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, March 30, 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, March 30, 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 **IoT!** (**IoT!**) Devices becoming more and more common, the number of devices capable of communicating with the **WWW!** (**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 [**SotE21**] over the coming years. Concomitant edge-computing also will become more important in the future and become an important role in modern **IT!** (**IT!**) architectures.

To be able to control distributed systems effectively **K8s!** (**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 [**servicemesh**]. 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 **??** 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 **DSR!** (**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 **??** "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 [**k8ssurv**]. **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 **??** "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.

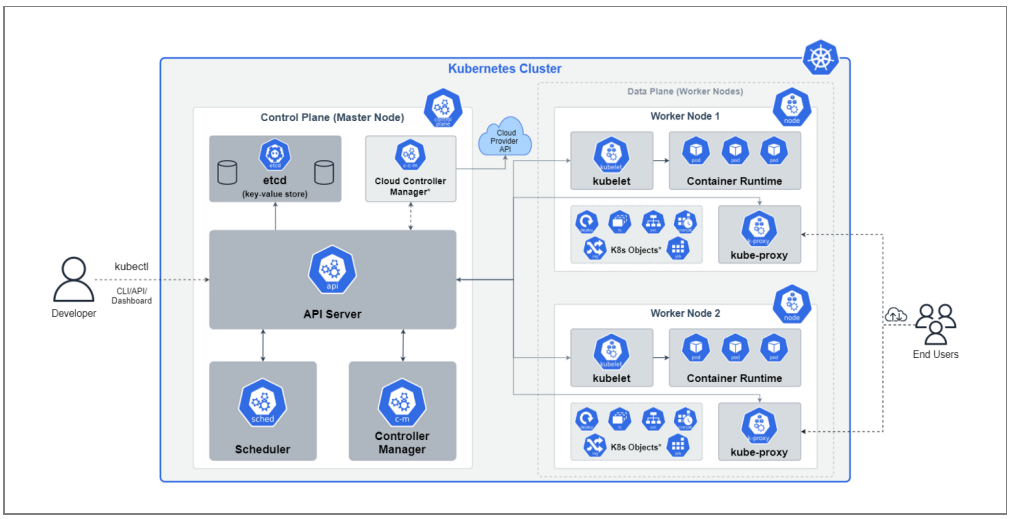

Figure 1: **K8s!** architeture overview [**pic-k8s-overview**]

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 [**k8scomp**].

- *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 **CLI!** (**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 **??** "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 [**minikube**] 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 [**k8scomp**]. 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 [**cri-runtime**].

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 [**k8sconc**].

- *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 [**intro-edge**]. 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 [**SotE21**]. 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 [**intro-edge**].

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.

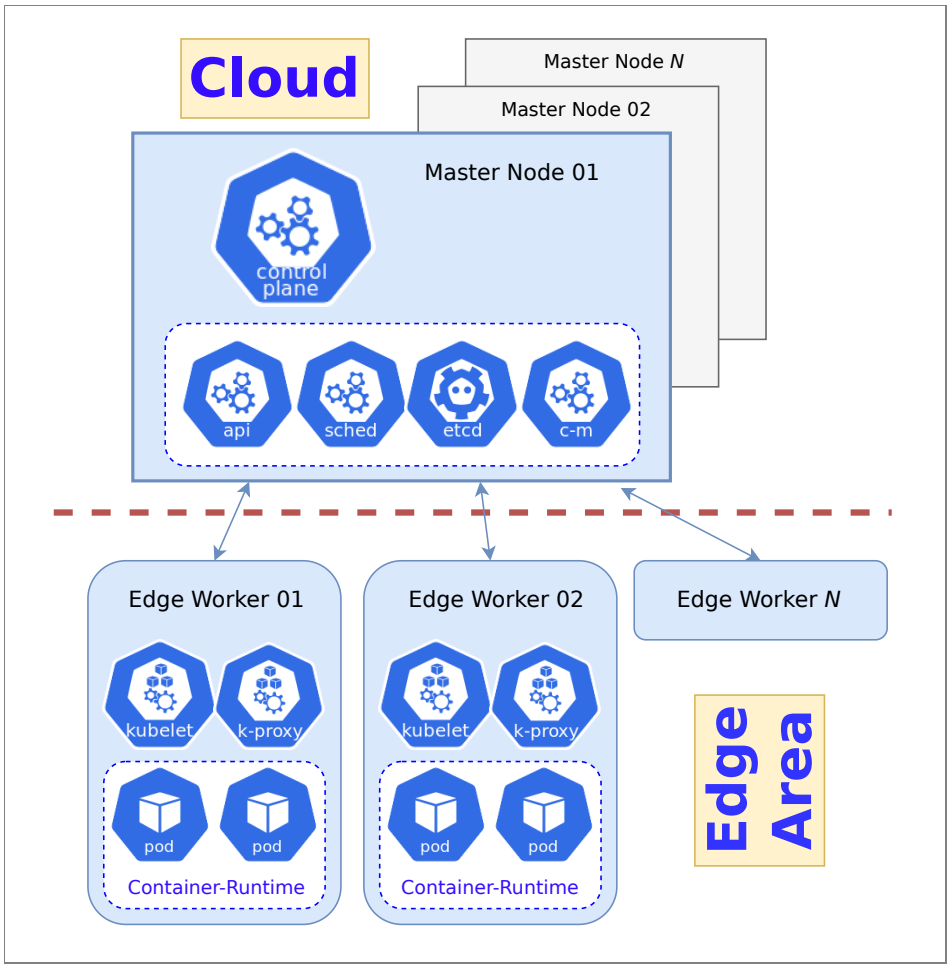

Figure 2: Default **K8s!** architeture

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 **CNI!** (**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 [**k8s-cni**]:

- 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 **DNS!** (**DNS!**) entries, because **K8s!** heavily relies on them for service discovery. NodeLocal DNS is the recommand way [**k8sdnslocal**] 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 **MQTT!** (**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 [**k8s-sharping-edge**][**tk-k8s-edge-scheduler**][**5g-k8s-scheduler**].

KubeEdge is an open-source **CNCF!** (**CNCF!**) project [**hal-kubeedge**] 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 [**kubedge**].

- *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.

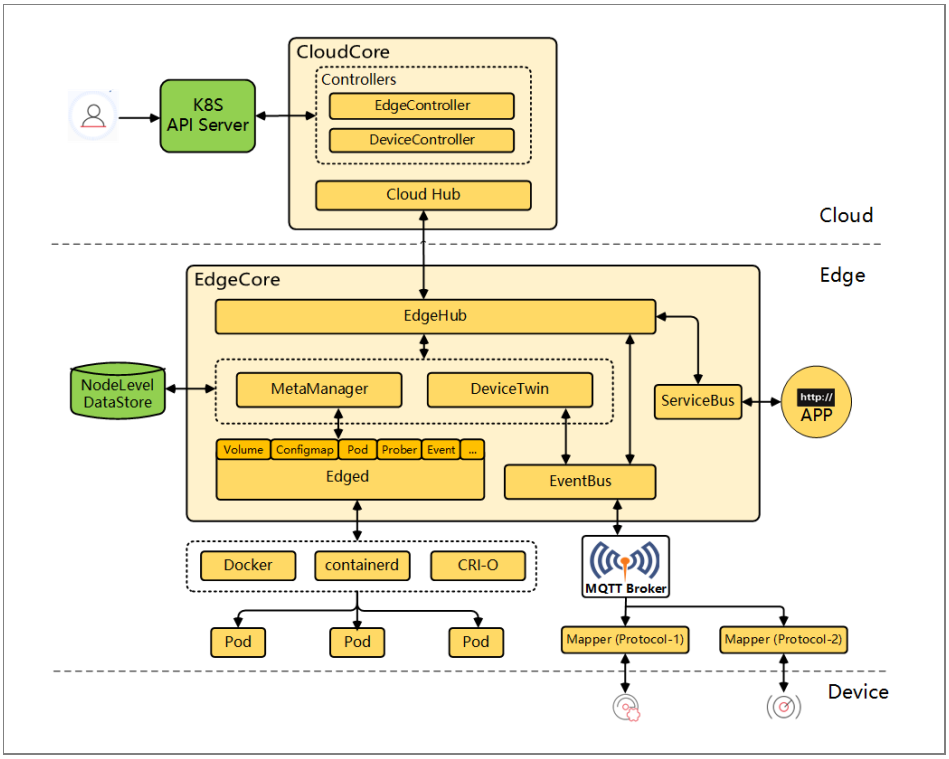

Figure 3: KubeEdge architecture[**ke-docs-why**]

In addition a component (EventBus) for device management and seemles integration with **MQTT!** is built-in to KubeEdge[**hal-kubeedge**]. 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.

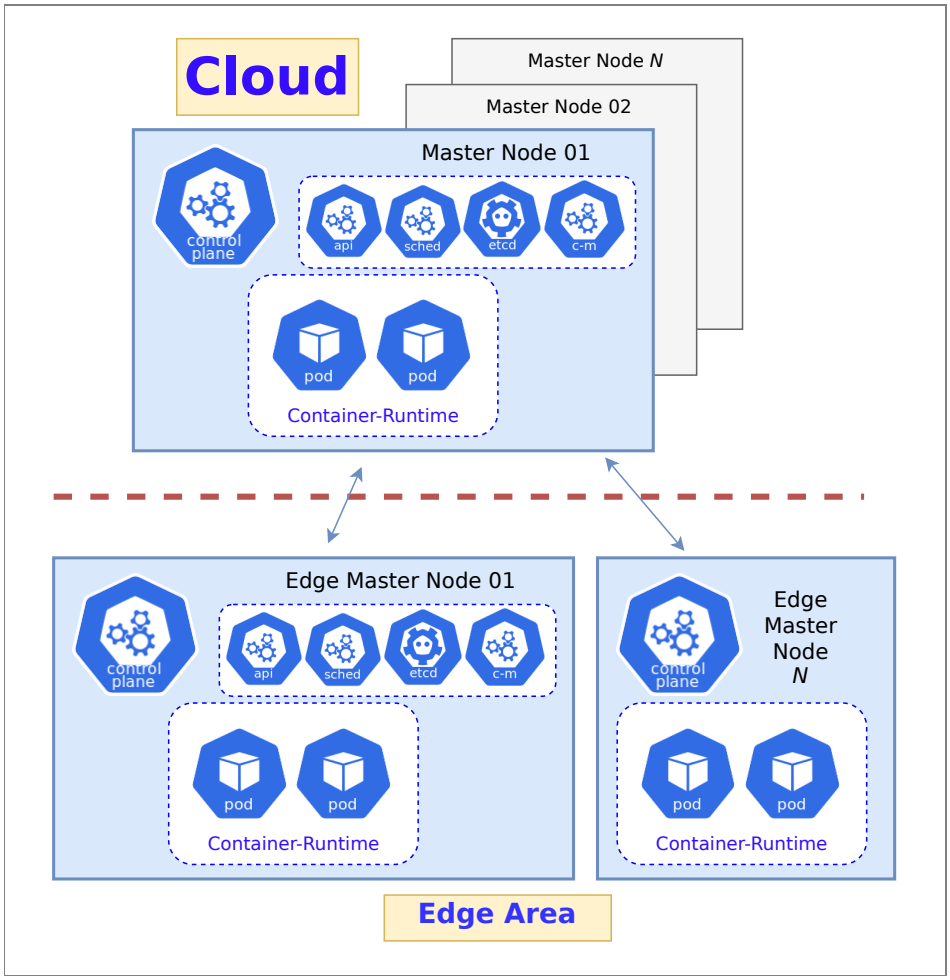

Figure 4: Distributed **K8s!** architeture

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 [**google-mcs**] in their public cloud. Due to the increased demand, the **KubeFed!** (**KubeFed!**) tool is officially being maintained and further developed by the **K8s!** project[**kubefed-github**]. 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 **API!** (**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!**[**kubefed-github**]:

- *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 [**ciliummesh**]. 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 [**intro-edge**].

- *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 **CPU!** (**CPU!**), **RAM!** (**RAM!**) and disk space at the edge is by far more elaborate than in a datacenter or in the cloud.

- ***QoS!*** *(**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 **??** "methodology".

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." [**dsr-method**]

3.1 Methodology

According to the description in the introduction before, **DSR!** is used as a scientific method to construct the **PoC!** (**PoC!**). For this purpose two real world exmaples are build, which are then gradually refined. Those exmaples are descripte in detail in the following **??** "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 **??** "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!**.
- ***NAT!*** *(**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.
- ***ARM!*** *(**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

OS The **OS!** (**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 **GUI!** (**GUI!**), is used.

Figure 5: Map of the **PoC!**[**googlemaps**]

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 **VPN!** (**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. VPNs are simply not designed for unstable connections with high latency and possibly rapidly changing **IP!** (**IP!**) addresses. The two tools used 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 **??**, 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 **??**.

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 **FQDN!** (**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-Cases

3.3.1 Web-application

The first test, which is to be carried out on each environment, is to make a simple web application accessible via ClusterIP and distributed at the edge-nodes. 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.

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 [**ke-ake-gh**].

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, proxy 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 doe not fit well: "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." [**ke-docs-why**]
- *CNI* - the next logical step seems to be to use a **CNI!**, which would make all functionalities available. However, this means that many of the advantages promised by KubeEdge are lost: 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 [**ke-cni-no**].
- *EdgeMesh* - as described in **??** an additional componente, part of the KubeEdge Project can be used.

KubeEdge
Network Issue: Build on Top of: https://github.com/kubeedge/kubeedge/issues/2760 Solution:

Edgemesh (not enable by default): https://edgemesh.netlify.app/ CNI is not a recommand solution: https://github.com/kubeedge/kubeedge/issues/1662 EdgeMesh no Raspberry Pi and CRI-IO: https://github.com/kubeedge/kubeedge/issues/2946 No Cross-subnet Pod communication: https://github.com/kubeedge/edgemesh/issues/278

"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." https://kubeedge.io/en/docs/kubeedge/ - einfach nicht wahr!

3.3.2 Enterprise VPN

3.3.3 Distributed Database

3.4 Performed Tests

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[**kiss**], those solutions with less complexity should be preferred. One investigates the installation routine, descripted in the **??** "environment", very different steps and necessary tools were discovered.

Dependencies The first important variable we look at in this appendix 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, even behind carrier-grade **NAT!**[**cg-nat**], important conditions must be checked for KubeFed, as descripted in **??** 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 **CRI!** (**CRI!**)[**k8scri**][**ke-cri-gh**], 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!**
    - KubeEdge binaries
    - K3s (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, is only done by using the binary. However, if logging is necessary (in almost all cases it should be), some adaptations must be made. For Kubefed, in contrast, the respective kubeconfig must be transferred to the master and added there as a custom context. Minor adaptation with little effort is necessary for both environments, yet the comparison is not balanced because KubeFed does not offer out-of-the-box the possibility to collect all logs of the edge devices.
- *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.

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

Type	Subtype	Criteria	**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	**3/2**	**4/2**
configuration	handling	necessary effort	**moderate**	**moderate**
configuration	documentation	clarity/actuality	**good**	**good**

Table 2: 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.

Type	Subtype	Criteria	**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 3: 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 [**ke-releases-gh**][**kf-releases-gh**]. 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.

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 4: Github Stats overview

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.

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 **SIG!** (**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 **CI/CD!** (**CI/CD!**) pipelines. KubeFed thus offers better independence.

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

Type	Subtype	Criteria	**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 5: future-proof overview

4.2 Criteria catalogue

The following catalogue of criteria can be compiled on the basis of the data obtained in **??**. 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"[**nwa**]. 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.

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

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%	4	3	0,08	0,06
		configuration	2%	3	3	0,06	0,06
		documentation	6%	4	4	0,24	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 6: Criteria catalog table

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?

Some introdution to K8s at the Edge, highlighting the main Architectures.

5.2 Extend Cloud to Edge with KubeEdge

Descripes KubeEdge and its advantages

5.3 Sharpening Kubernetes for the Edge

Sharpening Kubernetes for the Edge Make Kubernetes aware of the latency between the nodes at the Edge.

5.4 Ultra-Reliable and Low-Latency Computing in the Edge with Kubernetes

Similar to the paper before. Latecny awar pod deploymentm, but you also can deploy to regions and an custom re-scheduler is implementated taking care of redeploying when one node fails. Clustering node-groups based on latency.

6 Results

6.1 Findings

6.2 Conclusio

6.3 Discussion and further research

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

A Appendix