MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Smart City

Kubernetes on the Edge

By: Bernd KLAUS, BA

Student Number: 2010303012

Supervisors: Dipl.-Ing. Hubert Kraut
Dipl.-Ing. Andreas Happe

Vienna, April 3, 2022

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Vienna, April 3, 2022 Signature

Kurzfassung

Kubernetes wird als Schweizer Armemesser der Container-Orchestrierung bezeichnet. Auch im Bereich edge-computing bietet der Dienst eine Vielzahl an unterschiedlichen Werkzeugen und Tools an, welche Teils unterschiedliche Strategien und Ansätze verfolgen. Die Auswahl reicht von einem zentralen Kubernetes-Cluster der verteilte Geräte, sogenannte „Leafs“, steuert bis hin zu vielen einzelnen und verteilten kleinen Clustern an der Edge, welche zentral gesteuert werden. Entscheidend ist es den richtigen Anwendungsfall zu erheben, um sich für die optimale Lösung entscheiden zu können. Ebenfalls spielen sicherheitstechnische Aspekte bei derart komplexen Umgebungen eine wichtige Rolle. Die vorliegende Arbeit gibt Einblicke und Entscheidungsgrundlagen sowie Empfehlungen hinsichtlich der IT-Security. Belegt werden die Angaben durch Implementierung eines Proof-of-Concepts

Schlagworte: Kubernetes, edge-computing, distributed System, Proof-of-Concept

Abstract

Kubernetes is the de facto swiss-army-knife for orchestrating container-platforms. In addition, Kubernetes can also be used for deploying devices as well as applications on top of it on the edge of the network. However, there are different methods for archiving comparable results. On the one hand a possible solution is to build a central instance managing small distributed and independent clusters, on the other hand a centralized cluster with just leafs on the edge may be a better fit. This results in the challenge to find the best solution for the desired environment respectively use-case. The following thesis is making use of "Design Science Research" to give introductions on how to choose the proper architecture for the aimed environment.

Keywords: Kubernetes, edge-computing, geo-distribution, proof-of-concept

Contents

1 Introdutcion

Because of internet-of-things (IoT) Devices becoming more and more common, the number of devices capable of communicating with the world wide web (WWW) increases rapidly. Consequently, also the overall traffic generated as well the amount of data which must be processed increases accordingly. Regarding this development edge-computing is the rising start trying to solve that issues. Thereby data is not processed centrally like in traditional datacenters, but it is tried to handle those data close to the user within several distributed systems. Because of this methodology only really necessary data is transmitted to a central instance for further treatment and those the processing-power as well as the bandwidth necessary for processing required data is reduced significant.

It is expected that the number of IoT devices will continue to grow fast [1] over the coming years. Concomitant edge-computing also will become more important in the future and become an important role in modern information technology (IT) architectures.

To be able to control distributed systems effectively Kubernetes (K8s) is providing a lot of useful tools and functions. Fundamentally there are two different approaches regrading the architecture of how to build an edge-computing environment making use of K8s:

- **Default**: A centralized K8s Cluster controlling many leaf-devices (workers) on the Edge.
- **Distributed**: Small and distributed K8s Clusters running independent on the Edge controlled by a centralized Master-Instance.

Another upcoming approach of solving that issue is making use of the service mesh [2]. This ultimately uses or builds on both of the aforementioned technologies. However, since this thesis concentrates on the two main architectures and their differentiation, the service mesh is not the main focus and just mentioned for the sake of completeness.

1.1 Problem area

Problems arise when trying to find the proper architecture for a specific use-case. There is no clear winner when comparing the above-mentioned different variants. Each of them has their own pros and cons and may decide whether a project is successful or not. It is therefore all the more important to choose the proper architecture right before starting, changing the strategy in retrospect would take a lot of time and effort. However, there is no clear guidance on how to find the proper target environment, at least none which apply in general. Occasionally one finds

recommendations for a very specific use case, however the chance is slim low this findings fit your goals respectively enlighten the decisions. This leads us to the following research question.

1.2 Research question

This paper is going to answer the subsequent research questions:

1. What are the main differences of the in chapter 1 mentioned architectures regarding functionality, scalability, costs and security?
2. Which decision criteria must be defined respectively examined to create a catalog capable of choosing the proper architecture easier for IT managers as well as administrators?
3. Is there a trend in which technology is most likely to be used?

1.3 Goal

The main goal of this thesis is to highlight the pros and cons for each of the architectures defined in the Introduction. The focus will be mainly on the geo-distribution aspect. Although IoT is playing a major role in pushing the development forward, however it is not considered further in the present work. To find the proper architecture, or at least recommendations what could fit best for different desired use-cases, a catalog will be defined. An important part will become the decision criteria catalogue helping people making comprehensible decisions based on scientific research. The main characteristics which are taken into account are scalability, state-of-the-art, handling, costs as well as security.

1.4 Methodology

In the first part of the present work existing literature will be inspected. Related and relevant work will be examined accordingly and linked in the document. Also results will be incorporated to get out the most of it. The goal is to create a list with main criteria necessary for decision-making. As a result, a catalogue of criteria is drawn up in order to be able to decide on the right architecture. The design science research (DSR) method serves as a scientific method and to test the characteristics recorded in the catalogue. This chapter is given the most attention, it is the area where new techniques or architectural decisions are finally verified and the proof is given whether the catalogue works as expected or not. In the latter case, the catalog will be revised to reflect the findings of the last step and re-examined again using DSR.

2 State of the Art

2.1 Technology

The present chapter provides an introduction to the general thematic. The main components and objects of K8s are explained aswell as the layers of edge-computing are highlighted. If anyone is already familiar with the subjects, may you jump over to the section 2.2 "architecture" to read further.

2.1.1 Kubernetes

To promote modern development and be able to implement continuous deployment pipelines cumbersome monolithic applications are divided into many smaller units. Each of these units provides only one function. In order to establish the overall functionality, these units are communicating with each other and thus provide services or make use of other ones. This new method of delivering applications brings many advantages in terms of development but also introduce some new challenges and complexities regarding operation. To simplify the tasks around the management of this architecture, K8s has established itself as the de facto standard [3]. K8s was initially developed by Google and later donated to the opensource community. Over the course of time, a broad community has developed around K8s and a number of additional tools and extensions have emerged as a result. The most promising solutions regrading geo-distribution respectively edge-computing are highlighted in the subsequent section 2.2 "architecture". In order to be able to interpret the results of the use-cases, as well as building the necessary basic understanding, the following functionalities and components of K8s are of relevance.

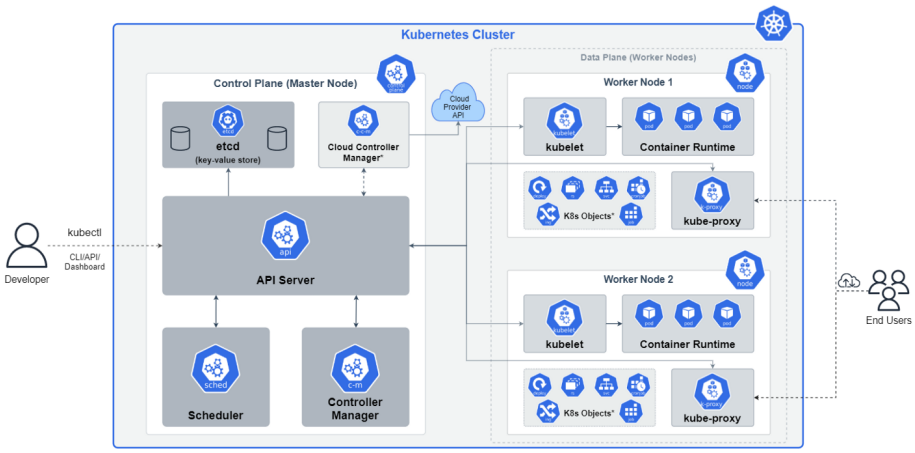

Figure 1: K8s architeture overview [4]

Master Nodes run the so-called *Control Plane* which is responsible for controlling the cluster itself and all the ressources within. The Control Plane consist of the following components [5].

- *kube-apiserver* - acts as frontend web-interface responsible for controlling the K8s cluster as well as the objects inside the cluster. Tools like kubectl abstract the *OpenAPI v2* endpoint and provide access in form of a simply understandable and usable command line interface (CLI).
- *etcd* - represents a high-available and consistent key value store responsible for storing the actual state as well as the desired configuration of the cluster.
- *kube-scheduler* - is responsible for scheduling pods on the available worker nodes. Decision variables such as available ressources, affinity-rules and constraints are taken into account. However, the default *kube-scheduler* ist not aware of any latency between the worker nodes nor the pods communicating with each other. As discovered in the following chapters, this appears to be an important variable for edge-deployments. However, some available white-papers already try to address those issues and show possible solutions by adopting a custom scheduler taking care of those values. More details on this can be found in the chapter 5 "related work".
- *kube-controller-manager* - consists of a single compiled binary controling the status of nodes, jobs, service-accounts and endpoints as well as creating or removing them.
- *cloud-controller-manager* - represents the interface to the underlying cloud-platform. This allows kubernetes to create and/or configure load-balancers, routes and persistent-volumes on the underlying cloud-infrastructure. In a local environment e.g. minikube [6] provide the *cloud-controller-manager* - becomes an optional component and is not required. The same may apply to edge-locations as those areas are outside the cloud most of the time.

Worker Nodes manage the workload, i.e. run the actual application(s). These nodes are composed of the following, see list below, parts [5]. It should be mentioned, that also the described *Master Nodes* are executing those components because some core-componentes are containerized (pods) itself.

- *kubelet* - is an agent which assures that the container is executed properly inside their associated pods according to its specifications defined via *PodSpec*. Also *kubelet* is responsible for monitoring the healthy state of the containers.
- *kube-proxy* - uses the packet filters of the operating system underneath to forward traffic to the desired destination. The resulting access points, also called *Services* in K8s-jargon, can be made available either internally or externally.

- *container runtime* - is the part that finally executes the containers. The default runtime at time of writing is *containerd*, however any runtime is supported that complies with the CRI specification [7].

Kubernetes Objects are persistent properties inside the K8s ecosystem representing the state of a cluster. The most important feature of those objects is to describe the target environment in a declarative way. For this purpose, most of the time, YAML files are used. Kubernetes now ensures that the desired state of the environment is actually achieved and continuously monitors the required objects to meet those defined requirements. This mechanism is also ideal for distributed systems, such as edge computing, as availability can be monitored at any time and a an action can be taken if necessary. Subsequent the main objects are cited starting with the smallest unit [8].

- *Containers* - decouple the actual application and its dependencies from the underlying infrastructure. The main properties of those containers are there immutability and repeatability. This means that the container can be rebuilt at anytime resulting in an identical clone. Likewise, the code of a running container cannot be modified subsequently.

- *Pods* include at least one or more *Containers*. In the most scenarios a single pod consists of a single container, in some cases a so-called sidecar container is used increasing the number of containers inside a pod. Containers which are in the same *Pod* share the same local Socks as well as volumes mounted.

- *Deployments*, *Statefulsets* and *Daemonsets* - are responsible for ensuring the actual workload is provided, to achieve this they control and scale the assigned *Pods*. When creating an application for K8s, it is most likely to create one of those objects. The *Pods* and *Containers* are merely an end product that is derived these objects.

- *Services* - provide an abstract way to make a set of *Pods* available on the network via a single endpoint. Additional deployed pods will automatically be added to the responsible *Services*. Thereby K8s is an excellent choice when it comes to scaling applications without any manual intervention. This also applies for deploying applications to the edge of the network, as illuminated in the course of this thesis. Closely related to the *Services* - is the *Ingress* resource, which is taking care of making the aforementioned objects available outside the cluster. An optional reverse-proxy (*Ingress-Controller*) must be installed in order to make use of the latter.
  A new feature, which is of relevance regarding edge-computing, currently in beta phase, is the so-called *Topology Aware Hint*. Basically its meta-data added to the endpoints defined previously suggesting the connection client on how to reach the destination efficiently (e.g. zones aware of different locations can be defined)

- *ConfigMaps* and *Secrets* - are pieces of information which can be mounted into to *Container* to adjust the configuration inside at runtime. Even whole files can be replaced using

on of them. *Secrets* are only different in the sense that they decode the content, however technically they are the same.

- *Volumes* - provide persistent storage which extends beyond the life cycle of the pods. Volumes can be mounted at any defined position inside the pods. The disadvantage is that the data written to those *Volumes* resides out of the K8s ecosystem and therefore the operator must take care of data security and replication. This becomes even more complicated in an edge-computing environment where nodes have higher latency between them.

2.1.2 Edge-Computing

Edge-computing is the model that extends cloud services to the edge of the network. The computing resources on the edge act as a layer between the user, who provides or wants to process data, and the centralized datacenter (e.g. the cloud). Because data can be processed earlier respective closer to the source, latency and amount of data transferred can be reduced [9]. Also, the required computing-ressources in the datacenter can be minimized because data can be processed at the edge. A major driver of the subsequent s technology is IoT. The amount of devices and resulting data volume, which must be processed, is increasing exponentially [1]. Another technology which depends on it are low-latency applications like e.g. video-streaming.

Hierarchy descripes the layers of the architecture. The following list enumerates the most important layers from top to bottom [9].

1. *Cloud* - centralized datacenter
2. *Fog* - distributed "smaller" datacenters
3. *Edge* - the closest unit to the end-user
4. *IoT* - device at the edge put into use

The main focus of this work is to efficiently combine the two layers *Cloud* and *Edge* and orchestrate between them using K8s. The layer *Fog* is skipped because it is often seen to be "the same" as the *Edge*. Also, current K8s based solutions do not make use of it.

Geo-Distribution characterises the aspect of the geographical propagation of the edge ressources. The goal is to provide computing power over wide areas, each close to the users. By establishing many of these locations in different regions, network latency can be significantly reduced from the user's perspective. However, the latency between the edge-nodes and the centralized cloud still remain.

2.2 Architecture

This chapter focuses on the two different architecture approaches which can be used for edge computing. After an overview the advantages and disadvantages as well as possible solutions are examinated.

2.2.1 Default

In order to be able to manage resources at the edge, a traditional architecture can be used. This is subdivided into a centralized control plane hosted in the cloud and distributed worker nodes near the edge. The same K8s architeture is commonly used when deploying to a single location aswell inside the cloud. The following graphic illustrates the architecture.

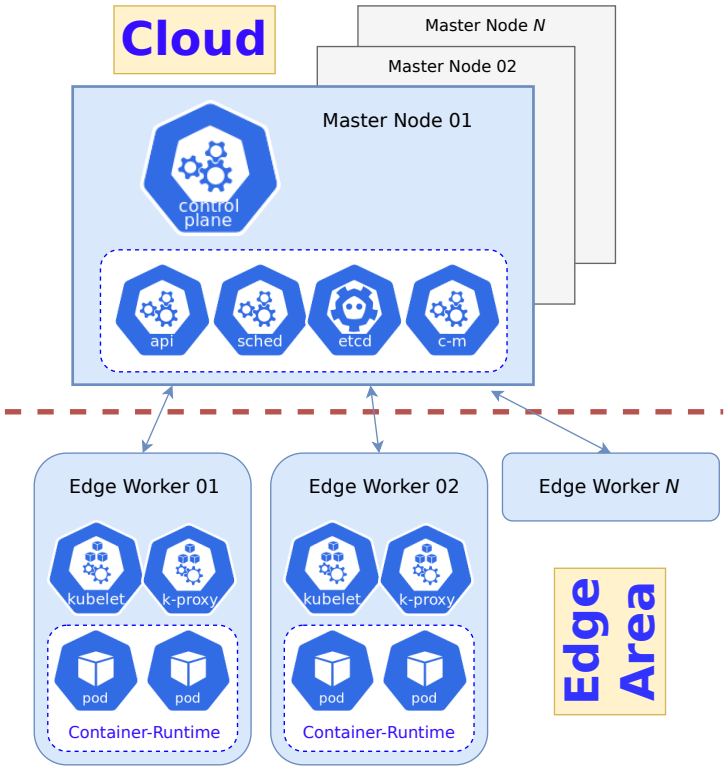

Figure 2: Default K8s architeture

Although the conventional components can also communicate with each other over long distances, there are still some challenges that need to be taken into account. An important role in this context is played by the container network interface (CNI). This part of K8s, which can be selected in the form of a plugin, is responsible for the cluster-internal network traffic. Most of these plugin providers offer advanced features that facilitate geographical distribution. Some of the most used CNI providers are [10]:

- Calico

- Canal
- Cilium
- Flannel
- Weave

Other issues, which must be observed, are the replication of metadata hold by the control planes. Because of the higher latency and-or unstable connection at the edge site data may cannot always be reliably retrieved. To circumvent this problem, asynchronous replication can be used for replicating metadata. Another solution is to make independent, thereby no big data chunks are requested at all. The important part of the metadata store are domain name system (DNS) entries, because K8s heavily relies on them for service discovery. NodeLocal DNS is the recommand way [11] to hold a copy on the worker nodes.

Another challenge regarding storage ist the handling of persistent volumes. Those logical data chunks must be replicated in an asynchronous manner because of the latency between the nodes. As this is not alway possible and data management is a generel challenge, making services stateless is the recommand way to work arround this issue. In contrast to this, forwarding data to the cloud is not an issue in most of the times when using suitable message queues. For that purpose, eg. KubeEdge, is making use of message queuing telemetry transport (MQTT).

In addition there is no awarness of where workloads are running and the level of latency between nodes when using vanilla kubernetes. There are some developments in this area, but they have not really caught on yet [12][13][14].

KubeEdge is an open-source Cloud Native Computing Foundation (CNCF) project [15] that already comes with many of these functions respectively requirements pre-charged. Likewise, this tool was explicit developt for edge-computing. Worker nodes can therefore be distributed across the hole globe without any major adjustments. To make this possible, some components were added or exchanged [16].

- *CloudHub* - webSocket endpoint responsible for sending data to the edge-nodes.
- *Edged* - kubelet is replaced with a lightweight custom agent, the EdgeCore. Thereby also devices with very limited ressources, e.g. a Raspberry Pi, can be used for running workloads.
- *EdgeController* - a K8s controller responsible for managing and distributing metadata.
- *EdgeHub* - the counterpart to the Edgecontroller; uses a websocket server for syning updates to and from the cloud.
- *MetaManager* - Message processor, coordinating between edgehub and edged. Can also store information in an SQLite database. In case of connectivity issues those database is used so that services can continue to run continuously.

- *EdgeMesh* - responsible for replacing the K8s default network functionalities such as the kube-proxy, DNS and CNI and making them fit for edge scenarios. Main goal was to achieve high availability, reliability and the agent as lightweight as possible. However, this agent also has some disadvantages, as we will discover in the use-cases later. It should also be mentioned that this agent is not installed per default and is being developed in a separate project.

The following graphic illustrates the interaction of the components.

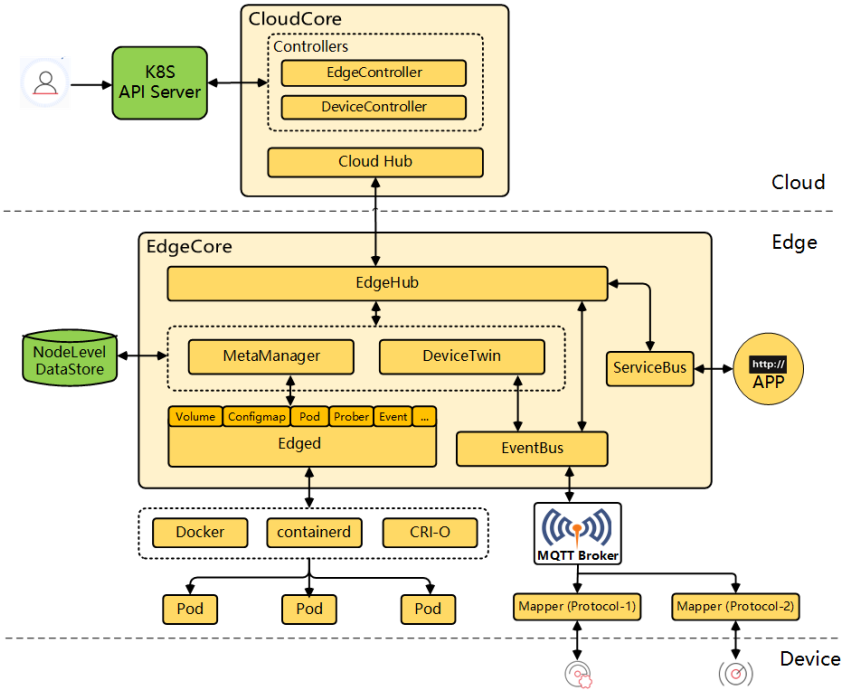

Figure 3: KubeEdge architecture[17]

In addition a component (EventBus) for device management and seemles integration with MQTT is built-in to KubeEdge[15]. However, because this thesis focuses on the geo-distribution aspect, this part is not researched further.

2.2.2 Distributed K8s

In this case, a different approach is chosen and, in contrast to the previous architecture, fully-fledged clusters are also operated at the edge. The obvious advantage is that they can be operated autonomously. There is basically no dependence on the other nodes or there workloads.

However, this also creates new challenges such as the distribution and coordination of workloads. Without additional functionality added, the workload running on the clusters does not know about their neighbours. To overcome this gap, the division "multi-cluster deployment" has excelled in recent years. Even Google, the original developer of K8s, offers such a service [18]

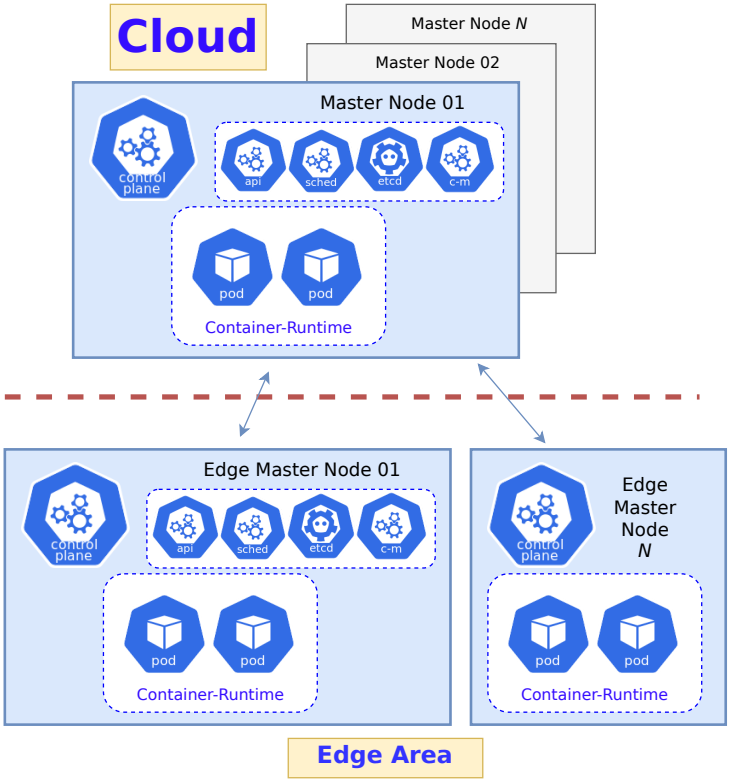

Figure 4: Distributed K8s architeture

in their public cloud. Due to the increased demand, the Kubernetes cluster federation (KubeFed) tool is officially being maintained and further developed by the K8s project[19]. Although the tool is still in beta, it is a very good choice for our field of application. One possible alternative would be the K8s manager by Rancher Labs.

KubeFed consists of a central hosting cluster which controls subordinate clusters via application programming interface (API) delegation. Within KubeFed you can define which configuration should be managed by the hosting cluster. The methodes used are intentionally low-level and can thus be expanded well for different edge-deployment variants. Two different types of information is used to configure KubeFed[19]:

- *Type configuration* - determines which API types are handeld.
- *Cluster configuration* - determines which clusters should be targeted.

The type configuration itself consist of three main parts:

- *Temaplates* - defines a reprenentation of common ressources across the clusters.
- *Placement* - defines which destination-cluster the workload should run on.
- *Overrides* - defines varation of the templates on a per cluster level.

One disadvantage of KubeFed is the lack of network integration. Workloads running on distributed clusters cannot communicate with their neighbours. The integration of the standard CNIs listed above is not possible. However, additional components or configurations can now also solve this problem, as Cilium demonstrates [20]. However, since this paper focuses on the comparison of the two architectures themselves, such extensions are not considered.

2.3 General Challenges

Challenges that appear when creating respectively operating such infrastructures are diverse, as can be seen in the following list [9].

- *Variety* - a lot of different locations, technologies as well as methods on how to control devices on the edge is challenging for both development and operation. The better these different factors can be abstracted and simplified, the more effectively the infrastructure can be used.
- *Integration* - edge-computing evolves very quickly, thereby things could change quickly. The more important it is to keep the provided interfaces extensible. This way, new devices or application can be put in use swiftly.
- *Awareness* - the devices and/or end-user do not care about how their traffic is routed, or their data is processed. However, the architecture needs to take care of that to use the topology in the best possible way.
- *Resources* - scaling Resources like central processing unit (CPU), random access memory (RAM) and disk space at the edge is by far more elaborate than in a datacenter or in the cloud.
- *quality of service (QoS)* - The service provided at the edge should be reliable and provide a good user experience. Availability and performance play a central role in this context. As the availability can not be guaranteed to some extent, an appropriate failover mechanism should be in place.
- *Security* - physical access control as well as isolating applications from each other is a difficult task. Also, the data traffic must be separated accordingly. In general, IT security is a hot topic, and especially at the edge, it requires appropriate consideration for hardening the environment.
- *Monitoring* - another important factor is how to capture metrics and events (logs) from the edge. They need to be indexed on a centralized instance in order to get a general overview on what's happening. Because of the dynamic and rapid changes some kind of automatic discovery should be used for that purpose.

- *Environment* - Some locations may have to deal with difficult conditions regarding their surroundings. Increased dust exposure, poor internet connections or recurring power outages can be some of these factors. The system must be able to cushion or parry such failures accordingly.

The above-mentioned challenges provide a good starting point for defining the necessary tests to find the matching target architecture. Details about this test can be found in the section 3.1 "methodlogy".

3 Design Science Research

"A common topic when performing research in technical disciplines is to design some kind of artefact, such as a model, an information system, or a method for performing a certain task. To address this topic in a systematic and scientific way, Design Science Research (DSR) has stablished itself as an appropriate research method." [21]

3.1 Methodology

According to the description in the introduction before, DSR is used as a scientific method to construct the proof-of-concept (PoC). For this purpose two real world exmaples are build, which are then gradually refined. Those exmaples are descripte in detail in the following section 3.2 "environment". The same are then deployed onto each of these environments. Afterwards, Tests are then defined and carried out on the basis of the issues listed in section section 2.3 "genereal challenges". Based on the results, a preferred environment per property can finally be determined.

3.2 Environment

3.2.1 Generel

The main goal is to create both of the aimed architectures in a similar environment. The target environment should provide good coverage of the diversity encountered in edge environments. For that poropse the PoC is build across Hetzner Cloud and a local site connected via 4G mobile internet.

Coverage This combination allows a wide range of possibilities to be achieved:

- *Geographical distribution* - Hetzner Cloud offers location in Germany, Finland and the U.S for deployments. This allows communication to take place over long distances.
- *Scalability* - On the Hetzner Cloud scaling nodes can be realised quickly via API.
- *network address translation (NAT)* - At the local site, nearby vienna, no public IP is availabe for each of the nodes. Therefore, NAT is used to map a dynamic changing IP to the nodes downstream.
- *advanced RISC machines (ARM)* - Also at the local site, a Raspberry Pi is used as an edge-node to emulate devices with minial ressources.
- *Unstable* - The connection at the local site may be unstable from time to time because the connection is established over the public 4G network using mobile technology.

Locations Subsequent, a tabular lists all of the nodes that are used for each of the both test environments, followed by a graphic showing the locations on a map.

Node	Location	IP	CPU	Cores	RAM	Latency
Master	Nürnberg, DE	dedicated	AMD64	2	4GB	0ms
Edge-1	Ashburn, US	dedicated	AMD64	2	2GB	95ms
Edge-2	Helsinki, FI	dedicated	AMD64	2	2GB	24ms
Raspberry	Kirchberg, AT	dynamic	ARM64	4	4GB	40ms

Table 1: PoC nodes specification

Figure 5: Map of the PoC[22]

OS The operating system (OS) used on all cloud nodes is Ubuntu 20.04.3 LTS. For the local site on the Raspberry Pi the OS Raspberry Pi Lite, without graphical user interface (GUI), is used.

K8s For simplicity, the tool of choice for installing K8s is K3s. This is a lightweight K8s distribution optimized for IoT and deployments at the edge. All necessary features are supported, only the dependencies have been reduced to the essentials. In actual use, insofar as resources in the cloud only play a subordinate role, the standard K8s can also be used analogously. However, at the edge K3s if a perfect match. To illustrate the simplicity, the installation command used is shown below.

```
curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="--disable traefik
    --disable-cloud-controller" sh -s -
```

Code 1: K3s installation

VPN No virtual private network (VPN) is used for either environment. Although a VPN can increase security accordingly and services can communicate more easily, its use in a widely distributed edge environment with many nodes is problematic in most cases. Traditional VPNs are not designed for unstable connections with high latency and possibly rapidly changing internet procotcol (IP) addresses. The tools used in both architetures therefore use alternative approaches via HTTPs and RPC respectively.

3.2.2 KubeEdge

For the default architecture resrespectively KubeEdge variant, only a K8s installation on the master node is required. Edge nodes just need be to equipped with a supported containter runtime of choice. In our specific PoC Docker is used. KubeEdge takes over the roles of the kublete and the kube-proxy as shown in the figure "default K8s architecture". A single binary (keadm) is used to initialise KubeEdge. A token and the public IP are specified as parameters on the master. On the edge nodes, those information is used to join the master.

Challenges The most important steps and challenges encountered during the installation are listed below.

- *CloudStream* - While the standard installation can be done easily, activating logs is much more challenging. Activities such as generating certificates, setting environment vairables and adapting configuration files must be carried out manually.

Installation Steps for installation can be founf in the Github repository: Berndinox/K8sEdge

3.2.3 KubeFed

In contrast to KubeEdge, as also noted in subsection 2.2.2, KubeFed relies on individually acting clusters that are controlled by a central instance. Because of this, each node must be

equipped with a fully functional K8s cluster. Usually, the installation is associated with greater effort, but K3s simplifies this process enormously, as descripted in section 3.2.

Challenges The most important steps and challenges encountered during the installation are listed below.

- *Hosting Cluster* - The challenges arise when it comes to connecting all instances from the central cluster. The kubeconfig for each of them must be modified, to include the puclic ip or full qualified domain name (FQDN) of the node, and transmitted to the central hosting cluster. There, the configuration must be added as an additional context. Finally, a custom binary is used to add each of the defined contexts to KubeFed, as shown below.

```
kubefedctl join edge1 --cluster-context edge1 --host-cluster-context
    default --v=2
```

Code 2: KubeFed join context

- *Dynamic IP* - Because the central hosting cluster is initialising the connection to the nodes at the edge, these nodes must be reachable over a static adress. In most cases the use of static ips is not a problem. However, especially in edge-environments, the use of dynamic ip-addresses may be necessary. To circumvent this problem, a so-called dynamic DNS service is used to establish the connection. Although such a service can be set up relatively quickly, it does mean that an additional component has to be installed and maintained. In contrast, KubeEdge works without such a workaround.
- *dNAT* - As long as the target node is behind a router and connected with a private ip, destination NAT must be configured to forward the necessary port. Alternatively, a reverse-proxy can be used to publish the internal service. This step is also only required for the KubeFed variant.

Installation Steps for installation can be founf in the Github repository: Berndinox/K8sEdge

3.3 Use-Case

As a test, a web-application is distributed over several egde-nodes which is to be made accessible locally as well as via ClusterIP. The goal here is not to develop a complex application, but to uncover the behaviour of the environments. A ready-made nginx container, which is available for both arm and x86, is used as the test pod. The name of the docker image, available in the DockerHub, is: nginxdemos/hello. The K8s objects used are composed of a service definition and a deployment file. The most important parts are listed here:

```
kind: Deployment
spec:
  selector:
    matchLabels:
      app: nginx1
    spec:
      containers:
      - image: nginxdemos/hello
        ports:
        - containerPort: 80
---
kind: Service
spec:
  ports:
  - name: http
    port: 8000
    targetPort: 80
  selector:
    app: nginx1
```

Code 3: Web-application code

It is important to note that the above configuration only contains parts to increase readability and cannot be applied in this form. The full configuration can be found in the Github repository under the folder DOCs.

3.3.1 KubeEdge

Starting the web-application on KubeEdge environment is not as simple as assumed. If the deployment and the service definition are created via kubectl, this looks quite successful at first. The service is created and pods are scheduled accordingly. However, our goal was to make the service-ip accessible from within the nodes of the cluster via ClusterIP. But, when the service is called via curl, an error appears. Another problem is that the pod cannot be accessed either directly on the exposed port or via kubectl exec.

After some research, it turned out that KubeEdge does not use any CNI plugin to offer network-connectivity per default. However, a specially developed Kube-API endpoint is used for this, which is to take over the functionalities and improve them for the requirements of an edge environment. This endpoint does not currently cover all functionalities, which is why our deployment needs to be modified accordingly [23].

The following list represents the available Kube-API verbs:

- Get
- List
- Watch

- Create
- Update
- Patch

Not implemented at time of writing is the verb **"Proxy"**, which prevents us from connecting directly to the pod via kubectl exec. Ohter commands that rely on the proxy verb are attach, top, and logs. For the latter, there is already a solution in the official documentation, also noted in the code base of this thesis.

Three variants are available to solve the problem of missing network connectivity.

- *Without network* - the first approach is to simply omit the network part and make your delpoyment independent of each other. However, this means that the actual added value of the default architecture is lost; the cluster no longer behaves like a location-bound cluster. Also the desccription found in the docs of KubeEdge seem not be be very accurate, when not using additional tools: "With KubeEdge, users can orchestrate apps, manage devices and monitor app and device status on Edge nodes just like a traditional Kubernetes cluster in the Cloud." [17]
- *CNI* - the next logical step seems to use a CNI, which would make all the required functionalities available. However, this means that many of the advantages promised by KubeEdge are gone: The support of private-ip addresses behind NAT, high availability and resilience as well as latency-independencie. In addition, a developer from KubeEdge points out that making use of a CNI is not a recommand solution [24].
- *EdgeMesh* - as described in subsection 2.2.1 is an additional component which has to be installed manually. Although EdgeMesh is part of KubeEdge and integrates seamlessly, it is seen as a separate project whose development is independent. The addon promises to solve many of our problems and the supported functions seem optimal, but one should keep in mind that the project is still very young and only four releases have been issued.

In the course of solving the problem, two variants were examined in detail, firstly the one that does without all additional tools and secondly the variant with EdgeMesh. The variant with post-installed CNI is not considered further; in this case, one could fall back directly on vanilla K8s.

Without network As long as there is no need for complex network communication between the pods, this variant offers a simple solution. Deployments, StatefullSets and DaemonSets can be distributed and scheduled via K8s to the edge-nodes. The services can then be made available for local services only. This is achieved through the configuration of a so-called HostPort. It is important to note that making the port available cannot be done via NodePort, as in this case the port would be made accessible at all nodes, which in turn requires a functioning cluster network. The following is the modified deployment that makes the application accessible

on the edge-location. Please note that the object service was removed entierly, only relevant parts are cited.

```
spec:
  hostNetwork: true
  containers:
  - image: nginxdemos/hello
    ports:
    - containerPort: 80
    - hostPort: 8000
```

Code 4: Web-application HostPort

One possibility to establish the connection between the edge-locations in this variant would be the exchange via MQTT on the application level via the cloud component. However, this increases the complexity tremendous. In this case, one of the other two solutions is to be preferred at the current state of affairs.

EdgeMesh As already noted, this component must be installed before it can be used. Before the actual installation, which takes place within the K8s system, the KubeEdge configuration must be adapted. The installation is divided into the following steps.

1. *Load* - Download the EdgeMesh sources from Github
2. *CRDs* - Create the costum ressource definition (CRD) necessary
3. *KubeEdge* - Konfigure KubeEdg components
4. *Agent* - Apply the YAMLs provided by EdgeMesh
5. *Restart* - and verify your system-changes.

The complete process is available in the following document: Berndinox/K8sEdge. Please note that the project may change rapidly, the installation-process linked above is just a snapshot of the current state. As long as EdgeMesh could be installed without errors, the deployment and service object described above can be distributed and scheduled without any modification. With this solution, the native K8s approach is fully followed and ensured.

What is certain is that the documentation at this point is not sufficiently well formulated. It advertises native K8s support, which cannot be met with a default KubeEdge installation. Although it is pointed out that logging has to be configured separately, the networking section receives little or no attention. It was only through intensive testing and tracking of issues on Github that the above behaviour regarding connectivity could be uncovered. This fact is taken into account in the later evaluation.

3.3.2 KubeFed

In principle, the same considerations must be made for delpoying on KubeFed as for those on KubeEdge without network support. The application must be designed so that no communication can take place between the clusters. However, the native approach of K8s can be followed within the respective clusters at the edge. It would also be possible to operate several nodes at the respective edge-locations in the form of a cluster. Within this cluster, communication between the workloads would be possible without restrictions. Due to these properties, KubeFed is always particularly suitable when fewer and better developed locations are to be connected, which must be able to act independently. In order to roll out the deployment mentioned in the introduction of the chapter to the federated clusters, some adaptations are nevertheless necessary, which are illustrated below.

1. *Enable API-Ressource* - On the master instance, the ressources earmarked for distribution must be enabled. Per-default KubeFed does not distribute any ressources. In this specific case, we have to enable the deployment and serivce object as follows.

```
kubefedctl enable deployments.apps, services --kubefed-namespace
    kube-federation-system
```

Code 5: Enable federation

2. *Create Namespace* - The first step is to create a namespace at the master, which then contains all the resources that are distributed to the clusters at the edge. In our first spefici use-case the namespace ist called "test1".

3. *Create Ressources* - At this point it is crucial to understand that KubeFed distinguishes between two types of objects. The normal objects, which are however only executed and managed on the master, and specially federated objects which will be deployed on the edge-clusters. The subsequent picture illustrates the interaction.

   The conventional objects are simply preceded by the abbreviation "Federation". If the API-resource is also delegated, the objects are passed to the edge-clusters and schedulded there in form of the default objects. The environment at the edge does not even know about the existence of these special federated resources, because they are scheduld in form of a default object. The service and deployment mentioned at the beginning is therefore expanded as follows. The same approach applies for the service object. An important part of the configuration is the placement part, which defines on which edge clusters the respective object is to be created. The complete configuration, can be found in following file saved on: Berndinox/K8sEdge.

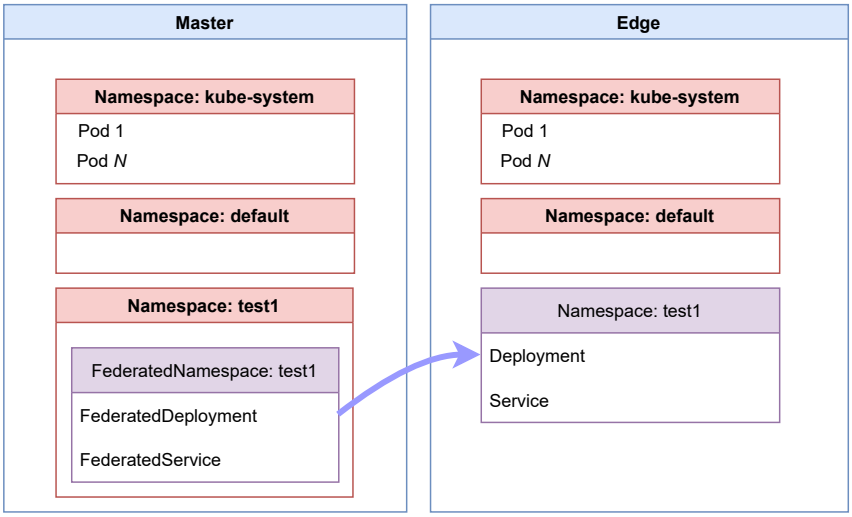

Figure 6: KubeFed objects

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedDeployment
spec:
  ...
placement:
  clusters:
  - name: edge1
  - name: edge2
```

Code 6: Federated Object

3.4 Performed Tests

In addition to the previously described feasibility, the operation of the respective environment is also of great importance. In the following section, tests are therefore carried out on connectivity, resource utilisation and the application itself. If problems arise or environments do not function as expected, adjustments can be made in accordance with DSR guidelines. These adjustments are documented and described accordingly and can be regarded as best-practice.

3.4.1 Ressource usage

As already described in the section 2.3, the efficient use of resources is an important characteristic in the edge-computing environment. Due to the fact that we rely on K3s as K8s distribution for the expression with KubeFed, which is known for using few resources, comparable results can be expected. In principle, however, KubeEdge should be somewhat more lightweight, since only a CNI is used without full K8s distribution. The only unknown is the deployment of EdgeMesh which could increase the resource requirements. The tests are performaned on the edge-side choosing node "edge-1" located in US. The uptime of the nodes

for both environments, at time of the measurements, is exactly the same, approximately 15 minutes.

Idle

The first parameter to be considered is the CPU and Memory load while system is idle or under no significant load. The first test is carried out using the tool cpustat. "Cpustat is a program that dumps the CPU utilization of current running tasks (that is, processes or kernel threads). cpustat is useful to monitor the activity of long lived processes in a system, such as daemons, kernel threads as well as typical user processes" [25].

CPU The deployment created in the usecase is scaled to one pod per node, then the first measurement is performed in idle mode using the command: `cpustat -x -D -a 1 30 -r export.csv`. The utilisation of the CPU by the applications is measured every second for a duration of 30 seconds and exportet in a CSV. To provide a simple overview, the top 5 processes that caused the highest CPU load are shown based on their ticks consumed. "The processor clock coordinates all CPU and memory operations by periodically generating a time reference signal called a clock cycle or tick" [26].

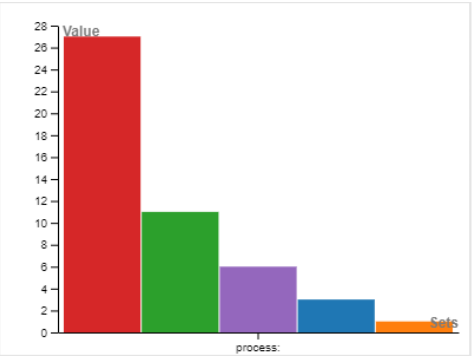

Figure 7: KubeEdge CPU without load

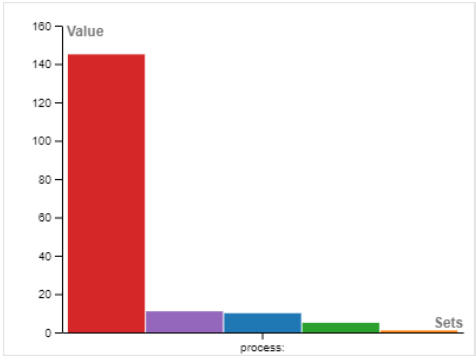

Figure 8: KubeFed CPU without load

In the diagram above, the process of cpustats itself and that of the SSH agent were not taken into account. Neither of them have any direct concern about the test performaed. Basically, the result can be interpreted in such a way that the load is low for both systems and no significant jumps were recognisable. Nevertheless, it must be noted that KubeEdge caused many times fewer CPU compared to KubeFed, while both system idle. The full CSV can be viewed in the designated folder on Github. The link is given at the end of the chapter.

Memory Htop[27] and Ansi HTML adapter (Aha)[28] were used to check the memory utilisation in idle mode. The measurement is carried out under the same conditions as the CPU test described above. The deployment was scaled to 1 on the corresponding node, no increased load must or had to be processed. The uptime of both system is 2 hours 15 minutes at time of

testing. Since the memory behaves less erratically than the CPU, only one sample were taken. The total consumption for the both systems are:

- *KubeEdge* - 239 MB
- *KubeFed* - 600 MB

The following table lists, in descending order, the 5 processes with the most RAM use and their percentage share of the total utilisation.

Process	Usage in %
dockerd	4,5
edgecore	4,2
systemd-journald	3,2
edgemesh-agent	2,6
containerd	2.2

Table 2: KubeEdge RAM uasge

Process	Usage in %
k3s	24,0
containerd	6,9
metrics-server	2,4
systemd-journald	2,2
coredns	2.2

Table 3: KubeFed RAM uasge

As the data shows, KubeEdge gets by with less RAM. KubeFed, on the other hand, uses almost three times as much. Although 600 MB may not seem like much in today's times, this can be important in some scenarios, especially in the context of edge-computing.

Disk The last resource to be examined in this section is that of hard disk usage. The board tool "df" provided by Ubuntu shows the respective space allocated.

- *KubeEdge* - 3.5 GB allocated
- *KubeFed* - 4.1 GB allocated

Similar to the results before, KubeEdge is also more efficient in this section.

Load

In the next part of the tests-series, the behaviour during increased CPU and memory load is examined. Since we are only concentrating on the performance part of individual nodes and its effects in this section, no horizontal pod autoscaler (HPA) is used but only a pod that simulates high load. The behaviour of the overall system, when distributing and scaling application load is examinated in the subsection 3.4.3. The same environment as for the previous tests is assumed as the basis for conducting the test. To test the described scenario, the tool "kube-stresscheck"[29] is used. This tool creates a pod on one of the available edge-nodes which, on the one hand, occupies the entire RAM and, on the other hand, utilises all available CPU cores.

It should be noted that the standard QoS of K8s is applied or maintained for system pods[30]. This mechanism provides three different levels of execution: Guaranteed, Burstable, BestEffort. As the name suggests, pods with the level guaranteed are assigned the highest priority. The test that is carried out is only assigned the level BestEffort and should therefore theoretically not be able to influence system pods that are assigned a higher level. In real operation, high loads can still cause considerable problems under certain circumstances. In productive settings, such tests should be refrained from.

KubeEdge Due to the K8s native support KubeEdge comes with, the configuration provided by kube-stresscheck is ready as is. The version selected is the one that only occupies one node with full load. In order to test the node that was selected as the input for the tests before, the other nodes are temporarily drained. Although the developers of the test application describe it as follows: "Usually pods like kube-proxy, nginx-ingress-controller, calico-node are crashlooping. If kubelet or docker was affected by stress test then node will become NotReady"[29]. However, no such effects could be observed in the test run carried out. The following values were observed and recorded during the stress test on the affected node.

- *Memory usage*: **100%** - a flickering was noticed because the underlying OS is trying to free-up unused allocated memory. On the other hand, the pod fills the RAM directly again.
- *CPU usage*: **100%** throughout on both cpu cores during the test periode.

The following behaviour was observed during the test, which was carried out over a period of one minute.

- *Stability* - All functionalities of the environment were maintained during the test. Also all pods were available during the entire run and did not get re-scheduled or exited. All local services and agents at the edge-node also performed their services unhindered.
- *Logs* - No errors could be read in the local logs of the respective nodes or in the kubernetes-specific logs. No indication of any problems that could be traced back to the load tests could be found within the test pods either.
- *Responsiveness* - The test deployment responds and is fully available during the test period. The response times are somewhat slower, as described in the following subsection 3.4.2.
- *Recovery* - After the end of the test, operation was continue without any consequences. The CPU as well as RAM utilisation go back to the initial values.

In summary, KubeEdge has done exceptionally well. No failures or noteworthy events were detected during the tests. Although the performance of the services that are located on the same node suffers, these effects were within limits and to some extent predictable. The effective use of QoS could further improve overall performance.

KubeFed The test is carried out under the same conditions as the test above. The test deployment is scaled to a pod on the target system, edge-1. However, since KubeFed does not offer an CNI for intra-cluster service communication, the port is changed from ClusterIP to NodePort. This allows the connection test to be carried out using curl with the same conditions. The following values were observed and recorded during the stress test on the affected node.

- *Memory usage*: **100%** - similar to the behaivor with KubeEdge, a flickering was noticed.
- *CPU usage*: **100%** throughout on both cpu cores during the test periode.

The following behaviour was observed during the test, which was carried out over a period of one minute. With the exception of the response times, the same results were achieved. For the sake of completeness, these are listed again below.

- *Stability* - All functionalities of the environment were maintained during the test. Also all pods were available during the entire run and did not get re-scheduled or exited. All local services and agents at the edge-node also performed their services unhindered.
- *Logs* - No errors could be read in the local logs of the respective nodes or in the kubernetes-specific logs. No indication of any problems that could be traced back to the load tests could be found within the test pods either.
- *Responsiveness* - Similar to the test before, the deployment of the test pod was reachable during the test. Details of this test, as mentioned earlier, can be found in the network-section in subsection 3.4.2.
- *Recovery* - After the end of the test, operation was continue without any consequences. The CPU as well as RAM utilisation go back to the initial values.

In summary, KubeFed did also behaive very well during the tests. Even under heavy load, our test service performs well and only stands out due to slightly higher response times. As with KubeEdge, this could be further improved through the use of QoS.

Result and interpretation

Both systems can be considered resource efficient. KubeFed benefits from the use of K3s instead of vanilla K8s. Nevertheless, KubeEdge is the winner in terms of minimum requirements and may be preferable for edge locations with weak devices. Under high load, both environments could prove high stability.

3.4.2 Network connectivity

Reponse time

In the course of the performance tests, which can be found in the subsection 3.4.1, network tests were also carried out to test the behaviour both under load and without load. For this

purpose the tool curl is used which can measure time values via output parameters. The test-command is shown below.

```
curl -w "@curl-output.txt" -o /dev/null -s IP:8000
```

Code 7: Curl command

The curl-output.txt file defines the processing and output specifications. The call via curl is made from the respective master node located in Germany, as also shown in the following Table 1. The target in each case is the node edge-1 which is hosted in the US and performs our target deployment. Due to the differences within the architecture of the two environments, different methods are used at this point. While the internal ClusterIP can be addressed for KubeEdge, which brings the data transfer to the target pod internally via EdgeMesh, a NodePort must be made accessible via the internet for KubeFed. This means that in the case of KubeFed, the curl command must be issued against the public IP of the target node. The service object of our usecase, describe in section 3.3, was therefore modified as follows for KubeFed.

```
type: NodePort
ports:
- name: http
  port: 8000
  targetPort: 80
  NodePort: 30298
```

Code 8: Curl command

It should be noted that the default port-range for NodePorts is from 30000 to 32767.

KubeEdge based on the command declared above, the following data could be obtained.

Measuring point	Time
time_namelookup	0.000059s
time_connect	0.000581s
time_pretransfer	0.000663s
time_starttransfer	0.192938s
time_total	0.193011s

Table 4: KubeEdge: curl no load (ClusterIP)

Measuring point	Time
time_namelookup	0.000070s
time_connect	0.000245s
time_pretransfer	0.000316s
time_starttransfer	0.338320s
time_total	0.338480s

Table 5: KubeEdge: curl with load (ClusterIP)

The total time to complete a call increased by about 75% in the test conducted. While this may sound like a large increase at first, it must be mentioned again that the node was under absolute full load. In this respect, this value is relativised and could even be considered quite good.

In order to provide a complete overview, although functions that constitute a decisive added value were deactivated, KubeEdge was also tested via HostPort. This variant is more comparable with the NodePort described above.

Measuring point	Time
time_namelookup	0.000079s
time_connect	0.093761s
time_pretransfer	0.093832s
time_starttransfer	0.187626s
time_total	0.188056s

Table 6: KubeEdge: curl no load (HostPort)

Measuring point	Time
time_namelookup	0.000033s
time_connect	0.093475s
time_pretransfer	0.093552s
time_starttransfer	0.201756s
time_total	0.201889s

Table 7: KubeEdge: curl with load (HostPort)

The test via NodePort, which bypasses the internal network structure based on EdgeMesh, is similar to the following test of KubeFed. However, this variant also means that a decisive added value of KubeEdge is lost. This shows that EdgeMesh brings with it a certain overhead that should not go unnoticed.

KubeFed under the same conditions, except for the adjustments to the service, the test was run through for KubeFed. The result is displayed below.

Measuring point	Time
time_namelookup	0.000085s
time_connect	0.089800s
time_pretransfer	0.089906s
time_starttransfer	0.180030s
time_total	0.180333s

Table 8: KubeFed: curl without load

Measuring point	Time
time_namelookup	0.000042s
time_connect	0.089541s
time_pretransfer	0.089756s
time_starttransfer	0.225031s
time_total	0.225544s

Table 9: KubeFed: curl with load

The test only shows an increase of 25%. Considering that the target node is under full load, this is an enormously good value and exceeds expectations. Especially under heavy load, Kubefed resp. a single target cluster, can process the request faster compared to KubeEdge and the EdgheMesh-agent.

Connection restoring

In addition to the ability to keep services available under heavy load, seamless restoration of availability is also important in the event of a disconnection. As with the previous runs, the

test-deployment for this setup is scaled to a single pod on the node edge-1. The connection between the master and the node is then disconnected via the firewall function provided by Hetzner. This allows the connection to be cut without manipulating the nodes themselves. In summary, the steps are as listed below in an orderly sequence for each environment.

1. Disconnect "edge-1" for at least 30 minutes with Hetzner firewall.
2. Verify the availability of the pod via NodePort or HostPort
3. Query the status of the node on the master
4. Scaling of the test-deployment on the (unavailable) target node to two pods
5. Restore the connection; disable the firewall rule
6. Verify if the node is marked as available again without any action being taken
7. Check if the deployment has been scaled accordingly

During the entire run, any anomalies in the underlying system and within K8s ecosystem are observed. The time until the system is fully restored and the deployment is scaled is also measured. Likewise, during disconnection, the service should continue to be available through HostPort for KubeEdge or NodePort in case of KubeFed.

KubeEdge Actually, KubeEdge should be able to handle network interruptions excellently, following the description of the developers. Unfortunately, this could **not** be proven in the test carried out. The first four steps of the above procedure were completed without any problems. The affected node is marked as NotReady and the pod that is exposed via HostPort is still accessible. However, the edge-node could not reconnect to the cluster after 30 minutes off-time. In the course of the problem, all error messages in the logs were searched through, leading to the following findings.

First of all, the EdgeMesh-Agent, runnin inside a pod could establish the connection. This source of error can therefore be excluded. Then the EdgeCore component was examined, where the following error stood out.

```
Apr 02 17:37:20 edge-1 edgecore[583]: W0402 17:37:20.670742     583
    context_channel.go:180] The module websocket message channel is full,
    message: {Header:{ID:183c3a0f-3c6f-40f7-ade9-2ec0b9f0e8dd ParentID:
    Timestamp:1648921040658 ResourceVersion: Sync:true MessageType:}
    Router:{Source:>
```

Code 9: EdgeCore Error

This error occurs because the MetaManager still wants to transmit the status of the node and the pods. The messages are accepted by the local EdgeHub process, but cannot then be forwarded to the Cloudcore component because of the connection beeing cut off. Instead the messages are saved in a message queue, so that the messages can be transmitted after

restoration of the connection. While it seems like a good idea in principle to cache local status and events so that they can be processed later, the implementation is obviously failing.

However, the problems continue to accumulate as all messages are passed in one go to the CloudCore queue, after reconnecting the node, which also overruns it. In this status, not only one node is now affected, but all of them, as a status query illustrates.

```
kubectl get nodes
NAME              STATUS                        ROLES                  AGE
edge-2            NotReady,SchedulingDisabled   agent,edge             3d5h
edge-1            NotReady                      agent,edge             3d5h
raspberrypi       NotReady,SchedulingDisabled   agent,edge             3d5h
kubeedge-master   Ready                         control-plane,master   3d5h
```

Code 10: EdgeCore nodes unavailable

The functionality could not be restored without manual intervention. Both the component on node edge-1 and on the master had to be restarted manually. It must be mentioned that at least the distributed pods were recognised without any further action and our service was available during the hole test. Furthermore, the problem only occurs after longer network disconnections, because the queue has to fill up first. However, depending on the environment and workload, this period can change considerably.

In another test, with only 5 minutes of disconnection, the problem could not be provoked.

Researching the issue described above leads us to the following issue #3576 which has already been reported in the Github repo of kubeEdge. Unfortunately, it must be noted that the problem has been open for over two months without any feedback. Considering the challenges from section 2.3, that connections at the edge are often of poor quality and terminations have to be compensated accordingly, KubeEdge does not seem very suitable for such scenarios. The full logs and commands run through during the test could be found in the thesis repo.

In addition, the pods of the deployment could not be removed cleanly after the test. This problem has already been reported as Issue 3736. At least the KubeEdge team is already working on fixing this issue.

KubeFed Due to the nature of the architecture, no such problems were detected during the test on the KubeFed environment. In simple terms, a kubectl hosted in the central cluster controls other clusters. Each cluster functions completely independently of the others and can therefore remain separated for hours without any problems. This is also confirmed by tests that have been carried out. After reconnecting, the disconnected instance was recognised almost instantly and was again controllable from the central unit. It must also be mentioned that the separated cluster remains locally operable during the time of separation. This remains even if the connection is no longer established at all. In this case, one could simply continue to use the

K8s cluster as a singular instance.

Result and interpretation

Fulfilling expectations, KubeFed is characterised by particularly good error resistance in the network area. Both under load and complete disconnection, services remain well accessible. KubeEdge also offers good response times under increased load, although those over the internal CNI has somewhat increased latency. However, this is due to the architecture itself and is within acceptable limits. However, the fact that the local message queue overflows if the connection is interrupted is problematic and, as a result, also causes the master component to crash. Although the functionality could be restored by restarting the services, a larger environment with many edge-nodes cannot be operated sensibly in this form. However, it can be assumed that a fix will be provided for this.

3.4.3 Application stability

This part of the test focuses on the scalability of the environment and the stability achieved in the process. For this our test deployment from the previous chapters is reused. The first step is to scale from 0 to 100 pods, then all events of the namespace and start-times are evaluated and compared. Any system-relevant peculiarities or problems are also examined and recorded. All the steps performed during the test for both environments are listed below.

1. Scale test-deployment to zero pods
2. Scale up the test-deployment to 100 pods
    - KubeFed: the pods are evenly distributed over the nodes
    - KubeEdge: the decision is left to the Scheduler
3. Recording of the following events
    - Namespace events
    - Pod status and startup times
    - Cluster events and behaviour
4. Scale test-deployment down to zero pods

The command to monitor the state of the pods is as easy as:

```
watch -n 2 'echo $(echo -n $(kubectl get pods -n=test1 | grep "Pending" | wc -l)
    && echo -n "/" && echo $( kubectl get pods -n=test1 | grep "Running" | wc
    -l)) | ts  >> running.log'
```

Code 11: Pods status

The code displayed above queries every two seconds the count of the pods in pending and running state and is writing those information as well as an added timestamp to the file called "running.log". It should also be noted that the image (nginxdemo/hello) is already cached on all nodes, and the same image is used for each pod.

KubeEdge After just under one and a half minutes, KubeEdge starts all 100 pods and completes the task flawlessly. The following command confirms that the deployment scaled properly.

```
kubectl get deployment -n=test1
NAME      READY     UP-TO-DATE   AVAILABLE   AGE
nginx1    100/100   100          100         3d23h
```

Code 12: KubeEdge scaling to 100

Besides the evidence that the procedure has worked, however, efficiency is also an important variable. Therefore, as previously described, the pods were monitored during the test and the state of each was recorded with a timestamp. The evaluated data can be presented graphically as follows.

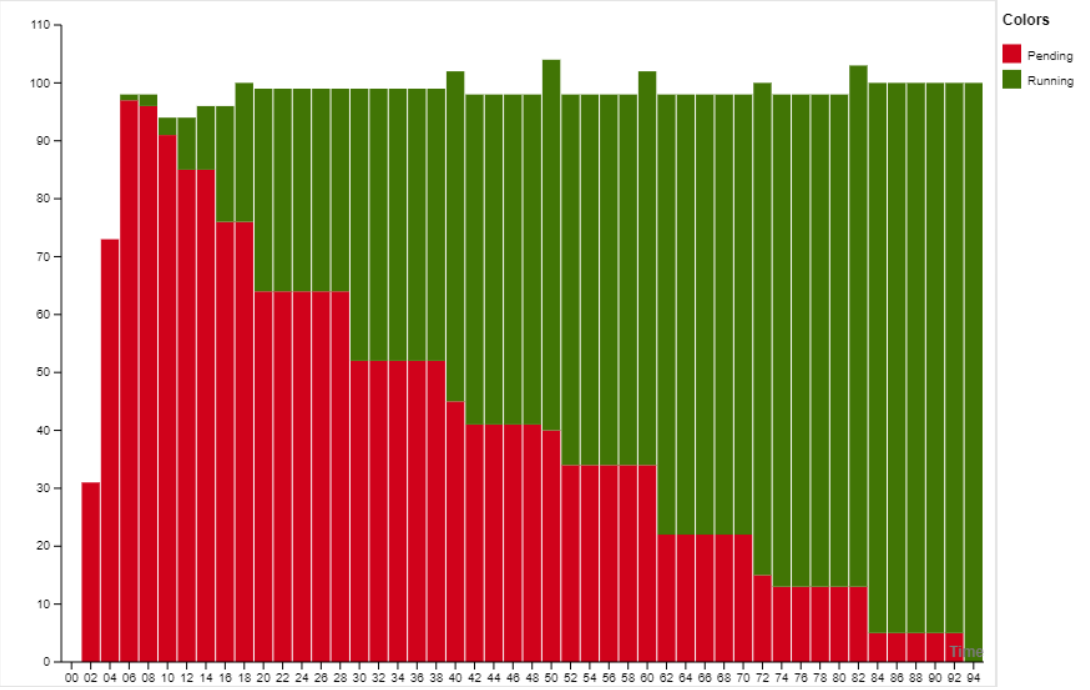

Figure 9: KubeEdge scale

It only takes 8 seconds for all pods to reach the "pending" state, the pods are then started linearly in tranches and are all in the "running" status after exactly 94 seconds. While these values are considered quite positive, considering that the test environment only uses low-resource

nodes, the results regarding distribution are less convincing. The following table illustrates the distribution of the pods across the nodes on the edge.

Node	Pods
Edge-1	6
Edge-2	82
Raspberry	13

Table 10: KubeEdge: scale distribution

The exact causes of the uneven distribution could not be determined exactly, a connection with the respective latency of the nodes would be quite conceivable, as Table 1 illustrates. The node with the lowest latency to the master was able to launch significantly more pods in the tests. An actual connection could not be proven.

KubeFed Due to the different mode of operation, the distribution of the pods for KubeFed is done manually in the configuration, which is why a more even distribution is achieved. While this behaviour may be advantageous for the test at hand, the functions of a dynamic scheduler may work more efficiently in environments with rapidly changing environments and a variety of different deployments.

In order to scale to exactly 100 pods for the KubeFed environment, the deployment was adapted as follows.

```
    spec:
      replicas: 33
...
  placement:
    clusters:
    - name: edge1, edg2, edge3 // simplified line
  overrides:
    - clusterName: edge2
      clusterOverrides:
        - path: "/spec/replicas"
          value: 34
```

Code 13: KubeFed scale adaptations

Since an uneven number of edge-nodes is used, one node must be loaded with an additional pod via override. Similar to KubeEdge, the node with the lowest latency was selected. The deployment was then distributed and the following values were determined. In the course of the distribution of the above-mentioned deployment, the following values were collected. For ease of classification, the values are also shown graphically.

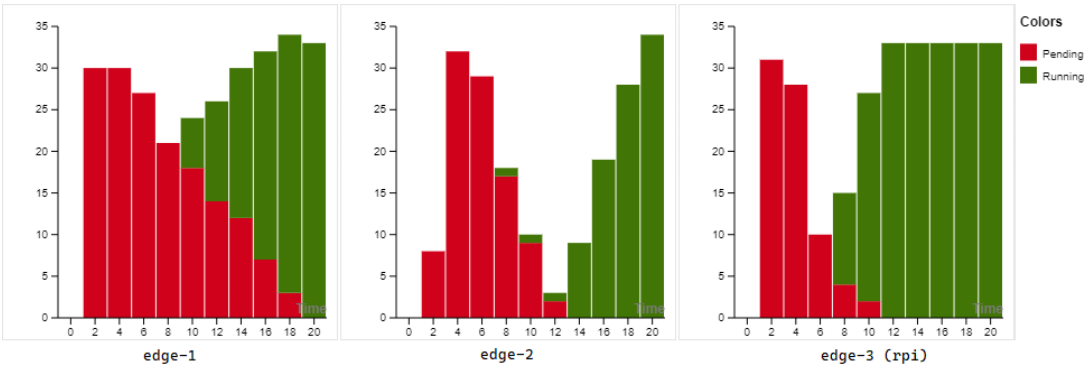

Figure 10: KubeFed scale

After only 20 seconds, all pods could be started successfully. KubeFed thus takes just under a third of the time as KubeEdge. This is also where the performance efficiency of the RaspberryPi node comes into play, requiring only 12 seconds for 33 pods. Nevertheless, it is important to note that KubeEdge started almost all pods on the same node. Assuming linearity and starting all pods in the KubeFed environment on one node, the start time would be about 60 seconds. That would still be faster. The gaps in the above diagram are caused by the "waiting" status, which is not covered by the monitoring command. However, this is only an intermediate status which has no relevance for measuring the total time.

Result and interpretation

Both architectures, KubeEdge and KubeFed were able to scale without major problems. Due to the manually performed symmetrical distribution of the pods on KubeFed nodes, the start-up process is significantly faster overall. KubeEdge, on the other hand, has problems with distribution and starts most pods on the node with the supposedly lowest latency. In the specific case, this leads to a disadvantage of KubeEdge, but in larger environments with increased dynamic and complexity, the use of a dynamic scheduler still brings great advantages. The more detailed study of scheduling is reserved for future work.

3.5 Analysis

3.5.1 Relevant Magnitudes

3.5.2 Outcome

3.5.3 Paraphrase

4 Catalog

4.1 Decision Variables

4.1.1 Installation complexity

The first decision variable that is examined is the effort required for the installation and the associated complexity. According to the KISS principle[31], those solutions with less complexity should be preferred. One investigates the installation routine, descripted in the section 3.2 "environment", very different steps and necessary tools were discovered.

Dependencies The first important variable we look at in this context are the dependencies. This paragraph considers requirements that must be met in order for the installation to proceed successfully.

- *Connectivity* - While KubeEdge works with literaly any connection when combined with EdgeMesh, even behind carrier-grade NAT[32], important conditions must be checked for KubeFed, as descripted in subsection 3.2.3 under the paragraph "challenges". For the sake of completeness, the requirements for KubeFed are listed subsequent.
  - Static IP or dyn DNS agent if dynamic assigned
  - NAT rule if the device is located behind a router
- *Device Support* - Both solutions offer a wide selection of different types of devices supported, like: ARM64, ARMv7, x86, or x64 based systems. However, because KubeFed relies on a full functional K8s cluster those extensive support is provided by the use of K3s and the container-runtime packaged within. In contrast, KubeEdge only requires on one of the supported container-runtimes as well as the KubeEdge binary. Because KubeEdge is taking special care of implementing lightweight agents compatible with container runtime interface (CRI)[33][34], an even wider range of devices can be supported. Due to the design, even edge-devices with fewer resources can be used effectively. In summary, both solutions support a wide range of devices. KubeEdge, however, goes one step further and is therefore slightly in the lead.
- *Other dependencies* - Other requirements such as bandwidth, connection quality and e.g. storage space have little or no influence on the installation itself and are therefore only considered in the following operational part.

Configuration The second step is to examine the installation routine respectively the configuration steps itself.

- *Required Tools* - One important part is the number of tools and binaries used respectivley required to be able to run the environments. The fewer components are used, the less complex the installation and operation. The fewer components used, the greater the tendency to keep the environment simple. Although individual components can also involve quite a high degree of complexity, this variable must not be disregarded. The individual configuration steps are evaluated in the following sections. All tools are listed for each environment subsequent.
  - *KubeEdge*
    - CRI (e.g. Docker)
    - KubeEdge binaries
    - K3s (Master only)
    - EdgeMesh (Master only)
  - *KubeFed*
    - K3s
    - KubeFed binaries (Master only)
    - Helm (Master only)

  It should be noted that K3s includes or builds on CRI.

- *Configuration steps* - This property describes the necessary steps that were required during the installation without additional tools. For KubeEdge the basic installation, without any logging capability and a working CNI, is done by using a single binary. However, if logging is necessary (in almost all cases it should be), some adaptations must be made. The same applies to the installation of EdgeMesh, as described in section 3.3. As the documentation is not clear enough, increased complexity and the associated effort are to be expected. For Kubefed, in contrast, the respective kubeconfig must be transferred to the master and added there as a custom context. This context is then added via kubefed binary. If one only examines the steps that are necessary to operate the two architectures in a meaningful way, these are much easier to accomplish for KubeFed. On the other hand, KubeEdge requires additional configurations and tools to be installed in order to be able to use the full range of functions. If you are looking for a simple and straightforward solution, KubeFed is the winner in this section.

- *Documentation* - Both documentations appear well maintained and kept up to date. The KubeEdge updates seem a little more recent according to the Github history. The KubeFed documentation can only be read in the form of markdown files within Github, but there is a separate web page for KubeEdge. It must be noted, however, that the part concerning network and connectivity is unclear or not sufficiently described. After research, the extension EdgeMesh, which offers its own documentation, could be found. However, the interaction between the components KubeEdge or the Autonomic Kube-API

Endpoint, the MetaManager and EdgMesh is insufficiently described. Overall, therefore, the KubeFed documentation is easier to read and understand.

Summarised assessment The following table presents the above facts in a clear form.

Category	Criteria	Property	**KubeEdge**	**KubeFed**
dependency	connectivity	w/o static IP	✓	✗
dependency	connectivity	w/o dNAT	✓	✗
dependency	device support	extensive support	✓	✓
dependency	device support	ressources efficiency	**high**	**moderate**
configuration	required tools	master/edge	**4/2**	**4/2**
configuration	handling	necessary effort	**high**	**moderate**
configuration	documentation	clarity/actuality	**moderate**	**good**

Table 11: Installation complexitiy overview

4.1.2 Community and support

Another important factor is the support and the community for the respective application. Especially with relatively new and intuitive software, it is important to either have access to the community to exchange ideas and discuss problems, or to have a manufacturer who can provide support if needed. This is even more true if you want to use the environment productively and commercially.

Community Both products offer an active community with regular meetings, an mailing list and Slack channel. The open source approach promotes the formation of these open communities. KubeEdge is somewhat more active in research than KubeFed, but this does not necessarily mean that you will be better with one or the other.

Support At time of writing, we could not find any official company offering commercial support for the products. This is also, as mentioned above, due to the opensource approach. For KubeEdge, however, there is a list of companies that already use the product and support it accordingly. Among them are well-known companies such as Huawei, Orange and ARM. For KubeFed such a list is not disclosed. In compensation, the project is managed by Jeremy Olmsted-Thompson (Google) and Paul Morie (Apple) for this purpose. For both projects, issues can be submitted via Github.

Activity During the investigation of the topic, KubeEdge seems to be a bit more in the hype than KubeFed. This indirectly also leads to getting the necessary support faster. Details on the topicality can be found in the next subsection "future proof".

Summarised assessment The properties revealed above are tabulated below.

Category	Criteria	Property	**KubeEdge**	**KubeFed**
community	communication-channel	Slack	✓	✓
community	communication-channel	mailing-list	✓	✓
community	communication-channel	regular meeting	✓	✓
support	commercial	maintenance	✗	✗
support	non-commercial	Github issues	✓	✓
support	sponsors	official list	✓	✗
activity	hype	movement	**high**	**medium**

Table 12: Community and support overview

4.1.3 Future proof

Another important element when deciding between the two options is how promising each is for the future. Especially, as in our case, when the project is mainly supported and developed by the community. In the following, the background of the projects, the topicality of the software updates as well as the media impact will be examined. The companies that use or support the project also play an important role in this context.

Releases An important factor are the release cycles and the associated update frequency. An overview of all versions and updates can be found in the "Releases" site of each Github project. Both projects offer new versions on a regular basis, but KubeEde has slightly shorter cycles. Since the beginning of the year (2022), KubeEde has released two versions and KubeEdge has released one [35][36]. The difference is only small at this point, both products shine with topicality.

Github Stats Other characteristics that can define the future viability are the current stats from Github. The following table compares the two projects.

It should be noted that the table shown above is only a snapshot. The values may change accordingly. It must also be noted that the higher number of reported problems may also result from more frequent use. Therefore, the correct interpretation is important.

Github Stat	**KubeEdge**	**KubeFed**
Stars	4800	2100
Forks	1300	470
Releases	30	32
Contributers	229	87
Open issues	147	23
Open pull-requests	34	8

Table 13: Github stats overview

Membership KubeEdge is listed as a CNCF project in the incubation phase. CNCF is a central home for the fastest growing projects in the cloud native category. Even K8s itself was part of it. In contrast, KubeFed is supported and further developed by the K8s oriented special interessts group (SIG). It stands out that due to the group membership clear rules concerning the further development have to be observed.

Sponsors As already mentioned in section "community and support", KubeEdge is officially sponsored by some well-known companies. It should be noted, however, that these are mainly Chinese companies.

Independencie A final important characteristic with regard to future-proofing is the extent to which a solution can be used without further development of the respective application. Although this assumption represents a worst-case scenario that seems rather unlikely in the foreseeable future, it should nevertheless be taken into account. KubeEdge uses a completely self-developed solution on the worker nodes. To replace it, all nodes must be reinstalled with the vanilla component of K8s. Besides the effort involved, important functions concerning edge-computing are lost. KubeFed offers a better solution here, due to its different architecture. Each node on the edge is a complete and independent worker. If KubeFed is no longer used, applications continue to run for the time being; no new installation is necessary on the nodes. The control could be taken over by alternative approaches, e.g. via continuous integration / continuous delivery (CI/CD) pipelines. KubeFed thus offers better independence.

Summarised assessment The characteristics relating to future viability are summarised below.

Category	Criteria	Property	**KubeEdge**	**KubeFed**
future-proof	releases	frequency	**very-high**	**high**
future-proof	Github	stars	**5k**	**2k**
future-proof	Github	activities	**high**	**moderate**
future-proof	membership	development programm	**CNCF**	**SIG**
future-proof	sponsors	companies	✓	✗
future-proof	independencie	alternative option	✗	✓

Table 14: future-proof overview

4.2 Criteria catalogue

The following catalogue of criteria can be compiled on the basis of the data obtained in section 4.1. Zangemeister defines the method as follows: "Utility analysis is the analysis of a set of complex alternative actions with the purpose of ordering the elements of this set according to the preferences of the decision maker with respect to a multidimensional goal system"[37]. By awarding points for each sub-area, the target environment with the highest effectiveness can be selected. The multiattribute value function is used for this purpose. The sum of the calculated individual attributes results in the total value per solution.

$$v(a) = \sum_{r=0}^{m} w_r v_r(a_r)$$

w(r) must be greater than 0 and the condition of validity of the value function must be fulfilled.

$$\sum_{r=0}^{m} w_r = 1$$

Provided that the above mathematical principles are followed, individual weightings can be assigned. This achieves a high degree of flexibility of the catalogue.

4.2.1 Table

The following table acts as a criteria catalogue. Based on the previous qualitative research using the DSR method, the values were obtained. Due to the many different areas of application or objectives that are targeted, this table can be adapted according to one's own requirements.However, the values given serve as orientation points or generalised starting values. There is an empty table in the Appendix A that serves as a template so that adjustments can be made easily.

To define your own values, proceed as follows. First, the categories are given the appropriate weighting. The total value of all categories must amount to 100%. The more important a

Category		Criteria		Rating		Weighted rating	
Name	Weight	Criteria	Weight	KubeEdge	KubeFed	KubeEdge	KubeFed
Installation	30%	connectivity	10%	5	1	0,5	0,1
		device support	10%	5	4	0,5	0,4
		required tools	2%	3	3	0,06	0,06
		configuration	2%	2	4	0,04	0,08
		documentation	6%	2	4	0,12	0,24
Commuity	6%	communication channels	3%	5	5	0,15	0,15
		actual hype	3%	4	3	0,12	0,09
Support	10%	commercial maintenance	5%	0	0	0	0
		bug reporting	4%	5	5	0,2	0,2
		sponsor-list	1%	5	0	0,05	0
Future-proof	10%	Github stats	3%	4	3	0,12	0,09
		release cycles	3%	4	3	0,12	0,09
		independencie	3%	1	4	0,03	0,09
		commercial sponsors	1%	5	0	0,05	0

Table 15: Criteria catalog table

category is or is to be included in the evaluation, the higher the percentage value assigned should be. Then the percentage value of a category is divided into the criteria below it. So if the category is rated 20%, two criteria can be assigned 10% each, for example. The next step is to evaluate the respective criteria for the two environments. The pre-filled values above serve as an initial value respectively approximation. A value between 1 (low, bad) and 5 (high, good) is indicated. The weight is then calculated using the following formula.

$$\frac{CriteriaWeight}{100} * Rating$$

The permitted value range for the rating must be taken into account.

$$Rating = \{1, 2, 3, 4, 5\}$$

4.3 Exclusions and Special Cases

TODO: If somethin is really a requirement. How to mark it at the catalog?

5 Related Work

5.1 Kubernetes and the Edge

The paper[15] provides an excellent introduction to the topic and offers the necessary basics. However, it mainly examines the vanilla K8s, which is only slightly suitable for use in edge-computing without additional tools. In the related work chapter, the paper refers to the use of KubeEdge and KubeFed, on which the present thesis is based on. These theses are therefore to be understood as an extension of the cited paper.

5.2 Extend Cloud to Edge with KubeEdge

The whitepaper[16] provides a good insight into the internals of KubeEde. However, some components of KubeEdge have since been revised or even replaced. KubeBus, for example, has been replaced by EdgeMesh. Also, no realworl exmaple is tested on the environment in the paper. Nevertheless, many mechanisms remain unchanged, which is why the paper is still of great relevance.

5.3 Sharpening Kubernetes for the Edge

As shown in this scientific paper, schedulers do not always work as desired. An approach to solving this problem is described in the white paper "Sharpening Kubernetes for the Edge"[12]. The authors describe the function of a scheduler that takes into account the respective latencies between the nodes and can thus make intelligent decisions. This could be a well-functioning extension for K8s at the edge.

5.4 Ultra-Reliable and Low-Latency Computing with K8s

In the research paper "Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes"[38] the approach of the latency aware scheduler is pushed one step further. However, logical regions can be defined and pods can be launched preferentially in these. A re-scheduler has also been implemented, which makes it possible to automatically redistribute deployment when the environment changes in order to achieve optimum results.

6 Results

6.1 Findings

6.2 Conclusio

6.3 Discussion and further research

Combine KubeEdge and KubeFed (distributed hybrid Kubernetes) Dynamic Scheduler (Für KubeFed?; für KubeEdge verbessern!)

Bibliography

[1] Fay Arjomandi, Matt Trifiro, and Jacob Smith. *State of the Edge 2021. A Market and Ecosystem Report for Edge Computing*. Research rep. The Linud Foundation, Aug. 1, 2020. URL: https://stateoftheedge.com/reports/state-of-the-edge-report-2021/.

[2] CNCF. *Service Mesh - from technical selection to best practice*. Ed. by Malphi. Mar. 15, 2018. URL: https://www.cncf.io/wp-content/uploads/2020/08/rfma-servicemesh-cncf.pdf.

[3] Portworx. *Kubernetes Adoption Survey*. Research rep. PureStorage, Mar. 26, 2021. URL: https://www.purestorage.com/content/dam/pdf/en/analyst-reports/ar-portworx-pure-storage-2021-kubernetes-adoption-survey.pdf.

[4] Patel Ashish. *Kubernetes — Architecture Overview*. Aug. 12, 2021. URL: https://bit.ly/3sMQyEE.

[5] The Kubernetes Authors. *Kubernetes Components*. Ed. by Tim Bannister. Oct. 17, 2021. URL: https://kubernetes.io/docs/concepts/overview/components/.

[6] The Kubernetes Authors. *Minikube Start*. Ed. by Tian Yang. Nov. 19, 2021. URL: https://minikube.sigs.k8s.io/docs/start/.

[7] The Kubernetes Authors. *Container Runtime Interface (CRI). a plugin interface which enables kubelet to use a wide variety of container runtimes.* Ed. by Tim Bannister. Jan. 28, 2022. URL: https://github.com/kubernetes/cri-api.

[8] The Kubernetes Authors. *Kubernetes Concepts*. Ed. by Tim Bannister. June 22, 2020. URL: https://kubernetes.io/docs/concepts/.

[9] Al-Dulaimy Auday et al. *Introduction to edge computing*. Sept. 1, 2020. DOI: 10.1049/PBPC033E_ch1.

[10] Mike Mackrory. *The Ultimate Guide To Using Calico, Flannel, Weave and Cilium*. June 7, 2021. URL: https://platform9.com/blog/the-ultimate-guide-to-using-calico-flannel-weave-and-cilium.

[11] The Kubernetes Authors. *Using NodeLocal DNSCache in Kubernetes clusters*. Ed. by Tim Bannister. Jan. 31, 2022. URL: https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/.

[12] Haja David et al. *Sharpening Kubernetes for the Edge*. Research rep. MTA-BME Network SoftwarizationResearch Group, Mar. 17, 2020.

[13] Reincke Karsten and Steiner Lukas. *k8s-edge-scheduler*. Ed. by Reincke Karsten and Steiner Lukas. Mar. 31, 2019. URL: https://github.com/telekom/k8s-edge-scheduler.

[14] Chima Ogbuachi Michael et al. *Context-Aware Kubernetes Scheduler forEdge-native Applications on 5G*. Research rep. SoftCOM, Mar. 1, 2020.

[15] Manaouil Karim and Lebre Adrien. *Kubernetes and the Edge?* Research rep. Inria Rennes, Oct. 22, 2020.

[16] Xiong Ying et al. *Extend Cloud to Edge with KubeEdge*. Tech. rep. Seattle Cloud Lab, Oct. 27, 2018. URL: https://ieeexplore.ieee.org/document/8567693.

[17] KubeEdge. *Why KubeEdge*. Nov. 27, 2020. URL: https://kubeedge.io/en/docs/kubeedge/.

[18] Google Inc. *Multi-Cluster-Dienste*. Ed. by Google Inc. Feb. 15, 2022. URL: https://cloud.google.com/kubernetes-engine/docs/concepts/multi-cluster-services.

[19] The Kubernetes Authors. *Kubernetes Cluster Federation*. Ed. by The Kubernetes Authors. Mar. 10, 2022. URL: https://github.com/kubernetes-sigs/kubefed.

[20] Chandra Gokul. *Kubernetes Multi-Cluster Networking -Cilium Cluster Mesh*. Tech. rep. ITNEXT, Mar. 10, 2020. URL: https://itnext.io/kubernetes-multi-cluster-networking-cilium-cluster-mesh-bca0f5367d84.

[21] Benner Wickner Marian, Kneuper Ralf, and Schlömer Inga. *Leitfaden für die Nutzung von Design Science Research in Abschlussarbeiten*. Research rep. IUBH Internationale Hochschule, Nov. 1, 2020. ISRN: 2512-319X.

[22] *Google Maps*. Mar. 12, 2022. URL: https://maps.google.com.

[23] GsssC. *[Autonomic Kube-API Endpoint] feature Introduction*. May 14, 2020. URL: https://github.com/kubeedge/kubeedge/issues/2760.

[24] GsssC. *Where is the documentation for deploying the cni plugin?* May 14, 2020. URL: https://github.com/kubeedge/kubeedge/issues/1662.

[25] King Colin. *Ubuntu Manpage: cpustat*. Canonical Ltd. URL: http://manpages.ubuntu.com/manpages/bionic/man8/cpustat.8.html.

[26] B. Thompson Robert and Fritchman Thompson Barbara. *PC Hardware in a Nutshell*. June 1, 2002.

[27] *HTOP*. URL: https://htop.dev/.

[28] Matthes Alexander. *Ubuntu Manpage: aha*. Mar. 28, 2017. URL: http://manpages.ubuntu.com/manpages/bionic/man1/aha.1.html.

[29] Giantswarm. *kube-stresscheck. Script to check Kubernetes nodes on stress (CPU/RAM) resistance.* July 7, 2020. URL: https://github.com/giantswarm/kube-stresscheck.

[30] The Kubernetes Authors. *Kubernetes Configure Quality of Service for Pods*. Ed. by The Kubernetes Authors. May 20, 2021. URL: https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/.

[31] Smith Byron. *How to KISS: The Art of Keeping it Simple, Stupid*. Research rep. 723 W Aspen Ave, Ste: X Double Dot LLC, Apr. 23, 2012.

[32] Richter Philipp et al. *A Multi-perspective Analysis ofCarrier-Grade NAT Deployment*. Research rep. TU Berlin, May 1, 2016. URL: https://www.researchgate.net/publication/303330116_A_Multi-perspective_Analysis_of_Carrier-Grade_NAT_Deployment.

[33] The Kubernetes Authors. *Container Runtime Interface (CRI)*. Ed. by The Kubernetes Authors. Jan. 10, 2022. URL: https://kubernetes.io/docs/concepts/architecture/cri/.

[34] Gpinaik. *CRI Support in edged*. Ed. by Dai Looong et al. Sept. 2, 2021. URL: https://github.com/kubeedge/kubeedge/blob/master/docs/proposals/cri.md.

[35] Github. *KubeEdge Releases Github*. Mar. 27, 2022. URL: https://github.com/kubeedge/kubeedge/releases.

[36] Github. *KubeFed Releases Github*. Mar. 27, 2022. URL: https://github.com/kubernetes-sigs/kubefed/releases.

[37] Christof Zangemeister. *Nutzwertanalyse in der Systemtechnik*. Tech. rep. TU Berlin, 1976.

[38] Toka Laszl ´ o. *Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes*. Research rep. Journal of Grid Computing, Nov. 3, 2020. DOI: 10.1007/s10723-021-09573-z.

List of Figures

List of Tables

List of Code

List of Abbreviations

IT information technology

WWW world wide web

K8s Kubernetes

IoT internet-of-things

DSR design science research

CLI command line interface

QoS quality of service

CPU central processing unit

RAM random access memory

CNI container network interface

IP internet procotcol

NAT network address translation

DNS domain name system

MQTT message queuing telemetry transport

CNCF Cloud Native Computing Foundation

KubeFed Kubernetes cluster federation

API application programming interface

PoC proof-of-concept

ARM advanced RISC machines

OS operating system

GUI graphical user interface

VPN virtual private network

FQDN full qualified domain name

CRI container runtime interface

SIG special interessts group

CI/CD continuous integration / continuous delivery

CRD costum ressource definition

HPA horizontal pod autoscaler

Aha Ansi HTML adapter

A Appendix