenge 2 in Section 4.5.2.

2. Choice of Container runtime

   Usage of the Docker[2] container runtime is recommended, even though Docker runtime support is deprecated in K8s from version `v1.20.0` onwards and removed altogether in version `v1.24.0` to encourage the use of a Container Runtime Interface (CRI)-compliant container runtime. `containerd` runtime support is marked as experimental and results in additional issues, see also Challenges 3 and 5 in Section 4.5.2.

3. SELinux mode

   Although SELinux may be a sophisticated Linux kernel extension to improve security and access rights management, it is also a hassle to configure correctly, even more so if one does not have an exact overview of binaries and directory permissions needed for a service to run. As this is the case for the `kubekey` installer, more specifically the Docker and `containerd` binaries installed by it, it is advised (also by KubeSphere) to set SELinux, if employed, to *Permissive* mode by editing `/etc/selinux/config` and setting `SELINUX=permissive`.

   We tried adding policies for the binaries etc. of Docker and `containerd` installed by `kubekey`, but still, runtime issues occurred which were only solved by changing the SELinux mode.

4. Choice of CNI plugin

   Use of Flannel as CNI plugin is recommended over the use of Calico (the default CNI plugin in KubeSphere).

   After setting up and using a cluster with Calico for some time, we encountered an issue where Calico randomly encountered repeated failures and brought one node into an unstable state, crashing the whole KubeSphere environment in the process. Trying to solve the issue to recover the node and KubeSphere was way more troublesome than setting up the cluster anew.

   After switching to Flannel no such issue occurred anymore. However, the consequences of Challenge 2 in Section 4.5.2 have to be considered instead.

5. Firewall settings

   Correctly adjusting firewall configurations was a challenge, as not all required open ports were documented at a single source of truth. As certain open ports are crucial for successfully running a K8s cluster and inter-node communication, also in the context of KubeEdge, missing opening a port or port range can lead to problems and challenges further down the road. See also Section 4.2.1 and Table 4.2.

6. Kubeconfig inconsistency / High Availability preference

   After successfully running `kubekey`, as shown in Code Snippet 4.9, we can access

[2] `https://docker.com`

the K8s cluster using the Kubeconfig file created by the installer located in either `/etc/kubernetes/admin.conf` or `~/.kube/config`.

Now, during setup of the cluster, `kubekey` assumed we were running in a high-availability environment and are using a load balancer, even though we did specify otherwise in the cluster configuration (Code Snippet A.1). If we want to access the cluster from outside the main node, we have to adjust the K8s API server location in our local copy of the Kubeconfig, by changing `.clusters[].cluster.server` from `https://lb.kubesphere.local:6443` to `https:<Public IP of main node>:6443`.

7. `kubekey` [36] incompatible with macOS

   The latest `kubekey` version at the time turned out to be incompatible with macOS, thus remote cluster creation failed during one of the last installation steps, the initialization of the CNI plugin. Running the tool on one of the nodes to be bootstrapped itself circumvented this issue.

4.5.2. KubeSphere-KubeEdge Integration

After working through the challenges mentioned in Section 4.5.1 regarding setup of the KubeSphere environment, several more challenges presented themselves during setup and usage of the promised KubeEdge integration.

1. Unstructured and incomplete documentation

   The documentation for the KubeEdge integration is distributed across multiple pages and guides that do not necessarily relate to each other. Additionally, some components are not documented at all, e.g. the `edgewatcher` component also mentioned in Challenge 2, which is very important for node connectivity and proprietary to KubeSphere.

   This makes figuring out issues or debugging connectivity problems quite a bit more cumbersome than it should be for a feature that has been in a stable release for some time now (KubeEdge integration debuted in `v3.1.0`, released April 2021[3]).

2. `iptables` firewall and routing configuration

   The `iptables` system is internally used both by the Flannel version shipped by KubeSphere (see Challenge 4 in Section 4.5.1), and the KubeSphere-KubeEdge integration component called `edgewatcher`, to configure network packet routing between cluster nodes (Flannel) and between cluster and edge nodes (`edgewatcher`).

   However, the `iptables` system has been replaced by the new `nftables` system in CentOS 8, the `iptables` binary used both by Flannel and `edgewatcher` is no longer available. This reinforces the advice given in Challenge 1 from Section 4.5.1 regarding OS selection.

[3] `https://github.com/kubesphere/kubesphere/releases/tag/v3.1.0`

One could of course still use another CNI plugin, however, our experience described in Challenge 4 from Section 4.5.1 suggests otherwise. Still, the `edgewatcher` dependency on `iptables` would not be solved by switching the CNI plugin.

3. Outdated KubeEdge version `v1.7.2`

   KubeSphere ships the slightly outdated KubeEdge version `v1.7.2`[4] (current version: `v1.9.1`). This version still ships with an integrated *EdgeMesh* networking component (see Section 4.5.3 for more details), which only works when using Docker as a container runtime, reinforcing the outcome of Challenge 2 from Section 4.5.1.

   In versions after `v1.7.2`, the component has been decoupled from KubeEdge.

4. `CloudHub` configuration not respected by `kubekey`

   The deployment of the CloudHub gateway, the entry point to the KubeEdge CloudCore component, has been configured in Code Snippet A.1, the configuration file used by the `kubekey` installer. Namely, the `.spec.kubeedge.cloudCore.cloudHub.advertiseAddress` value is set to the public IP of our main node to be reachable. In addition, the `.spec.kubeedge.cloudCore.nodeSelector` field instructs K8s to only schedule the component to our main cluster node.

   However, these configurations seemingly are ignored by `kubekey`, an issue we also encountered in Challenge 6 from Section 4.5.1. As a result, we need to manually patch the deployed K8s resource for CloudHub to accord to the values we originally specified for the component to be used successfully.

5. KubeEdge installer Segmentation faults

   The `keadm` installer tool used to initialize an edge node throws a `Segmentation fault` trying to install the KubeEdge version recommended by KubeSphere, `v1.7.2`, during initialization of a node configured to use `containerd` as a container runtime. This can be circumvented by using a newer version of KubeEdge, e.g. `v1.9.0`, or by using Docker as the container runtime on the edge node.

After encountering all these challenges, we came to the conclusion that the KubeEdge integration into KubeSphere is not as simple or working right out-of-the-box as promised. The main issues that we were not able to resolve were networking between cloud and edge nodes as well as DNS resolution on edge nodes, issues most likely caused by the proprietary `edgewatcher` component KubeSphere employs.

This lead to us scrapping the KubeSphere approach for our architecture altogether, experimenting with an approach based on a native K8s distribution in combination with native KubeEdge instead.

[4] `https://github.com/kubesphere/ks-installer/blob/release-3.2/roles/kubeedge/files/kubeedge/kubeedge/values.yaml`

4.5.3. KubeEdge EdgeMesh / Cross-Cloud-Edge Networking

After scrapping the KubeSphere environment and trying out a different approach on a native K8s distribution, there still were challenges to conquer. Namely, challenges regarding networking across the cloud-edge barrier, meaning communicating with an edge node from a cloud node or vice-versa. These challenges all relate to the *EdgeMesh* component of KubeEdge.

1. Outdated, incomplete and contradicting documentation

   Similar to Challenge 1 in Section 4.5.2, KubeEdge and especially the *EdgeMesh* component suffer from either outdated, incomplete or contradicting documentation, depending on the issue one is looking into.

   This has improved in parts, the EdgeMesh documentation is coherent after its decoupling from KubeEdge, but still is not as exhaustive as one wishes it to be. Also, missing version references in existing documentation certainly do not help, especially if information about the decoupling of EdgeMesh into a separate component can only be found if digging through the project's GitHub repository.

   Granted, we are talking about a project in the CNCF incubating stage, but still, one can wish for improvements as they would have made debugging issues a more pleasant experience.

2. Complex setup and configuration

   Setting up KubeEdge and EdgeMesh, one has to face a multitude of configuration options, many of which are not documented, as outlined in Challenge 1. Given the already complex topic of networking configuration in general, this adds lots of potential points of failure during setup, resulting in a setup that might not work as expected.

   We got stuck multiple times trying to configure KubeEdge and getting nodes to communicate across the cloud-edge barrier. Even after following documentation regarding setup or advice given from maintainers of the project in GitHub issues, the setup was still not functioning correctly. We tried several approaches, both with a K8s cluster bootstrapped via the official `kubeadm` installer and a cluster based on K3s [20], without success at first.

Now, even though the EdgeMesh component has been difficult to set up and troubleshoot, one of our attempts seemed to work out in the end. Although not perfect, at least with functionality sufficient for us to continue our research, deploy and invoke functions, and run our evaluation.

4.5.4. Cluster Monitoring

K8s Cluster Monitoring, meaning the continuous collection of metrics such as CPU or memory usage on a per-resource basis, is a prerequisite for the automatic up- and

downscaling of deployments based on request load. Both the K8s integrated *HPA* and the *OpenFaaS Scaler* included in OpenFaaS Pro rely on metrics data to compute scaling decisions.

When it comes to monitoring solutions for K8s clusters, there are two popular setups:

- Kubernetes Metrics Server [26], works by scraping each node's *kubelet* for metrics data from container runtime

- Prometheus[5], works by running a `node-exporter`[6] pod on each node and forwarding system data about resource usage to a central instance

Both configurations did not work out-of-the-box for the edge nodes, due to the specialized KubeEdge architecture which is not running a kubelet per se but rather emulating one. This ruled out the Prometheus setup in the end.

However, the Metrics Server approach worked with a few configuration tweaks, both on KubeEdge's and the metrics server's side. For KubeEdge, we needed to configure a mixture of concepts, namely *CloudStream*, *EdgeStream*, and *Tunnel Port*.

For the metrics server to work, we adjusted and added some command line arguments to the deployment spec, see the relevant excerpt in Code Snippet 4.10. Line 13 has been altered to prefer the external IP address of nodes for scraping kubelets, away from the default of preferring the internal IP address, otherwise the edge node can not be scraped correctly. Line 16 has been added, as the kubelet's certificates are only valid for their internal IP address, the aforementioned change therefore lets TLS connections fail due to invalid certificates. In a production environment, this change should not be applied, instead the underlying issue should be resolved by regenerating certificates for nodes whilst also including their external IP addresses.

It should also be mentioned that metrics collection did not function at all during experiments with the now-disbanded KubeSphere approach (see Section 4.5.1 for reasons), even though advertised otherwise.

4.5.5. NodePort Routing from Cloud to Edge

Normally, exposing a K8s service through the service type `NodePort` exposes a port on all nodes of the K8s cluster. Sending a request to any of the nodes at that port will relay the request to a pod belonging to the service, no matter on which node these pods are running, and especially if no pod of the service is running on the targeted node itself.

However, this type of routing does not work as expected in our implementation, or at least only partially. Requests to edge nodes will always reach their destination, even if no pod is running on the node itself, as is the expected behavior.

If a request is, however, targeted at a cloud node, and pods belonging to the targeted service are only running on edge nodes, the request will time out. Yet, if at least one

[5] `https://prometheus.io`

[6] `https://github.com/prometheus/node_exporter`

```
1  apiVersion: apps/v1
2  kind: Deployment
3  [...]
4  spec:
5    [...]
6    template:
7      [...]
8      spec:
9        containers:
10       - args:
11         - "--cert-dir=/tmp"
12         - "--secure-port=4443"
13         - "--kubelet-preferred-address-types=ExternalIP,InternalIP,Hostname"
14         - "--kubelet-use-node-status-port"
15         - "--metric-resolution=15s"
16         - "--kubelet-insecure-tls"
```

Code Snippet 4.10: Metrics server deployment manifest.

pod is also running on a cloud node in addition to edge, routing will fall back to the cloud pod and work as expected.

The cause of this abnormal behavior is unclear, but it might be due to yet another issue with the setup and configuration of the KubeEdge EdgeMesh component, as already detailed in Section 4.5.3.

4.5.6. OpenFaaS Portal

The OpenFaaS portal, which can be used to deploy, manage and invoke functions, will fail with an `Internal Server error` when invoking functions through it that exclusively have pods running on edge nodes. This behavior may be linked to the issue described in Section 4.5.5.

Curiously, accessing the pods from within the K8s network via means of either individual Pod IP, Service Cluster IP or Service DNS name in the form of `<SERVICE>.<NAMESPACE>.svc.cluster.local:<PORT>` works as expected without a problem.

As a result, this issue does not really have an impact on function usage and management, as functions can still be deployed from either the Portal or the `faas-cli` [21], invoked via means of direct requests, and configured natively in K8s via their respective resource manifests.

4.5.7. Readiness / Liveness Probes on Edge

For function deployments on edge nodes, pods scheduled on edge nodes fail to successfully answer to Readiness and Liveness probes from the kubelet, or in our case, EdgeCore. This is occurring even though the pods have started up completely and are answering to requests if invoked directly via their Pod IP. Whether EdgeCore actually does not probe pods correctly or simply fails to relay information about the outcome of probes to the K8s API Server on the main node of the cluster is unclear. This issue might as well be linked to the EdgeMesh component, as described in Section 4.5.3 and Section 4.5.5.

As a consequence, in order for deployed pods on edge to be marked as *Ready* and therefore addressable by K8s services, and to prevent incorrect restarting of pods due to failing liveness probes, the deployment's readiness and liveness probes have to be removed from the K8s resource, if present.

5. Evaluation

Now, after designing an architecture to run a FaaS-framework on a hybrid edge-cloud-infrastructure and setting up the required infrastructure to run it, we want to test and evaluate our implementation.

This chapter describes the evaluation approach used as well as the results of the evaluation tests applied to different scenarios.

5.1. Method

In order to collect data for our evaluation, we need to generate requests for functions running on the infrastructure.

To generate these requests, we rely on the tool `k6` [37]. Over the span of 3 minutes, we let `k6` generate requests appropriate to the current function under test. We configured `k6` to gradually ramp up the number of *Virtual Users (VUs)*, which basically act as `while` loops running a script generating a single request, up to a limit over the span of 2 minutes. After the ramp-up, `k6` is instructed to then continue to run for an additional minute at the height of the ramped-up VU count.

During the whole `k6` run, we continuously query the K8s API server retrieving the number of pods associated with the function under test, as well as the node on which they are running.

As a result, this data collection approach yields timestamped data about request duration associated with the overall request load represented by the VU count, as well as the number of pods responding to the requests, including their distribution across cloud and edge nodes.

The complete list of steps taken during evaluation for each function from Section 5.3 and scenario described in Section 5.2 is as follows:

1. Deploy the function, either via the OpenFaaS portal or the `faas-cli`, an example for the latter is also shown in Code Snippet 4.7.

2. Alter the generated K8s service to conform to type `NodePort` instead of `ClusterIP`, due to issues described in Section 4.5.6.

3. Define deployment resource requests for HPA, as functions do not always specify one.

4. Configure an HPA for the deployment, as shown in Code Snippet 5.1. The parameter `--max` specifies the upper limit of pod scalability, whilst `--cpu-request`

Table 5.1.: Evaluation Scenarios.

Scenario		Request Entrypoint	Initial Pod Locations	HPA	Scheduling Locations
Base	Cloud	Cloud	Cloud	Active	Cloud only
	Edge	Edge	Edge		Edge only
Evaluation	1 (E1)	Edge	Edge	Active	Cloud & Edge

specifies the average percentage of CPU load per pod that should be achieved by the HPA.

5. Configure tolerations and affinities of the deployment depending on the applied scenario from Section 5.2.

6. Remove liveness and readiness probes of deployment, due to issues described in Section 4.5.7 (only in scenarios involving edge nodes).

7. Run evaluation script based on `k6` and collect data.

```
$ kubectl autoscale <FUNCTION> --namespace openfaas-fn --max 25
  --cpu-request 50
```

Code Snippet 5.1: HPA configuration for deployments.

5.2. Scenarios and Objectives

Table 5.1 displays the different scenarios according to which we evaluate our implemented infrastructure.

The first two scenarios, *Base Cloud* and *Base Edge* have the goal of collecting data about the base performances of the two sites, by only exploiting resources of either cloud or edge nodes at the same time.

We target our generated requests only at a node belonging to the respective site under evaluation, on which the initial pod is also running. The respective deployment's HPA will also schedule pods only to the respective site due to the deployment's affinities and tolerations being adjusted in Step 5, as displayed in Code Snippet 5.2 and Code Snippet 5.3. See also Section 4.4 for more information on `nodeAffinity`, tolerations, and taints.

The third scenario, *Evaluation 1 (E1)*, aims to collect data of a hybrid edge-cloud configuration.

Requests are targeted at the node on the edge site on which the initial pod runs, partially due to the issue described in Section 4.5.5. The configured HPA will schedule

```yaml
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "node-role.kubernetes.io/edge"
                  operator: DoesNotExist
```

Code Snippet 5.2: Deployment configuration for Scenario Base Cloud.

```yaml
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      tolerations:
        - key: "node-role.kubernetes.io/edge"
          operator: Exists
          effect: NoSchedule
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                - key: "node-role.kubernetes.io/edge"
                  operator: Exists
```

Code Snippet 5.3: Deployment configuration for Scenario Base Edge.

to both cloud and edge nodes in this scenario, according to the deployment affinities and tolerations displayed in Code Snippet 5.4. This configuration resembles the one displayed in Code Snippet 4.8, the only change being the weight parameter, modified from a moderate `50` to `80` (out of `100`) to express a stronger preference of edge nodes over cloud nodes.

Apart from performance data, we also want to collect data about the pod distribution across cloud and edge in this scenario. This will allow evaluating scalability on edge and to test if pods overflow to cloud nodes to handle request load in case the resources of the edge site are exceeded. The impact on performance in this case is of interest, too.

```yaml
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      tolerations:
        - key: "node-role.kubernetes.io/edge"
          operator: Exists
          effect: NoSchedule
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
            - weight: 80
              preference:
                matchExpressions:
                  - key: "node-role.kubernetes.io/edge"
                    operator: Exists
```

Code Snippet 5.4: Deployment configuration for Scenario E1.

5.3. Examined Functions

Table 5.2 lists all serverless functions examined and used for data collection during the evaluation, including a short description of each function.

The functions originate either from the OpenFaaS Store [38] or are custom functions also used by Jindal et al. in “Estimating the Capacities of Function-as-a-Service Functions” [39]. Each function serves a different purpose, meaning they each target one or more specific resources of the host they are running on, such as CPU, Memory, Network

Table 5.2.: Functions used for Evaluation.

Name	Description	Purpose	Origin
`nodeinfo`	Returns basic characteristics about node, such as CPU count and hostname	CPU-intensive under heavy load	[38]
`gzip-compression`	Creates a 10 MB file with random data and compresses it using `gzip`	Memory & CPU-intensive	Custom
`dd`	Converts a 128 byte block 10 times using UNIX `dd` command	Disk I/O-intensive	Custom
`shasum`	Generates a shasum for the given input, in our case a 1 MB file with random data	Network I/O & CPU-intensive	[38]

or Disk I/O.

5.4. Results

Now, after describing the process of data collection, we want to have a look at the resulting data. The following section describes and interprets the sampled data visualized by plots for each examined function.

5.4.1. `nodeinfo` (CPU)

Figure 5.1 shows the results according to the Scenarios Base Cloud and Base Edge from Section 5.2 for the `nodeinfo` function, which is rather CPU-heavy on pods and their hosting nodes under heavy request load.

We can observe a relatively stable response time up to approximately 150ms on both cloud and edge, disregarding some outliers pushing request durations up to 300ms and 400ms respectively for cloud and edge. The increasing request load represented by the climbing number of VUs seems to have no effect on request durations, due to scaling efforts on the respective nodes.

Regarding scaling, in the Base Cloud scenario, the deployment is upscaled up to the maximum number of pods allowed by the HPA, increasing together with the request load, while the number of pods maxes out at 6 in the Base Edge scenario. This can be explained by the fact that the Cloud site simply has more resources in terms of CPU

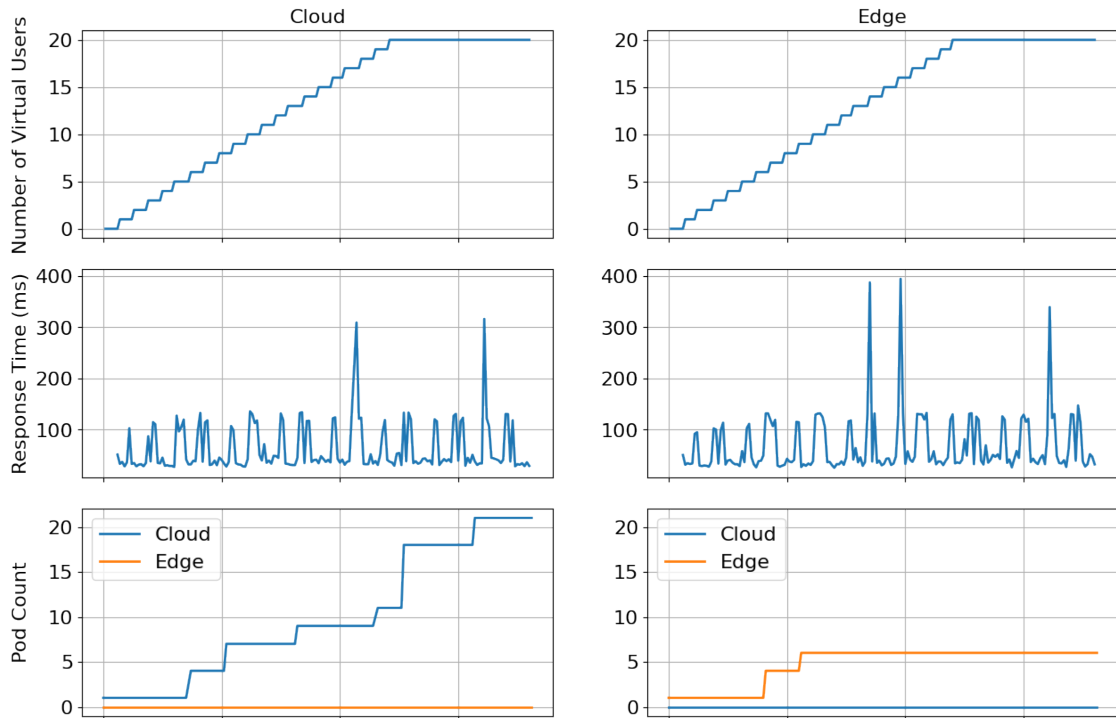


Figure 5.1.: `nodeinfo` Base Scenarios.

capacity at hand (see Table 4.1), and the Edge site can not handle more than 6 pods of the deployment at a time. Nevertheless, this seems to have no effect on the request durations. It is likely the HPA would have scaled up the deployment given enough resources to fulfill the CPU request requirement across pods, but pods were not yet as overwhelmed as to have an impact on response times.

It is to be mentioned that during data collection in these scenarios, the metrics scraping performed by the metrics server sometimes deviated from the defined resolution of 15 seconds when scraping the edge node. This is due to metrics scraping requests to the kubelet, or in this case, CloudCore together with EdgeCore on the edge node, timing out, meaning EdgeCore was not able to tunnel metrics data back to CloudCore for consumption by the metrics server when under heavy load. However, this does not seem to have an effect on the collected data, as this issue only occurred when the deployment on edge had already been maxed out due to exhausted resource capacity.

Figure 5.2 shows the collected data relating to the Scenario E1 from Section 5.2.

We can observe request time behavior similar to that of Scenarios Base Cloud and Edge, with the request time being consistently under 150ms, disregarding one outlying request again. However, two average request duration trends can be observed too, one lying between 100 and 150ms, the second one being around or under 50ms per request. One possible explanation could be delays caused by internal K8s service load balancing

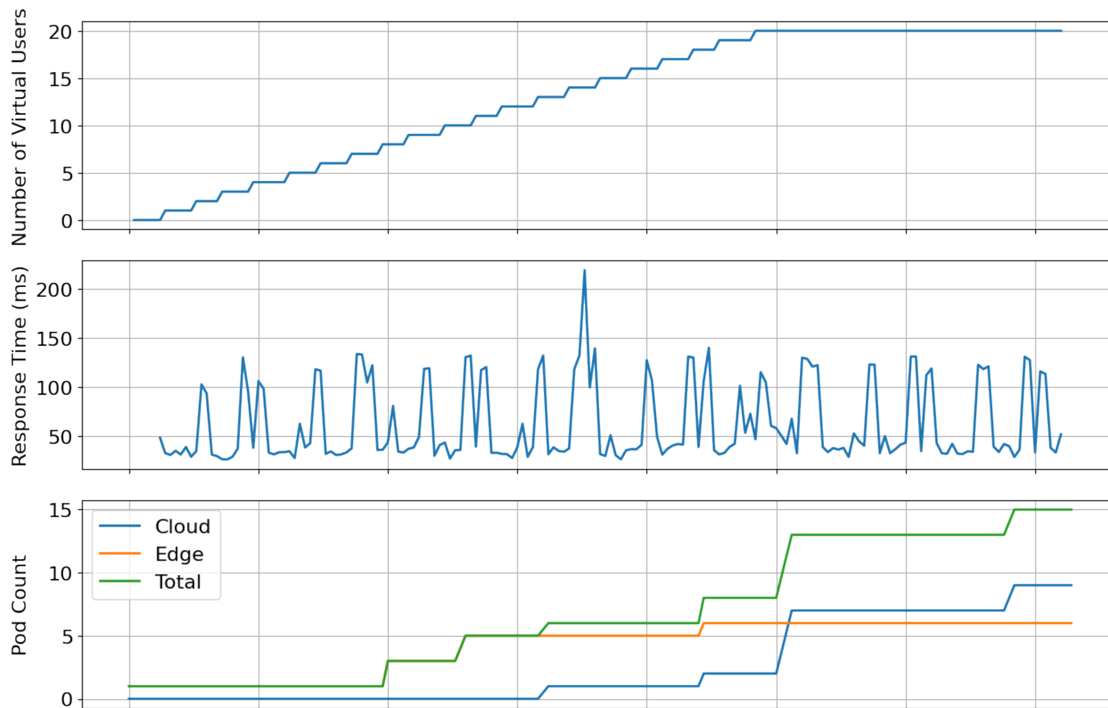


Figure 5.2.: `nodeinfo` Scenario E1.

and routing distributing requests across different nodes and possibly even the barrier between cloud and edge nodes.

Having a look at the scaling behavior, we can see that at first, additional pods were scheduled at edge, only when edge CPU capabilities were nearly exhausted, pods were also scheduled to cloud nodes to cope with the increasing request load, eventually overtaking the edge pod count.

Overall, the combination of cloud and edge nodes seems to have handled the request load better, as the maximum number of scheduled pods reached only 15, as opposed to 25 in Scenario Base Cloud. Yet, this could also be due to the aforementioned issue with metrics scraping of pods at edge under heavy load, thus reported CPU usage of edge nodes deviating from actual usage, potentially leading to skewed HPA behavior.

5.4.2. `gzip-compression` (Memory & CPU)

Figure 5.3 shows the results according to the Scenarios Base Cloud and Base Edge from Section 5.2 for the `gzip-compression` function, which mainly uses memory and CPU resources of the hosting node.

For the Base Cloud scenario, we observe request times ranging from under 1 second up to approximately 4 seconds, with a steady incline relating to the increasing request load represented by the VU count. Due to higher resource usages of the function, we examine that the deployment is upscaled to the maximum capability of 6 pods already

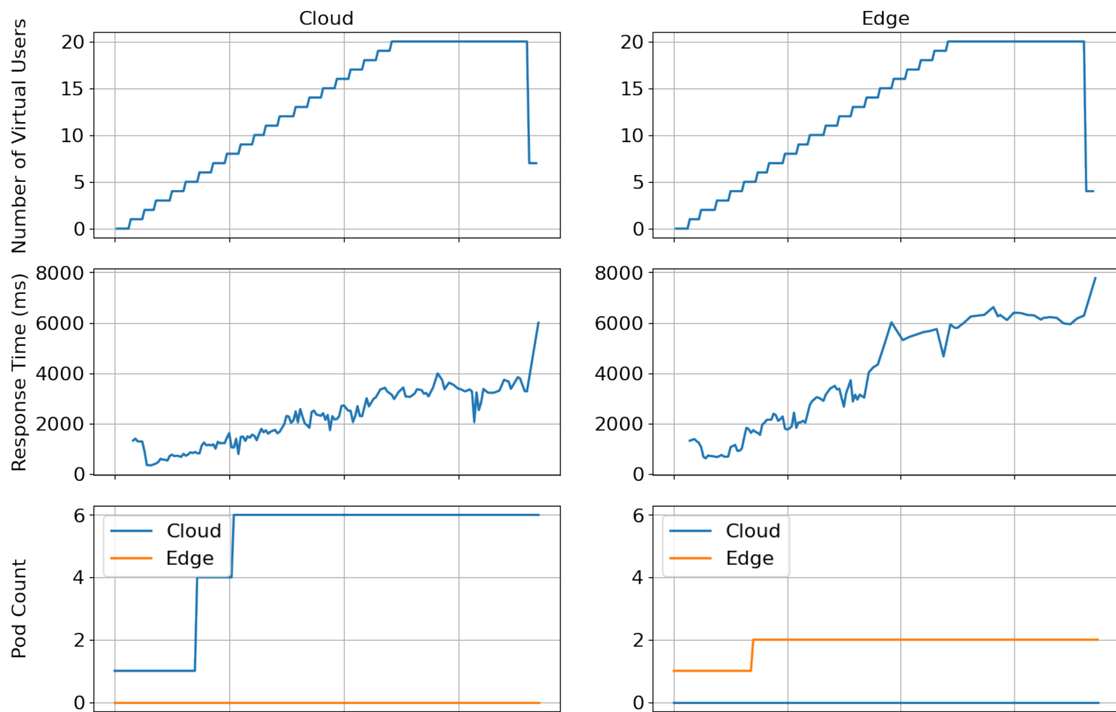


Figure 5.3.: `gzip-compression` Base Scenarios.

in the first third of the evaluation run. After this point, no more pods can be scheduled to cope with the increasing load, thus, request durations increase slowly but steadily from that point onwards.

Comparing to Base Edge, we start out with comparable request durations under 2 seconds. But, as expected due to fewer resources at edge, the rise in request time under increasing load begins earlier and is steeper compared to Base Cloud, up to a maximum of about 6 seconds after approximately the first half of the run. Having a look at the scaling behavior, we can see the correlation, as the edge node maxes out on scheduled pods at a count of 2, even earlier than in the Base Cloud Scenario.

For both Base Cloud and Base Edge, the sudden fall in VUs and sudden uptick in request time at the end of the data collection interval can be explained with `k6`'s behavior of cancelling requests at the end of the execution time after a grace period. Due to a longer request round-trip time, this shutdown behavior is also reflected in the collected data.

Figure 5.4 shows the collected data relating to the Scenario E1 from Section 5.2.

Starting with 1 pod at edge, and another one each at edge and cloud kicking in shortly after start, we observe request times comparable to Base Cloud and Edge: starting at under 2 seconds and increasing steadily together with rising request load, up to a maximum of just under 6 seconds. As more and more cloud pods are scheduled to assist with the load, request duration drops and maintains a steady trend of around 1

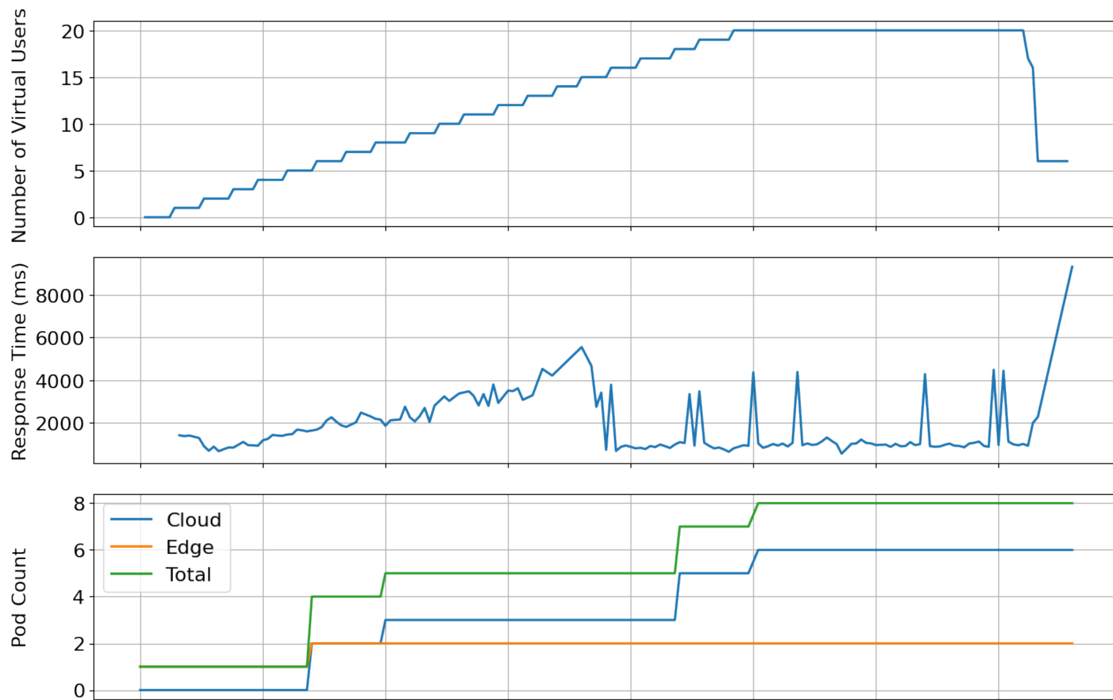


Figure 5.4.: `gzip-compression` Scenario E1.

second about halfway through the evaluation run, disregarding some outlying requests.

The slight discrepancy between more pods being scheduled and request time only decreasing some time after may be explained by the missing readiness probes for pods due to the issue described in Section 4.5.7.

Again, the sudden drop in VUs and sudden uptick in response time at the end of the evaluation window may be explained by `k6`'s shutdown behavior.

5.4.3. `dd` (Disk I/O)

Figure 5.5 shows the results according to the Scenarios Base Cloud and Base Edge from Section 5.2 for the `dd` function, which mainly impacts the host's disk by executing disk I/O operations.

Examining the response times for the Base Cloud scenario, we can observe a steadily rising trend in the first half of the evaluation run, correlating with the climbing request load represented by the number of VUs. Pods have already reached the maximum possible count of 8 shortly after the first quarter.

In the third quarter, we notice some jumps of response times between about 5 and 10 seconds, potentially related to routing delays between the two participating cloud nodes. However, in the last quarter, request times drop, with some requests experiencing times under 5 seconds. This may be explained by `k6`'s behavior, where the tool is not generating any new requests but simply waiting for existing requests to finish towards

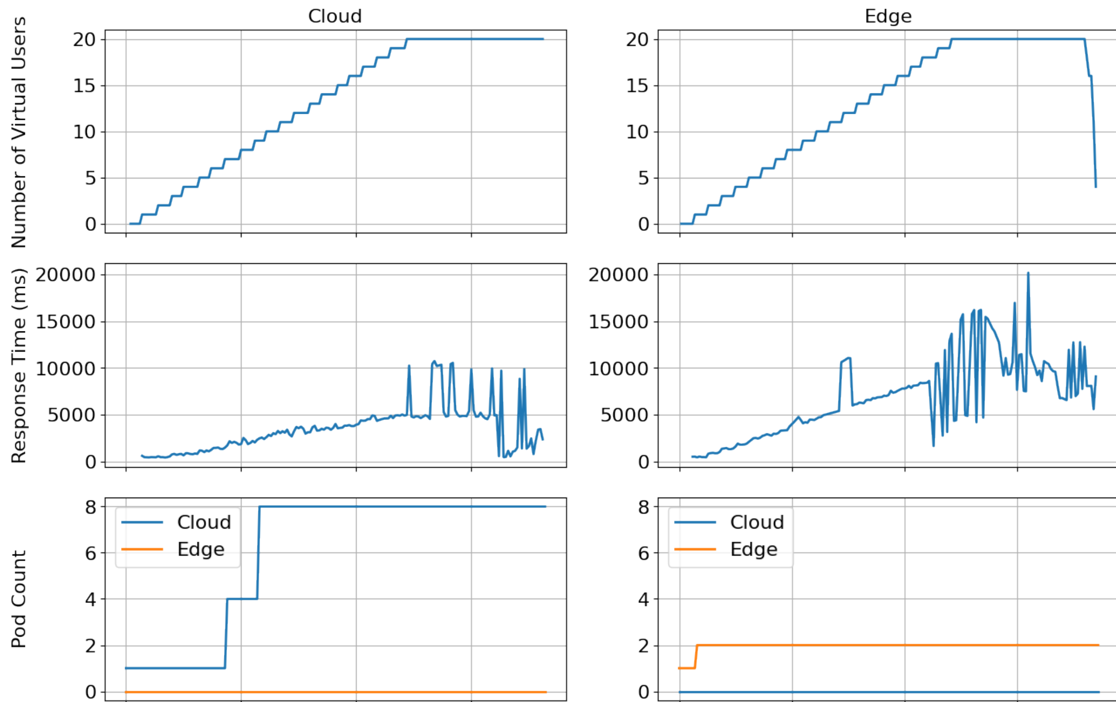


Figure 5.5.: `dd` Base Scenarios.

the end of execution.

Having a look at the results belonging to the Base Edge scenario, we see a similar rise in request time correlating with the rise in VUs in the first half, although steeper up to around 8 seconds. This checks out when bringing in the pod count, which has already maxed out at 2 almost immediately after the start of the evaluation run.

In the third quarter of the run, we can observe some jitter in the response times, jumping between about 3–5 seconds and up to 17 seconds, while experiencing maximum request load by the VUs. The same behavior is observed well into the last quarter, although not quite as severe, with times jumping between about 7 and 13 seconds, disregarding some outlying data points. One possible explanation for this observation may be CPU scheduling on the resource-limited edge node with one CPU core (see also Table 4.1) under heavy load. Depending on which of the pods on the edge node the request is routed to and which pod currently holds CPU access, requests may either complete sooner or later.

Figure 5.6 shows the collected data relating to the Scenario E1 from Section 5.2.

In the first quarter, with only pods scheduled at edge, the request duration trend is similar to the one of the Base Edge scenario, with request times steadily increasing together with load generated by VUs up to around 5 seconds. Afterwards, we observe a peak in request time while more cloud pods are scheduled to assist, probably related to a delay in pod readiness due to the issue described in Section 4.5.7.

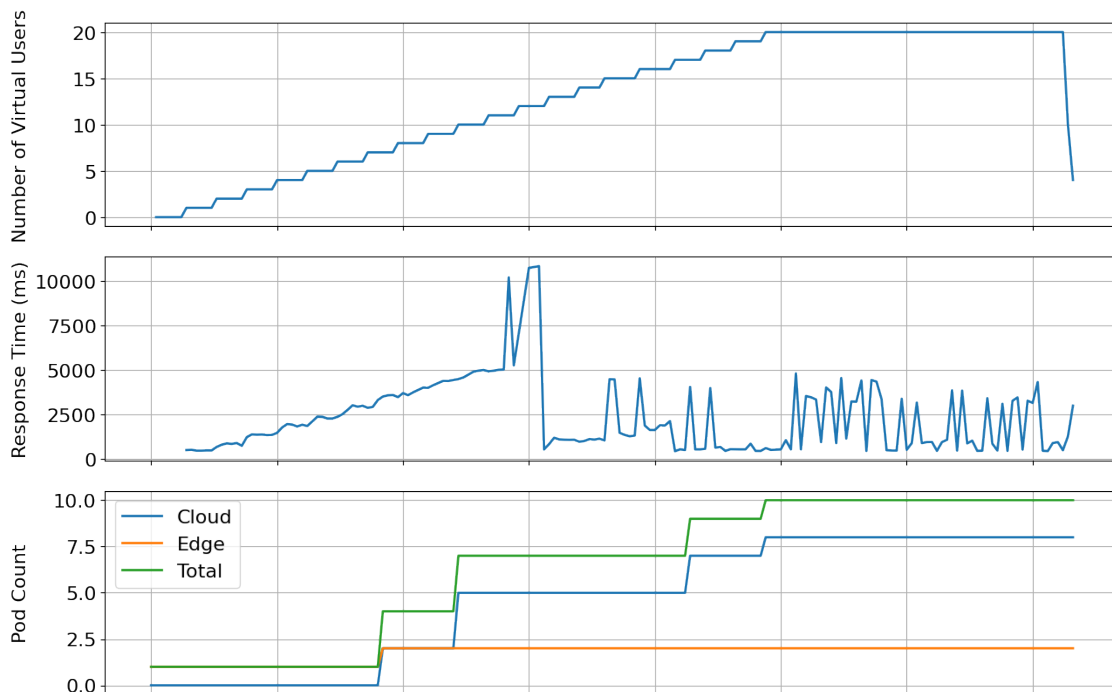


Figure 5.6.: `dd` Scenario E1.

From there on, request times stay consistently under 5 seconds even under maximum load with the maximum possible amount of pods deployed, an improvement to the individual findings of each the Base Cloud and Base Edge scenarios. Some jitter in this range is still observed, likely due to the reason already mentioned in relation to the Base scenarios—the routing of requests between the request entrypoint node and the node hosting the pod actually answering to the request may add a delay.

The sudden drop in VU count at the end of the request run can again be explained by `k6`'s shutdown behavior, as detailed in Section 5.4.2.

5.4.4. `shasum` (Network I/O & CPU)

Figure 5.7 shows the results according to the Scenarios Base Cloud and Base Edge from Section 5.2 for the `shasum` function, which was chosen mainly to target the network and partly the CPU capabilities of nodes.

This function stands out quite a bit compared to the previous functions: as this function is targeted at network capabilities, we have to deal with requests timing out. This is represented by `k6` with a request duration of 0, resulting in the sudden drops noticeable in the data plots. We also had to limit the VU count to 10 instead of the maximum of 20 used for the previous functions in order to collect meaningful data, otherwise too many requests timed out.

Assessing the Base Cloud scenario, we notice there is a lot of jitter in request times

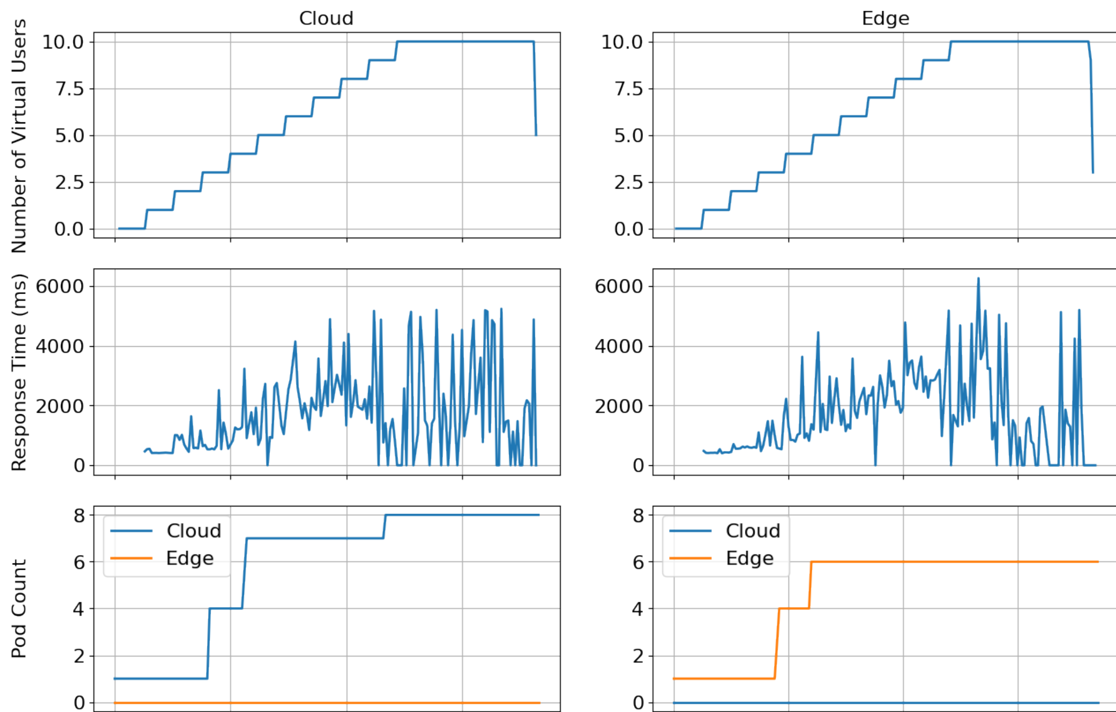


Figure 5.7.: `shasum` Base Scenarios.

throughout the whole evaluation run. However, a trend is still visible. In the first half of the run, request times climb from about 1 second up to about 4 seconds, correlating to the increasing load generated by the rising number of VUs. The pod count almost climbs to the maximum amount possible shortly after the first quarter mark.

In the second half of the data collection window for the Base Cloud scenario, the jitter becomes even more noticeably, with more requests timing out now. Requests that complete successfully however either have a quick or rather long response time, ranging between about 1 second up around 5 seconds, with deployed pods having reached their maximum count of 8.

Having a look at the results of the Base Edge scenario, we notice results similar to Base Cloud, with a similar pod scaling behavior, but instead capped at 6 pods. Due to comparable pod count, request times behave similar, too, climbing from around 1 second to around 4 seconds in the first half of the run, although with some spikes in between.

In the second half, similar to Base Cloud, we observe some more jitter in request times, with more requests timing out, and some resulting in request durations of 5 to 6 seconds under maximum request load by VUs. However, some requests finish quite fast with a duration of about 1 to 2 seconds.

Again, for both Base Cloud and Edge, the sudden drop in VUs at the end of the evaluation window may be explained by `k6`'s graceful shutdown behavior, as detailed in

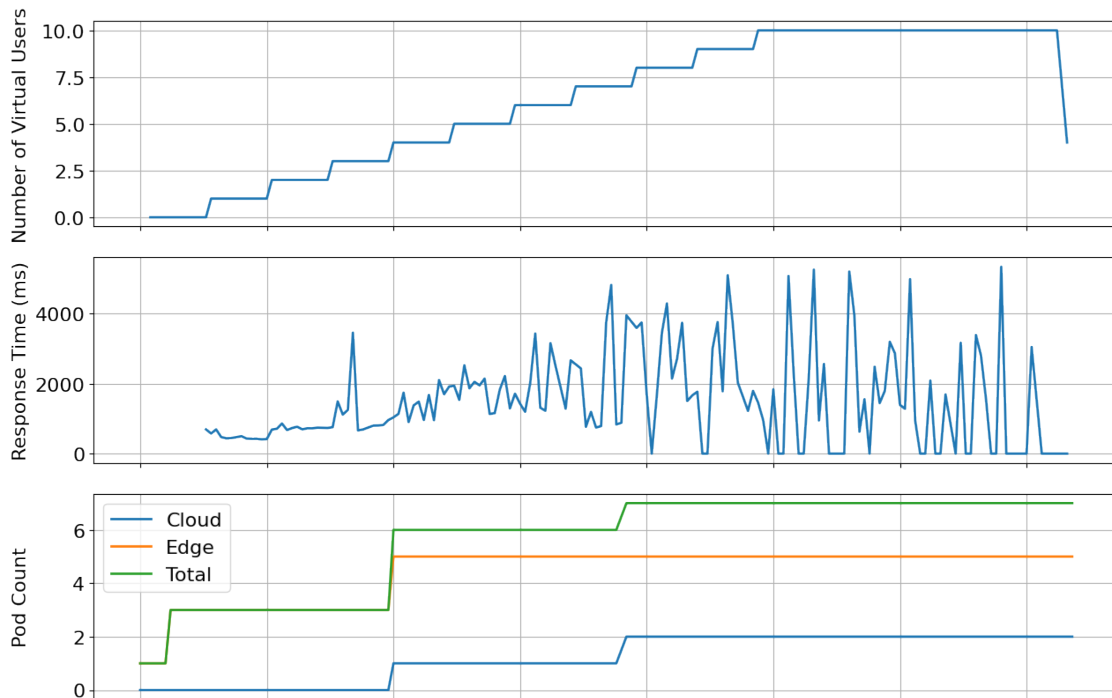


Figure 5.8.: `shasum` Scenario E1.

Section 5.4.2. The drop in request times at the end of the run may as well be explained by `k6` not generating any new requests near the end of execution, waiting for existing requests to complete or time out instead. The same behavior has already been observed in Section 5.4.3 (`dd` Base Cloud).

The similarities of Base Cloud and Edge seem to stem from similar pod counts in both scenarios. In this scenario, we can clearly see the advantage of the specialized KubeEdge architecture of not running all management components on the rather resource-scarce edge node, instead moving some to the cloud, thus freeing up more resources for actual (FaaS) workloads on the nodes.

Figure 5.8 shows the collected data relating to the Scenario E1 from Section 5.2.

We can observe a behavior similar to both the Base Cloud and Base Edge scenarios, with request times climbing from around 1 second to around 4 seconds in the first half of the data collection window, correlating with rising VU count. Scaling behavior is similar too, with most of the load being handled by 5 edge pods after one quarter of the run, with assistance of 1 and 2 cloud pods respectively at the first and second quarter mark, resulting in a total pod count of 7 for the remainder of the evaluation run.

In the second half, we can again see the jitter in request times, with requests either climbing to durations around 5 seconds, completing quite fast with durations around 2 seconds, or timing out. In this scenario, the timeouts seem to happen more frequently when compared to Base Cloud or Base Edge individually. One possible cause may be

the increased amount of nodes involved in Scenario E1, with both cloud and edge nodes participating as opposed to the individual Base Scenarios.

As mentioned earlier, the dropping VU count towards the end caused by `k6` can be observed in the Scenario E1 as well.

6. Conclusion and Future Work

Let us look back and see if our objectives from Section 1.2 motivated by the points from Section 1.1 have been fulfilled.

In this thesis, we have designed an architecture for heterogeneous edge-cloud infrastructures based on K8s and KubeEdge, integrating a FaaS platform, namely OpenFaaS.

We have implemented this architecture successfully, whilst developing a means of deploying the complete setup in a reproducible and extensible manner. Additionally, we have highlighted how to configure deployed workloads with regard to scheduling on both edge and cloud nodes.

As our Evaluation (Chapter 5) proves, we have achieved the Edge-Cloud-Continuum for FaaS workloads mentioned in Section 1.1, which allows for intelligent and transparent scheduling on one K8s cluster composed of both cloud and edge nodes. Furthermore, the collected data suggests that the application of such an approach to the right use-case promises significant improvements regarding metrics such as latency, most noticeable in the results gathered with the `gzip-compression` function (Section 5.4.2).

Applications such as IoT or the closely connected Industry 4.0 may certainly profit from the use of an infrastructure such as the one proposed and implemented in this thesis.

Future Work

Even though the implementation developed in this thesis delivers promising results, future work is always possible to improve and evolve building on our findings.

First, the remaining issues described in Section 4.5 could be ironed out, mainly the issues regarding NodePort routing (Section 4.5.5), the OpenFaaS portal not working for edge-only deployments (Section 4.5.6), and Pod probes (Section 4.5.7).

Another possibility lies in exploring a different K8s-integrated edge computing approach as a basis for integrating edge nodes, such as the technologies *K3s* [20] or *OpenYurt* [28] already mentioned in Chapter 3. This approach could also rule out any issues related to the KubeEdge EdgeMesh component, which proved to be error-prone as described in Section 4.5.3, but may come with other issues or disadvantages which have to be considered.

Furthermore, one could research more reliable monitoring solutions that also integrate well with KubeEdge's special architecture as an alternative to the Kubernetes Metrics Server [26] used in our approach. Instead of looking into other solutions, trying to integrate metrics data collected by the Metrics Server into Prometheus, e.g. by writing

an adapter, would also be feasible to allow for a more professional monitoring solution, also enabling analysis and visualization with tools like Grafana[1].

Further work could also explore if tools such as `kustomize`[2] could be integrated into the FaaS deployment workflow with currently employed tools such as the `faas-cli` [21]. This could assist with the patching of new and existing K8s resource manifests in order to configure them for deployment on edge-cloud infrastructures according to Section 4.4. If this approach does not work out, a custom tool solution may be implemented.

Lastly, as an alternative or addition to the previous point, one could explore automatic patching of all new K8s resources, to automatically and reliably add adjustments such as tolerations or nodeAffinity configurations. One possible approach to achieve this would be looking into creating a custom *Mutating Admission Webhook*[3] to patch resources created in a specific K8s namespace, such as the OpenFaaS-specific `openfaas-fn` namespace.

[1] https://grafana.com
[2] https://kustomize.io
[3] https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/

List of Figures

4.1. Overall Architecture. 12

5.1. `nodeinfo` Base Scenarios. 32
5.2. `nodeinfo` Scenario E1. 33
5.3. `gzip-compression` Base Scenarios. 34
5.4. `gzip-compression` Scenario E1. 35
5.5. `dd` Base Scenarios. 36
5.6. `dd` Scenario E1. 37
5.7. `shasum` Base Scenarios. 38
5.8. `shasum` Scenario E1. 39

List of Tables

4.1. Hardware specifications. 13
4.2. Port Requirements for inter-node communication. 13

5.1. Evaluation Scenarios. 28
5.2. Functions used for Evaluation. 31

List of Code Snippets

4.1. KubeEdge installation with `keadm`. 14
4.2. Tainting edge nodes with `NoSchedule` effect. 14
4.3. OpenFaaS deployment with `arkade`. 15
4.4. OpenFaaS rollout status. 15
4.5. OpenFaaS credential retrieval. 15
4.6. OpenFaaS Gateway node and port retrieval. 16
4.7. OpenFaaS login and function deployment via `faas-cli`. 16
4.8. Deployment configuration for functions at edge. 17
4.9. Cluster creation with KubeKey. 18
4.10. Metrics server deployment manifest. 24

5.1. HPA configuration for deployments. 28
5.2. Deployment configuration for Scenario Base Cloud. 29
5.3. Deployment configuration for Scenario Base Edge. 29
5.4. Deployment configuration for Scenario E1. 30

A.1. KubeKey cluster configuration file. 53

Bibliography

[1] T. Pfandzelter and D. Bermbach. "tinyFaaS: A Lightweight FaaS Platform for Edge Environments". In: *2020 IEEE International Conference on Fog Computing (ICFC)*. Apr. 2020, pp. 17–24. DOI: 10.1109/ICFC49376.2020.00011.

[2] Amazon. *AWS Lambda@Edge*. 2022. URL: https://aws.amazon.com/lambda/edge (visited on 01/15/2022).

[3] Amazon. *Amazon Web Services*. 2022. URL: https://aws.amazon.com (visited on 01/15/2022).

[4] Red Hat. *Ansible*. Version 2.12.2. 2022. URL: https://github.com/ansible/ansible (visited on 03/21/2022).

[5] Google. *Google Cloud Platform*. 2022. URL: https://cloud.google.com (visited on 01/15/2022).

[6] *KubeEdge*. Version 1.10.0. 2022. URL: https://kubeedge.io (visited on 01/15/2022).

[7] *Kubernetes*. Version 1.23.5. 2022. URL: https://kubernetes.io (visited on 01/15/2022).

[8] *KubeSphere*. Version 3.2.1. 2022. URL: https://kubesphere.io (visited on 01/15/2022).

[9] Microsoft. *Microsoft Azure*. 2022. URL: https://azure.microsoft.com (visited on 01/15/2022).

[10] *OpenFaaS*. 2022. URL: https://www.openfaas.com (visited on 01/15/2022).

[11] *OpenStack*. 2022. URL: https://www.openstack.org (visited on 01/15/2022).

[12] HashiCorp. *Terraform*. Version 1.1.7. 2022. URL: https://github.com/hashicorp/terraform (visited on 03/21/2022).

[13] Amazon. *AWS Lambda*. 2022. URL: https://aws.amazon.com/lambda (visited on 01/15/2022).

[14] Google. *GCP Functions*. 2022. URL: https://cloud.google.com/functions (visited on 01/15/2022).

[15] Microsoft. *Azure Functions*. 2022. URL: https://azure.microsoft.com/en-us/services/functions/ (visited on 01/15/2022).

[16] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (Jan. 2017), pp. 30–39. ISSN: 1558-0814. DOI: 10.1109/MC.2017.9.

[17] L. Baresi and G. Quattrocchi. "PAPS: A Serverless Platform for Edge Computing Infrastructures". In: *Frontiers in Sustainable Cities* 3 (2021), p. 58. ISSN: 2624-9634. DOI: 10.3389/frsc.2021.690660.

[18] B. Ellerby. *Why Serverless will enable the Edge Computing Revolution*. Apr. 20, 2021. URL: https://medium.com/serverless-transformation/why-serverless-will-enable-the-edge-computing-revolution-4f52f3f8a7b0 (visited on 01/15/2022).

[19] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan. "Osmotic computing: A new paradigm for edge/cloud integration". In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83. DOI: 10.1109/MCC.2016.124.

[20] *K3s*. Version 1.23.3+k3s1. 2022. URL: https://github.com/k3s-io/k3s (visited on 03/30/2022).

[21] OpenFaaS. *faas-cli*. Version 0.14.1. 2022. URL: https://github.com/openfaas/faas-cli (visited on 01/23/2022).

[22] Kubernetes. *Concepts: Nodes*. 2022. URL: https://kubernetes.io/docs/concepts/architecture/nodes/ (visited on 03/11/2022).

[23] Kubernetes. *Concepts: Pods*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/pods/ (visited on 03/11/2022).

[24] Kubernetes. *Concepts: Deployment*. 2022. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ (visited on 03/11/2022).

[25] Kubernetes. *Resource: Horizontal Pod Autoscaler*. 2022. URL: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (visited on 03/29/2022).

[26] Kubernetes. *Metrics Server*. Version 0.6.1. 2022. URL: https://github.com/kubernetes-sigs/metrics-server (visited on 03/30/2022).

[27] Canonical. *MicroK8s*. 2022. URL: https://github.com/canonical/microk8s (visited on 04/04/2022).

[28] *OpenYurt*. 2022. URL: https://github.com/openyurtio/openyurt (visited on 04/04/2022).

[29] Leibniz-Rechenzentrum. *LRZ Compute Cloud*. 2022. URL: https://doku.lrz.de/display/PUBLIC/Compute+Cloud (visited on 01/15/2022).

[30] KubeEdge. *keadm*. Version 1.9.1. 2021. URL: https://github.com/kubeedge/kubeedge/tree/v1.9.1/keadm (visited on 02/07/2022).

[31] KubeEdge. *Keadm Deployment Guide*. URL: https://kubeedge.io/en/docs/setup/keadm/ (visited on 02/07/2022).

[32] OpenFaaS. *Kubernetes Deployment Guide*. URL: https://docs.openfaas.com/deployment/kubernetes/ (visited on 01/22/2022).

[33] alexellis. *arkade*. Version 0.8.10. 2022. URL: https://github.com/alexellis/arkade (visited on 01/22/2022).

[34] M. Eisner. *KubeEdge OpenFaaS Automation*. URL: https://github.com/max-ae/KubeEdge-OpenFaaS-Automation.

[35] Kubernetes. *Concepts: Node Affinity*. 2022. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity (visited on 01/26/2022).

[36] KubeEdge. *KubeKey*. Version 1.2.1. 2021. URL: https://github.com/kubesphere/kubekey (visited on 02/08/2022).

[37] Grafana. *k6*. Version 0.37.0. 2022. URL: https://github.com/grafana/k6 (visited on 03/28/2022).

[38] OpenFaaS. *Store*. URL: https://github.com/openfaas/store (visited on 03/28/2022).

[39] A. Jindal, M. Chadha, S. Benedict, and M. Gerndt. "Estimating the Capacities of Function-as-a-Service Functions". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC '21. Leicester, United Kingdom: Association for Computing Machinery, 2021. ISBN: 9781450391634. DOI: 10.1145/3492323.3495628.

A. Appendix

Code Snippet A.1: KubeKey cluster configuration file.

```yaml
apiVersion: kubekey.kubesphere.io/v1alpha2
kind: Cluster
metadata:
  name: ba-faas
spec:
  hosts:
    - name: ba-cloud-node-1
      address: "<PUBLIC␣IP>"
      internalAddress: "<INTERNAL␣IP>"
      user: "centos"
      privateKeyPath: "<PATH␣TO␣PRIVATE␣SSH␣KEY>"
    - name: ba-cloud-node-2
      address: "<PUBLIC␣IP>"
      internalAddress: "<INTERNAL␣IP>"
      user: "centos"
      privateKeyPath: "<PATH␣TO␣PRIVATE␣SSH␣KEY>"
  roleGroups:
    etcd:
      - ba-cloud-node-1
    master:
      - ba-cloud-node-1
    worker:
      - ba-cloud-node-1
      - ba-cloud-node-2
  kubernetes:
    version: v1.21.5
    clusterName: cluster.local
  network:
    plugin: flannel
    kubePodsCIDR: 10.233.64.0/18
    kubeServiceCIDR: 10.233.0.0/18
    enableMultusCNI: false

---
apiVersion: installer.kubesphere.io/v1alpha1
kind: ClusterConfiguration
metadata:
  name: ks-installer
```

```
  namespace: kubesphere-system
  labels:
    version: v3.2.1
spec:
  kubeedge:
    enabled: true
    cloudCore:
      nodeSelector: { "node-role.kubernetes.io/master": "" }
      tolerations: []
      cloudhubPort: "10000"
      cloudhubQuicPort: "10001"
      cloudhubHttpsPort: "10002"
      cloudstreamPort: "10003"
      tunnelPort: "10004"
      cloudHub:
        advertiseAddress:
          - "<PUBLIC␣IP␣OF␣MAIN␣NODE>"
        nodeLimit: "100"
      service:
        cloudhubNodePort: "30000"
        cloudhubQuicNodePort: "30001"
        cloudhubHttpsNodePort: "30002"
        cloudstreamNodePort: "30003"
        tunnelNodePort: "30004"
    edgeWatcher:
      nodeSelector: { "node-role.kubernetes.io/worker": "" }
      tolerations: []
      edgeWatcherAgent:
        nodeSelector: { "node-role.kubernetes.io/worker": "" }
        tolerations: []
```