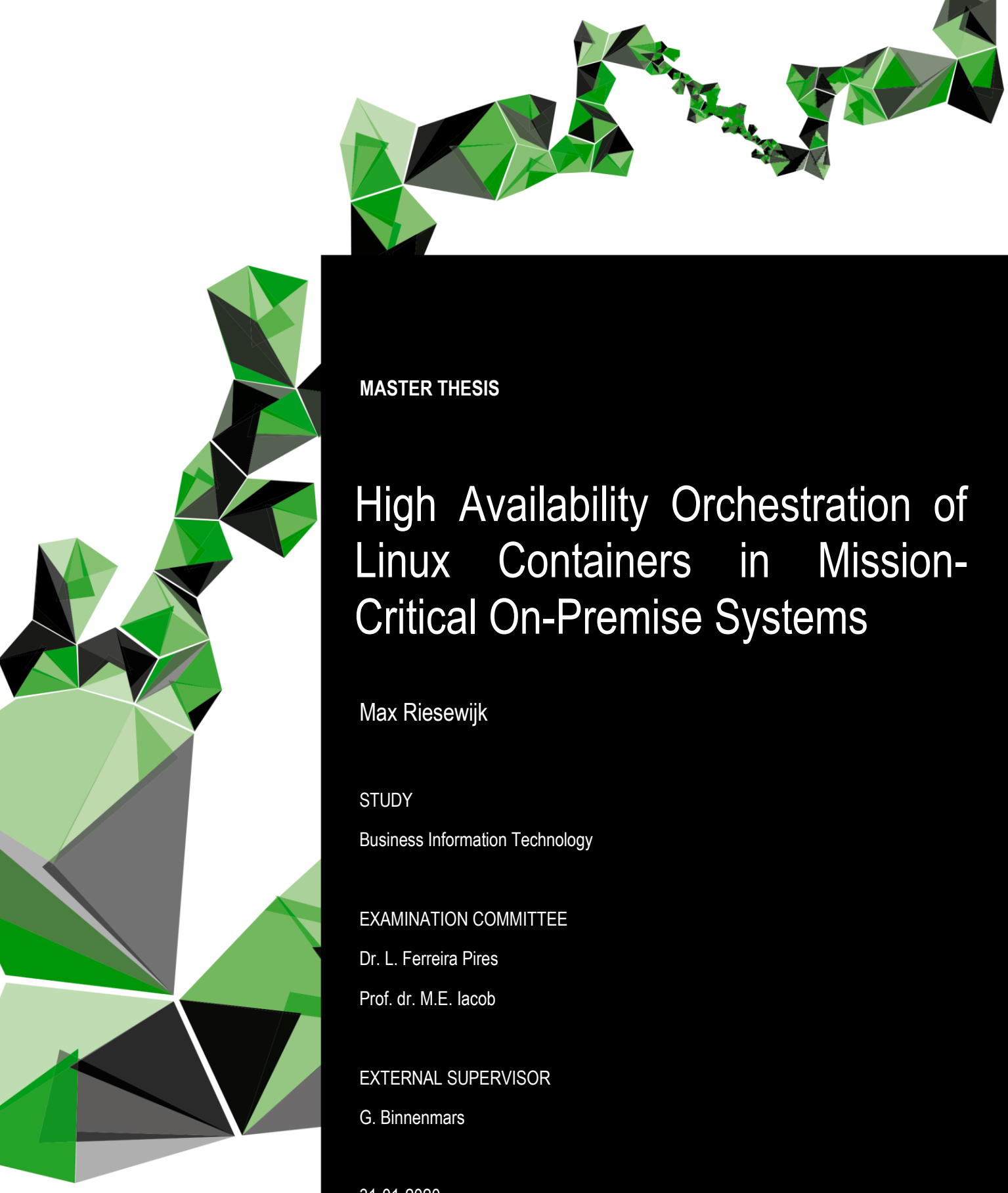

MASTER THESIS

High Availability Orchestration of Linux Containers in Mission-Critical On-Premise Systems

Max Riesewijk

STUDY

Business Information Technology

EXAMINATION COMMITTEE

Dr. L. Ferreira Pires

Prof. dr. M.E. Iacob

EXTERNAL SUPERVISOR

G. Binnenmars

31-01-2020

THALES

UNIVERSITY OF TWENTE.

Acknowledgements

I would like to thank several people without whom the completion of this thesis was not possible. First of all, I would like to thank my external supervisor at Thales, Gerrit Binnenmars, for his knowledgeable and continuous feedback, even during busy times.

Secondly I would like to thank my UT supervisors, Luís Ferreira Pires and Maria Iacob, for their help in shaping this thesis into the work it currently is. Their extensive knowledge of graduating at the UT also helped in correctly dealing with the sometimes bureaucratic process of graduating.

Lastly, I would like to thank my girlfriend, family and friends for their continuous support during the writing of my thesis.

Executive Summary

Maintaining mission-critical systems in the defence domain is a costly and labour-intensive process, since these systems have long lifetimes and strict requirements. These systems consist of multiple applications, of which some are Operating System (OS) or hardware dependent. This dependency complicates maintenance and inhibits innovation. Developers of mission-critical systems are now looking towards containerization to tackle this challenge.

Containerization allows applications to run independently from hardware or OS, this renders the overall system more maintainable. Containerization has gained a lot of popularity in the last years after Google's introduction of Kubernetes (K8s), which sparked the beginning of the Cloud Native Computing Foundation (CNCF). K8s is an orchestration technology for containers allowing clusters of containers to be managed with relative ease. The CNCF hosts and supports several technologies and projects that support containerized workloads. These technologies have a relatively low maturity compared to the technologies currently used in the defence domain. Companies active in this sector are therefore carefully considering these technologies before implementation.

One of the aspects which was not yet clear was how to achieve High Availability (HA) in an on-premise context using Cloud Native technologies. This research has set out to design a K8s cluster which satisfies the availability requirements of mission-critical on-premise systems in the defence domain. Several research steps have been conducted to achieve this goal.

Firstly, an overview of HA in cloud environments as described in scientific literature was gathered. After this, K8s was introduced. This technical introduction included an overview of the current scientific knowledge on K8s availability, as well as a technical overview of K8s Federation. By federating, multiple clusters can be managed by a single management interface. Domain experts indicated that federation could help in achieving the required availability. After the technical background a current example of a mission-critical on-premise system is discussed. Implementing containerization and K8s is currently considered for this system. This information was gathered through interviews with domain experts. These interviews were also used to gather knowledge on the requirements for a new system using K8s.

We have designed five reference architectures which fulfil the requirements of mission-critical on-premise systems. These architectures were evaluated based on the answers to a questionnaire conducted with domain experts. The domain experts found the federated solutions too complex for the context, and preferred single cluster solutions over the federated options. Two single cluster solutions have been prototyped. We have defined a testing strategy based on available strategies in literature and applied this strategy to the prototypes. The prototypes proved to react accordingly to several failure scenarios and therefore are likely to satisfy the availability requirements.

The single cluster designs differ considerably in one aspect, namely in the amount of Availability Zones (AZ) needed to achieve HA. The first design requires three AZs to eliminate all Single Point of Failures (SPOF), this is a more standard cluster design, but it does require an on-premise redesign to support three AZs. The other design eliminates all SPOFs by using two AZs, which fits to the current physical design of the on-premise context. This design is based on more experimental technology. We recommend implementing the design across three AZs if redesigning the on-premise context is not too costly and recommend deploying the design across two AZs if redesigning is deemed too costly and the in-house expertise is present to help in developing and correctly operating the experimental technology.

Table of Contents

List of Figures

List of Tables

Acronyms

A	Assumption
AZ	Availability Zone
CERN	European Organization for Nuclear Research
CIC	Combat Information Centre
CNCF	Cloud Native Computing Foundation
CPU	Central Processing Unit
DDS	Data Distribution Service
ECC	Environment Controlled Cabinet
HA	High Availability
IaaS	Infrastructure as a Service
K8s	Kubernetes
LSS	Logical Subsystems
MOC	Multifunctional Operator Console
MOM	Message-oriented middleware
MTBF	Mean Time Between Failures
MTTF	Mean Time To Failure
MTTR	Mean Time To Repair
OS	Operating System
PaaS	Platform as a Service
RA	Reference Architecture
RAM	Random Access Memory
RQ	Research question
Req	Requirement
SaaS	Software as a Service
SIG	Special Interest Group
SLO	Service Level Objective
SMR	State Machine Replication
SPOF	Single Point Of Failure
STONITH	Shoot The Other Node In The Head
VM	Virtual Machine
VMM	Virtual Machine Manager
VRRP	Virtual Router Redundancy Protocol

1 Introduction

The advent of Cloud Computing has created an environment in which computing power has become ubiquitous [1]. Large enterprises like Google, Amazon and Microsoft offer Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) services. These services offer elasticity in computing power, this eliminates the problem organizations face regarding under- and over-provisioning of computing hardware. Next to this, these services mitigate the need for upfront capital investment in expensive hardware.

In both PaaS and IaaS, the customer has access to an arbitrary amount of computing power in the cloud. This computing power is divided accordingly among the customers. A certain level of isolation has to be guaranteed for customers not to be able to enter each other's processes and to simplify the billing process. This need for isolation has created a new interest in both Virtual Machines and container software.

The rise of Cloud Computing and consequent increased interest in VMs and containers coincides with the increasing shift towards microservices architectures. Microservices replace large monolithic systems as smaller individual services with common interfaces. This way, the application landscape becomes easier to maintain (in theory), and system upgrades can be done more gradually, which allows for faster development and innovation.

Due to the popularity of cloud computing and its related technologies the Cloud Native Computing Foundation (CNCF) [2] has been founded. This foundation supports a set of open-source technologies that enable enterprises to facilitate deployment of applications and workflows in the cloud. Although the common association with cloud computing often involves large public cloud, the technologies supported by the CNCF can also be implemented in private clouds, which is the focus of this research.

This research has been conducted at a multinational company producing (among others) mission-critical systems for navies. The case study provided by the company has been used as a reference to mission-critical systems. The research considers systems deployed on naval ships, where these ships are regarded as being the on-premise context. Therefore, when discussing mission-critical, on-premise systems the context of defence systems on naval ships is meant, in case mission-critical systems in other domains are discussed it is specifically mentioned as being in a different domain.

1.1 Problem Statement

Currently, mission-critical systems are often monoliths where parts of the system strongly depend on hardware or Operating System (OS). This complicates the maintainability of these systems, and in combination with the long lifecycles of these systems (>30 years) it results in a very costly maintenance process. Next to this, the modern world has brought accelerated changes to the defence industry, which results in faster changing requirements. This requires systems to be highly adaptable, which current mission-critical systems do not adequately satisfy. To address these kind of challenges, industries in many different business domains have turned to containerization, which has been used to reduce complexity of maintenance [3] and make application hardware and software independent [4], while supporting microservices architectures. Introducing containerization does however also introduce a host of new uncertainties, which is not desirable, especially in the defence domain in which systems have to comply with stricter requirements than other domains.

One of the most important requirements for mission-critical systems is High Availability (HA), which is sparsely addressed in academic literature. Most HA systems deployed in the commercial domain boast

availability metrics commonly referred to as the 'five nines', which denotes an availability of 99.999%, meaning a yearly downtime of five minutes and fifteen seconds is tolerated. Systems in the defence domain are often subject to even stricter availability requirements. Next to this the Cloud Native community mostly focuses on public cloud solutions, whereas mission-critical systems are often on-premise solutions, these are not often addressed by the community. Therefore, it is unclear how to incorporate containerization into these types of systems. To effectively manage containers on computing hardware, orchestration technology can be used. Orchestration technologies (of which the de facto standard is Kubernetes [5]) can deploy and maintain the containers on the available hardware. Kubernetes can be configured in a plethora of different ways, some of which are more fitting for the on-premise mission-critical systems, while some are more fitting for the public cloud. Due to the focus of the community on public cloud, not a lot is known about the most appropriate configurations for on-premise mission-critical solutions. Therefore, a careful consideration of these configurations is needed.

1.2 Research Questions

To adequately address the challenges described above, a main research question has been defined.

How can container orchestration be deployed in an on-premise context so that it satisfies the availability requirements of mission-critical systems in the defence sector?

To answer the main research question, several subquestions have been defined, to answer parts of the main research question.

RQ1. What is often meant by High Availability (HA) and how is it generally achieved?
   a. What are the implications of HA in the cloud environment?
   b. What are mechanisms generally implemented to achieve HA in cloud environment?

RQ2. What is Kubernetes (K8s) cluster federation?
   a. What is K8s and what are its related concepts and technologies?
   b. What is the state of the scientific knowledge on availability with K8s?
   c. What are methods to achieve HA with K8s, including multicluster solutions?

RQ3. What are the business considerations for implementing Cloud Native technologies in mission-critical systems?
   a. What are current applications of Cloud Native technologies in the business domain?
   b. What are the business challenges for companies creating mission-critical systems and how can Cloud Native technologies help in addressing these challenges?
   c. What trade-offs should be considered when developing the system?

RQ4. What is currently a typical architecture for mission-critical systems and how are challenges regarding HA currently addressed?

RQ5. What are the proposed reference architectures that satisfy the requirements of mission-critical systems?
   a. What are the requirements of the on-premise container clusters, especially in relation to availability?

RQ6. Do the proposed reference architectures satisfy the requirements?
   a. How to evaluate and validate the reference architectures?
   b. Which architecture fits the specified requirements?

The product of this research consists of two main contributions, namely two reference architectures that can be used to implement cluster orchestration technology in mission-critical, on-premise systems in the

defence domain and a complementary advice on under which conditions which reference architecture can be implemented.

1.3 Research Approach

The problem statement and the main research question determine the goal of this research: designing a K8s cluster that achieves HA in the given context. The research sets out to create an artefact, the architecture of such a K8s cluster, and is therefore an example of design science. According to Hevner [6], design science creates and evaluates IT artefacts intended to solve an identified organizational problem. To create such an artefact with sufficient scientific rigor, several methodologies and frameworks can be used. Examples of available methodologies are the Design Science Methodology by Wieringa [7], which is a very in-depth method of conducting design research. Peffer et al. [8] provide a less exact method of conducting design science research, which leaves more room for the researcher to align the research with the practice.

The framework chosen for this research is Hevner's Three Cycle View of Design Science Research [9] due to its relative simplicity and ability to align the research with practice better than Peffers' method, while still maintaining scientific rigor. Alignment with practice is especially important since uncertainties about the designs should be reduced to a minimum due to the mission-critical nature of our target domain. The framework consists of three cycles, namely the Relevance cycle, the Design cycle and the Rigor cycle. These cycles are shown in Figure 1.

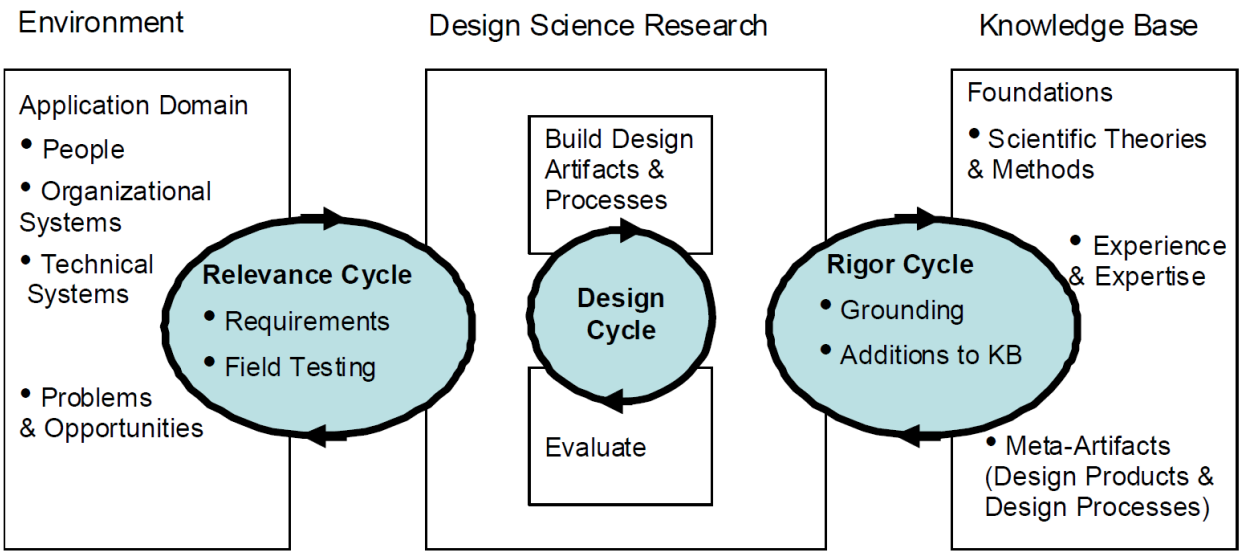

Figure 1: Design Science Research Cycles

In Hevner's framework, the Relevance cycle creates the context in which the to-be designed artefact should be applied. It provides the requirements and defines the acceptance criteria of the designs. The Rigor cycle provides the available (scientific) knowledge. We used the methods found in scientific literature to define a testing strategy. This cycle also shows the gaps in the knowledge base and ensures that the created designs are contributions to the knowledge base, instead of routine designs already available. The Design cycle consists of building and evaluating the artifact, which is the goal of this research.

This research has used a combination of literature research, interviews, questionnaires and design activities to create the artefact. The literature research has mainly been conducted in the Rigor cycle and provides the theoretical basis for the research, this theoretical basis was used to provide sufficient scientific rigor. This research activity also shows that the current knowledge base does not fit the considered domain very well, since it contains gaps. Interviews were conducted in the Relevance cycle, to acquire information on this research's context and on the requirements for the artefact to-be designed. Five reference architectures were designed in the Design cycle. The preferred way of addressing the challenges was found using a questionnaire to evaluate the proposed reference architectures. These preferred reference architectures have been prototyped. The testing strategy which was used to test the prototyped reference architectures was based on earlier testing strategies found in the Rigor cycle by literature research, these found strategies formed the basis for our testing strategy, which was again created in the Design cycle. This testing strategy has been used to asses the prototypes.

1.4 Thesis Structure

The thesis is further structured as shown in Figure 2:

- Chapter 2 provides the technical background of this research. It describes HA in the cloud environment, which provides an overview of the concepts that are applicable to this research. Next to this it gives an introduction of K8s' concepts, as well as the current literature on K8s availability and available multicluster technologies.
- Chapter 3 analyses the problem by describing common business considerations for introducing containerization into various business domains, this is complemented by specific considerations for the defence domain, as found during this research. The chapter continues with context of on-premise, mission-critical systems in the defence sector by describing a typical example of such a system. The chapter ends with the requirements for the system, which are a consequence of the specific domain in which the system should be deployed.
- Chapter 4 gives an overview of the designed reference architectures. By conducting questionnaires, the preferred architectures are selected for further research.
- Chapter 5 describes our testing strategy for testing some of the non-functional requirements, it includes the specifics on how the preferred architectures have been prototyped. The chapter also validates the strategy by applying it on a HA cluster. The chapter ends by validating the preferred reference architectures through the requirements and the testing strategy.
- Chapter 6 gives the report with the conclusion and discussion of the research.

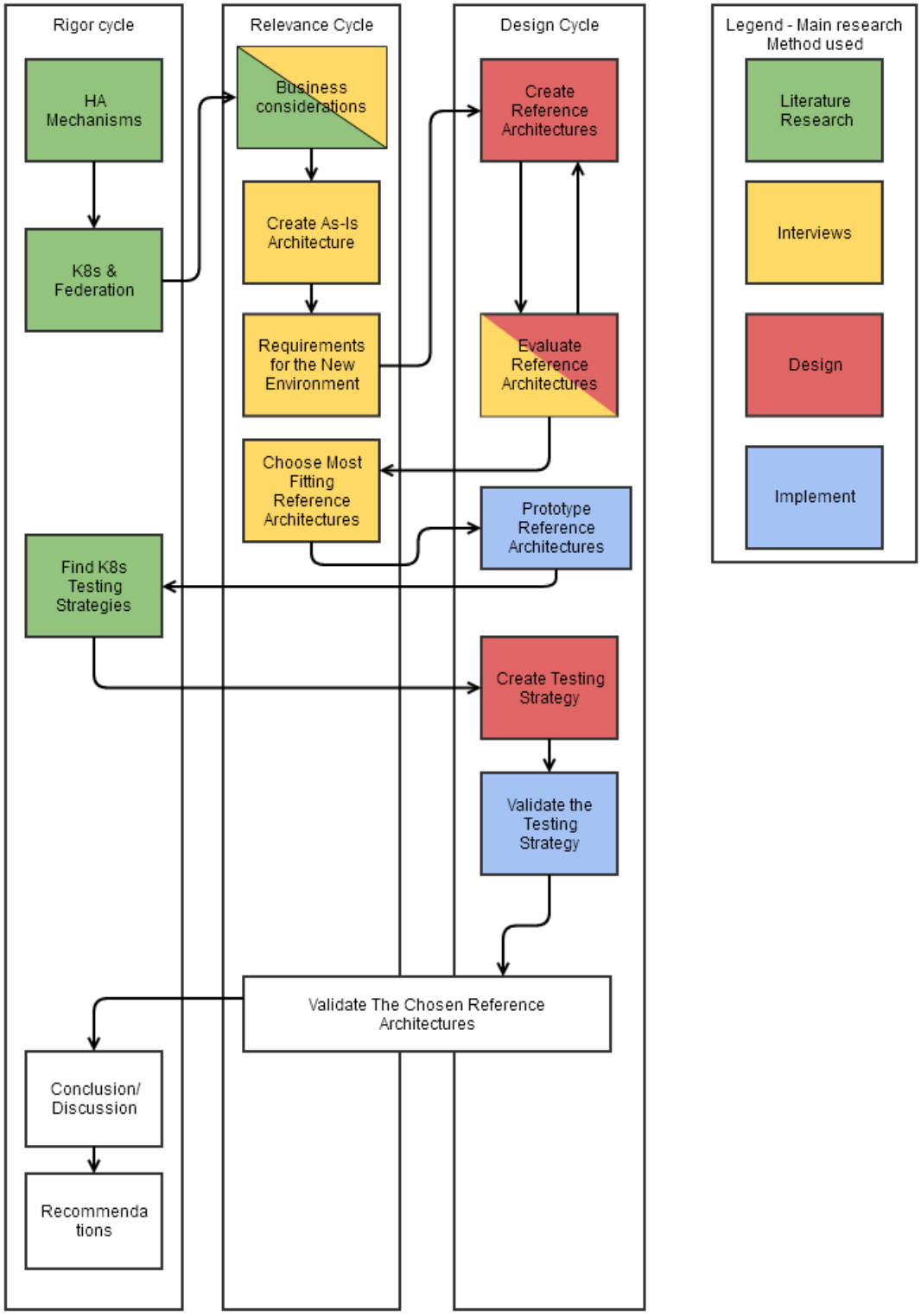

Figure 2: Overview of research steps

2 Technical Background

This chapter presents the information required for understanding the technical domain of the research. The information is gathered through literature research and by using a selection of technical documentation. The first part of this chapter focusses on High Availability (HA) in the cloud environment. The second part focusses on the technical concepts of K8s and containerization, it also touches upon the literature available on K8s availability and on how HA is generally achieved with a K8s deployment. The chapter ends with an overview of the available K8s solutions for multi-cluster management. This chapter describes research activities conducted in the Rigor cycle.

2.1 High Availability in the Cloud Environment

To make sure HA and its related concepts are applied consistently in this research, a definition of these concepts is needed. Therefore, scholarly work is used in this section to create a consistent view on availability and its related concepts for the further research.

2.1.1 Definition

Nabi et al. created a literature review on availability in the cloud [10]. This literature review is used as a reference to define the concepts surrounding availability. Nabi et al. [10] refer to Toeroe and Tam [11] for their definition of availability: 'the degree to which a system is functioning and is accessible to deliver its services during a given interval time'. This definition is represented by the formula:

$$Availability = MTTF/MTBF = MTTF/(MTTF+MTTR)$$

where MTTF is Mean Time To Failure, MTBF is Mean Time Between Failures and MTTR is Mean Time To Repair. High Availability is achieved when the service is available at least 99.999% (five nines) of the time for most business domains, although the defence domain has a stricter definition.

Toeroe and Tam also proposed a definition for service availability [11]:

$$Service\ Availability = Service\ Uptime/(Service\ Uptime + Service\ Outage)$$

Due to the nature of availability mechanisms employed by most container orchestration solutions, the Service Availability definition fits better in our research context. Container orchestration technologies mainly focus on supporting microservices architectures, in these architectures, components are often replicated several times. Because of this replication, the repair times of each individual component is not very important, since the replicated elements still provide service, which contributes towards Service Availability.

Nabi et al. also present a taxonomy for availability in cloud computing (Figure 3) [10]. This taxonomy can be used as a base reference for mechanisms and metrics for the further research.

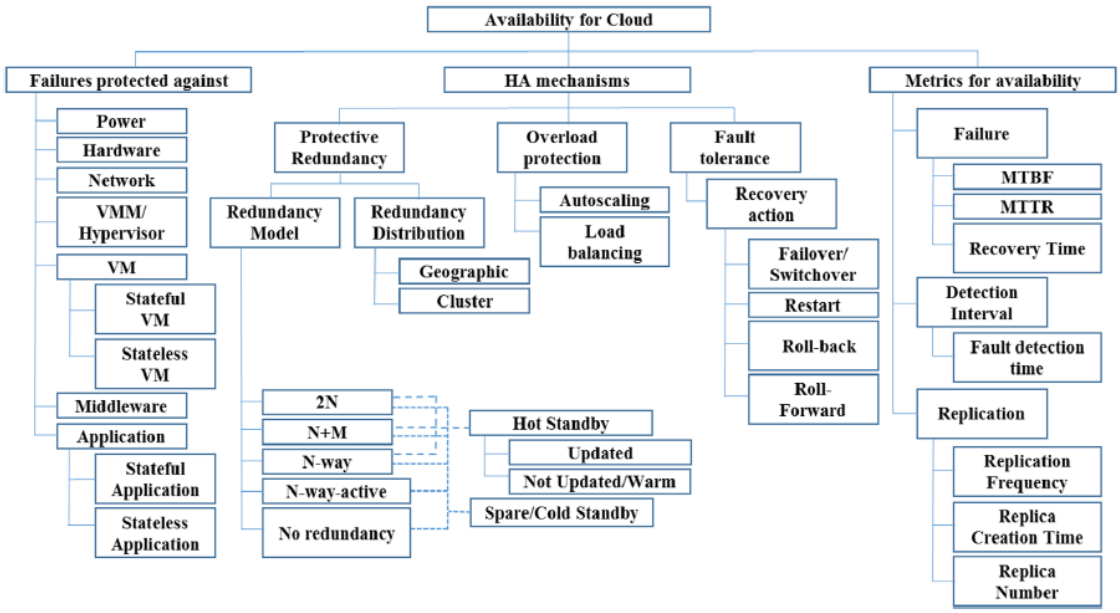

Figure 3: Taxonomy for cloud availability [10]

The taxonomy consists of three main categories: possible failures protected against, HA mechanisms and Metrics for Availability.

2.1.2 Failures Protected Against

The possible failure levels included in the taxonomy are listed and elaborated in Table 1. The taxonomy focuses on clouds running virtual machines, containers are not included. To make it more applicable to clustered, containerized environments we extended the list with container, container runtime and orchestration failure possibilities. VM failure is still included, since VMs are often provisioned as nodes in clusters.

Term	Explanation
Power failure	Loss of electrical power ranging from a specific component up to whole data centres
Hardware failure	Failure of a physical machine component (e.g. CPU, memory, hard disk)
Network failure	Failure of any network component
Virtual Machine Manager (VMM)/Hypervisor failure	Failure of the management system of virtual machines running on the host, resulting in failure of all VMs managed by the VMM
Virtual Machine failure	Failure of the operating system running within the VM, resulting in unavailability of the VM
Middleware failure	Failure of software components running on the OS which provide common functions to the applications
Orchestration failure	Failure of the orchestration software and its related components
Container Runtime failure	Failure of the container runtime software
Application failure	Malfunction or complete failure of components composing an application
Container failure	Failure of the container managed by the container runtime

Table 1: Failures protected against

VMs, containers and applications can be distinguished between stateful and stateless. Stateful elements keep track of their state. These states must be restored when an element fails, since states are often critical to the correct functioning of the element. Stateless elements do not keep track of their state, when these elements are replicated their state does not have to be restored to provide functionality.

2.1.3 Availability Mechanisms

The authors describe three overall availability mechanisms: fault tolerance, protective redundancy and overload protection.

Fault Tolerance

Fault tolerance is defined as the ability of a system to continue operating when a fault occurs. Several methods can be implemented to improve fault tolerance. These methods are listed in Table 2.

Concept	Explanation
Failover/switchover	Correcting a failure by moving the workload from a failed element to a redundant healthy element. In a switchover both elements are healthy, and transfer is done because of other reasons (e.g., load balancing)
Restart	The failed element is restarted to restore functioning state
Rollback	Restore the system or an element to a previous known working state
Roll-forward	Move the system into a new and working state

Table 2: Fault tolerance mechanisms

Protective Redundancy

Protective redundancy concerns the use of additional elements that would not be necessary if the system would always function correctly [11]. Active elements provide a service and when these elements fail, the redundant elements take over. Therefore, redundant elements are referred to a *standby*. A standby can be either a *Hot Standby,* meaning that the standby is instantiated and can take over the service from the active element in a relatively short time, or a *Cold Standby*, in which the element has to be instantiated after the active element has failed.

Several models for redundancy are defined in the taxonomy. These are listed in Table 3.

Concept	Explanation
2N	At least two redundant elements of which one provides a service and the other element protects the service as a hot standby.
N+M	The service is provided by multiple active elements (N) through workload distribution. The active elements are protected by several hot standby elements (M).
N-way	In contrast to N+M, redundant elements can serve some other service but still provide hot standby for the active elements.
N-way-active	No standby elements, all the elements provide service and are protected by other elements providing service, often by using a load balancer.
NoRedundancy	A single active element, services are protected by spares (cold standby).

Table 3: Redundancy models

Distribution

The taxonomy includes geographical distribution as a redundancy strategy. In the taxonomy's context, it considers geographical distribution across different Availability Zones (AZs). This is not an option in the on-premise context of our research. However, different physical machines on different places on-

premise are present (in a way, these can be regarded as different AZs). In this research, distribution will be considered across these different AZs.

Overload Protection

Overloading elements in the system can lead to performance degradation, or worse, outages of parts of the system. Therefore, measures have to be taken to prevent this from happening. By using auto scaling, resources can be adjusted dynamically to the current needs. To make sure that the workload is distributed across the scaled elements, a load balancer has to be implemented. Otherwise, requests could disproportionally go to a single element and not reach the scaled counterparts, leading to the aforementioned overloading problems.

2.1.4 Metrics for Availability

Several metrics for availability have been identified in the taxonomy. The formulas proposed by the authors for the 'Failure metrics' have been discussed in Section 2.1.1, therefore, these will not be repeated. The two remaining metrics are detection interval metrics and replication metrics. Detection interval metrics are described as 'the maximum time interval necessary to detect a failure' [10]. Replication metrics are related to the state synchronization methods between the active and redundant elements. These are listed in Table 4.

Concept	Explanation
Replication frequency	Time interval for state synchronization
Replica creation time	Total time required to replicate complete state information of an element
Number of replicas	The number of redundant replicas of a given element

Table 4: Replication Metrics

2.1.5 Additional concepts

Several mechanisms which are regularly employed in cloud environments are not discussed by the taxonomy. The additional concepts consist of Single Point of Failure (SPOF), quorum and STONITH/fencing. Two of these concepts, quorum and Shoot The Other Node In The Head (STONITH) do not directly increase availability, but these concepts are needed in conjunction with certain forms of protective redundancy to ensure data integrity.

SPOF

One aspect which can prevent a system from being HA is a SPOF [12]. A SPOF is a part of the system which, if it would fail, brings down a complete (or a large part) of a given system [13]. Preventing SPOFs by introducing redundant elements is paramount to achieving HA.

Quorum

Quorum is a protocol in which a group of nodes can only use a shared resource with the agreement of a majority of the nodes [12]. Nodes in this sense are software elements. Quorums are used to coordinate the state of stateful applications when protective redundancy is utilized (N+M, N-way and N-way active). When a quorum protocol is used, a majority of the replicated stateful elements must agree on data modification operations. If this majority is not reached the write operation will not be executed. This ensures that two groups of elements will not be able to write to the same data object, which could lead to data inconsistency, or worse, data corruption.

An effective quorum cluster consists of an odd number of nodes, otherwise an exclusive majority in case of network partitioning cannot be created. In a three-node cluster, the quorum would be two. In case one node loses communication with the other nodes, it will not try to write to the data object, since it

cannot acquire the majority vote (namely two votes). The other nodes can still write to the data objects, since these nodes can acquire the majority vote. Therefore, to form a quorum at least three nodes are necessary. If only two nodes are available, an administrator could consider installing a quorum witness, which is a small computing unit that is added to achieve the quorum. A quorum witness generally only saves metadata and can vote on read and write operations. This computing unit does not have to process large amounts of data or calculations, and can therefore consist of relatively inexpensive hardware.

STONITH and Fencing

Two node clusters require different techniques to ensure data consistency, since no quorum can be formed [12]. If the two nodes lose communication and both nodes assume the other one is down, a situation referred to as 'split-brain' occurs. In a split-brain situation both nodes assume leadership, assuming it can write to data objects, which could lead to inconsistency and data corruption. To prevent this, a method called fencing [14] can be used, of which two distinct categories exist. Resource-based fencing is the process of denying a node access to physical components such as storage, to make sure it does not corrupt data. STONITH based fencing is a software solution that powers down/reboots a (partially) failed node. This way the failed node can restart, which can solve a multitude of hardware and software problems. In both fencing methods, one node fences the other node in order to avoid data corruption. Fencing methods are used to create a certain situation out of an uncertain one. Nodes can lose communication through a myriad of causes (e.g., network failure, application failure, hardware failure). Because the other node does not know the actual cause of the failure, it uses a fencing method to make sure that the node does not write to shared resources and through this ensures data integrity. STONITH aims to reboot a (apparently) failed component, this component gets a reboot command with the intent to boot the component into an operational state in which it can successfully rejoin the cluster, without the risk of corrupting data objects. When implementing STONITH, careful consideration of the used configurations is required, in order to prevent a situation where the two nodes keep sending reboot commands to each other. This would result in a complete outage of the services provided by these nodes.

2.2 Cloud Computing & Kubernetes

This section introduces the technologies considered in this research. The overview is vital for understanding the remaining research. It provides information on Virtual Machines, containers and Kubernetes (K8). It dives deeper into K8s, due to the focus of this research and the complexity of the technology. Since the research focusses on availability, this section also describes how HA is generally achieved in K8s environments.

2.2.1 Virtual Machines (VMs) and Linux Containers (LXC)

VMs use Hypervisors to offer a way of running multiple operating systems on a single system. The single system acts as a host, provisioning resources to each guest VM. Often, each VM enables a single service in a service-oriented architecture. The Hypervisor ensures complete isolation between the VMs. Therefore, VMs can be used to easily deploy different applications in cloud services. Each VM runs its own OS installation. Virtual machines are used to run an isolated instance of a complete OS, allowing different OSs or different versions of an OS to exist on the same system. Containers on the other hand, share an OS and kernel [15]. Because of this, containers have less overhead and better performance [16, 17], compared to VMs. Boot times of VMs can vary between ten seconds to ten minutes [18, 19], whereas containers often boot within a second [19]. Mazaheri et al. found out that containers offer near bare-metal performance, showing that the performance impact of containerizing compared to bare-metal performance is not a large concern for most implementations [20]. Figure 4 shows a comparison between a server hosting VMs and a server hosting containers. Containers also allow applications to become hardware and host OS independent [4], which increases software maintainability. Therefore, containers are very suitable for maintaining software with a long lifetime. Several more container technologies exist (of which Docker is used in this research), but the technological details of the container technologies are not relevant for this report and are therefore left out of the scope of the research.

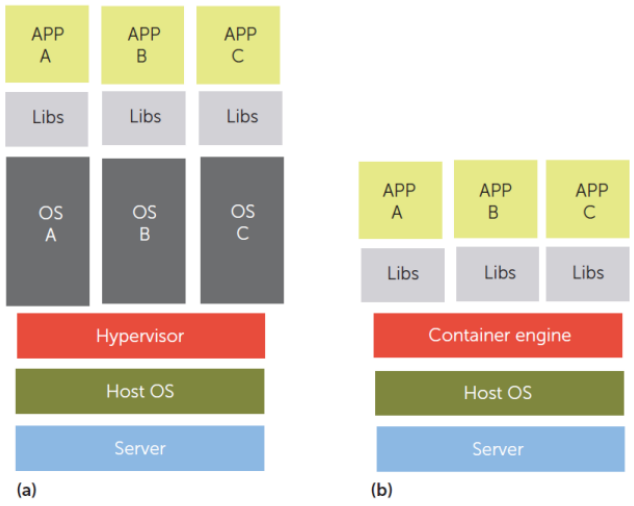

Figure 4: Comparison between hypervisor (a) and container-based (b) deployments [8]

2.2.2 Kubernetes (K8s)

This section describes K8s using information gathered from K8s documentation [21, 22, 23] and online resources [24]. Since the focus of the research is on availability, the impact of each component on availability is documented as well. Next to this, methods to increase availability for each component are

mentioned, focusing on eliminating SPOFs. First, a general introduction into K8s is given, followed by the different components in greater detail. Figure 5 shows how the K8s components are related in a cluster consisting of a single master node, a single worker node and a client which interacts with an arbitrary application deployed in the cluster. Figure 5 has purposely ignored implementation details.

K8s [5] is a container orchestration tool backed by the Cloud Native Computing Foundation. The support of large enterprises like Google, IBM and Red Hat makes K8s a potentially suitable option as an orchestration tool, when considering the requirements associated with mission-critical software (HA, multi-decade long maintainability and the availability of documentation and support). K8s has the largest community base of any container orchestration tool, currently having over 85000 commits submitted by more than 2300 contributors [25].

K8s' top-level component is a cluster. Clusters are computing pools consisting of a number of nodes. These nodes can reside on physical machines or on virtual machines.

Pods are the lowest-level concept in K8s, and these are scheduled on the nodes. Pods wrap one or more containers. These containers share the same resources and local network. Therefore, the containers in these pods can communicate with each other, lowering the network overhead for tightly coupled applications. Pods can easily be recreated, this can be done in order to resume service after a pod failure occurs. Another reason to recreate pods is to scale the application when demand increases, making use of the elasticity of the cloud. Therefore, it is advisable to only bundle containers in the same pods when these containers are very tightly coupled. Otherwise, parts of the application for which demand is not increased will also be scaled, resulting in resource waste. The ease of pod recreation contributes towards better availability. In case applications residing in the pods are stateful, it requires a form of state replication to properly restore the state when recreation is required. To increase availability, K8s offers livenessProbes, which check the health of each container instead of only the health of each pod.

Pods are managed by controllers. Different kinds of controllers for sets of pods exist. The most common controller is the 'deployment', which specifies the number of replications of a specific pod that should be running. A deployment monitors the specified pods it is configured to manage, and automatically creates a new pod when a pod under its management fails, trying to gracefully terminate the failed pod. Pods can be scaled horizontally, using a load-balancer to distribute requests between pods. This way, when a pod fails, the load-balancer can direct traffic to other (functioning) pods. In the meantime, a new pod is created and the failed one is removed, without interrupting the service provisioning to the user. Other controllers include Jobs, which make sure a specified number of pods successfully finish their job, and DeamonSet, which can specify which specific pods should run on each node. Several more controllers exist, but these are not considered relevant in this report.

By default, a pod is isolated from outside the node it resides on. To allow communication between pods in the cluster, services are deployed. To allow access to the pods from outside of the cluster, ingress controllers are used, which also offer load balancing between pods. Services provide an abstraction layer used to keep track of the IP-addresses of pods and are equipped to route traffic between clients and pods, and in inter-pod communication. Since pods die (or are terminated) and new ones are created regularly (depending on the demand fluctuation), the IP-addresses of these pods are subject to change. Services allow the pods to easily find each other by keeping track of these IP changes. Services can also offer the required load-balancing capabilities for horizontally scaled pods.

Nodes are managed in a master-worker architecture. One or multiple master nodes monitor and manage the available worker nodes. The master components form the cluster control plane, making global decisions, like scheduling. The components are essential to the correct functioning of K8s, therefore each component has an associated HA strategy. We will describe these strategies for each component.

Master Components

The first master component is the kube-apiserver. This component exposes the K8s api, allowing master and worker nodes to communicate, it also allows components to access the etcd cluster (introduced in the next paragraph). Next to this, the kube-apiserver allows other master components to communicate with the etcd cluster. This component scales horizontally, allowing several live replications to exist at the same time (active-active). If one fails, traffic is redirected through other instances by a load balancer. This load balancer has to be installed and configured separately, since it is not installed by default by most deployment tools.

The second component is etcd, which is a key-value store used by K8s. It contains all the data on the status of the cluster and the cluster's configurations. In the standard deployment, each master node has its own (stacked) etcd instance. Etcd clusters can be deployed independently from master nodes (external). This is necessary to guarantee HA in K8s clusters, because of the importance of etcd for the stability of K8s [26]. Dedicating an isolated environment to etcd clusters reduces the risk of resource starvation. A five-member etcd cluster is recommended in production. Etcd uses a quorum (as described in Section 2.1.5) to ensure HA, the quorum algorithm is implemented by Raft [27]. Next to replication, a back-up strategy for the data in the etcd clusters has to be in place as well.

The third component is the kube-scheduler. This component manages the allocation of pods to the worker nodes. It reads the etcd storage for unscheduled pods (via the kube-apiserver) and assigns these pods to the nodes. Kube-scheduler can be replicated by creating multiple master nodes, where the leader is decided through leader election. This leader election does not use quorum as an election mechanism, but uses leases which expire after a certain period. The leader will actively renew the lease, if the leader fails to renew the lease within a set timeframe, other instances will try to acquire the lease. The current lease is saved to the etcd storage, all instances of the kube-scheduler can retrieve the current lease from the storage through the kube-apiserver.

The fourth component is the kube-controller-manager. The data generated by this component is also saved in the etcd storage. It runs as a single process but consist of four controllers:

- Node controller: monitors the nodes
- Replication controller: maintains the specified amount of pods
- Endpoint controller: joins services and pods to create reachable endpoints
- Service Account and Token controllers: create the default accounts and create API access tokens for new namespaces

The kube-controller-manager can be replicated by having multiple master nodes, where the leader is decided by leader election (similar to the kube-scheduler).

The last master component is the cloud-controller-manager. It contains controllers that can communicate with specific cloud providers. Since the research ignores on clusters in the public cloud, functions offered by this component are outside the scope of this research.

Node Components

Node components run on both worker and master nodes and consist of three components. The first component is the kubelet, which is an agent that makes sure the pods described in the configuration are running and healthy. It communicates this information and the node's overall status to the master nodes and acquires the configuration needed for further operation. It is one of the few components that has to be configured as a system daemon and cannot be managed in the cluster. When a kubelet fails, it is registered by the master nodes as a node failure. This point of failure can be mitigated by replicating nodes. Next to this, the system init system must be configured in such a way that the kubelet is restarted after failure, to reduce the impact of a failure.

The second component is kube-proxy, which maintains network rules on the host and performs connection forwarding. A kube-proxy can be deployed inside a pod, so that when the pod fails the container-runtime will create a new pod. By replicating nodes, services remain functional when the failed pod is being recreated.

The last node component is the container-runtime, which manages the containers inside the pods. Several container-runtimes are supported by K8s. Which container-runtime is the most suitable depends on several factors. This choice is out of the scope of the current research. Container-runtimes scale with node scaling.

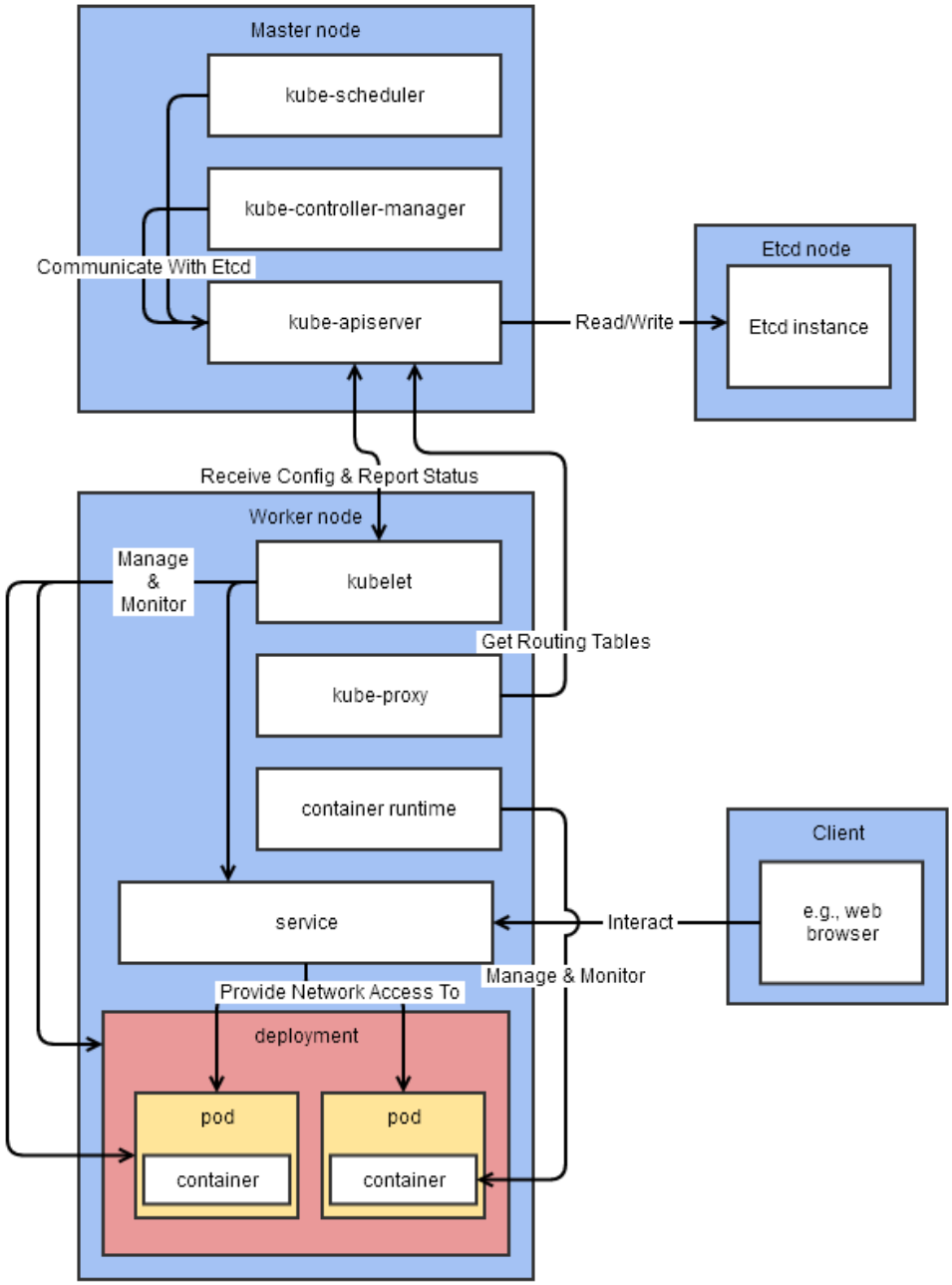

Figure 5: Single Master and Worker Node

2.2.3 K8s HA for a Single Cluster

This section describes the HA mechanisms for K8s introduced in Section 2.2.2 in more detail. In some parts, the leap between the low-level HA mechanisms and the high-level overview required additional information, which was gathered from online resources [28, 24, 23] and literature search [29]. For clarification, an UML deployment diagram [30] of the setup described is given in Figure 6. This deployment diagram intentionally ignored implementation details for more clarity. It shows three master nodes connected to a five-member etcd cluster. The master nodes communicate through a load balancer with an arbitrary amount of worker nodes. Each node and etcd member reside on its own server.

To mitigate critical failures of the master node components, at least two replications of the master nodes have to be created, although most deployments consist of three master nodes. Having three master components supports HA even when one of the nodes is updating. This way, a rolling update could update the master node in series and two master nodes will always be available. Two of the master node components cannot work in parallel (the scheduler and the controller manager). Leader election determines which master node will handle the leadership of these master components. The master components handle data objects. Therefore, horizontally scaling these components is not a viable option, since it leaves the possibility for two components to modify the same data object at the same time. Hence, only the leader is allowed to write to the storage. Leadership of different components can be distributed across the nodes, because leadership election is done by K8s on service endpoints, and each service (group of components) creates its own endpoints [31, 24]. In other words, each group of components chooses its own leader. Having the two master nodes on different physical machines ensures a more resilient system with regard to hardware failure. However, the failure of all master nodes simultaneously does not result in complete system failure, since the worker nodes will still be able to handle requests and reschedule pods by itself. However, the cluster will not be available to react to changes in the environment (e.g., node outage or increased requests).

Introducing leader election does introduce the risk of a situation referred to as split brain, as described in Section 2.1.5. While implementing a HA K8s cluster, countermeasures for this problem may have to be implemented. K8s does not employ any countermeasures itself but relies on the ability to converge its components to the desired state. One of these countermeasures is having (physically) redundant communication paths to make sure the communication is not interrupted [32]. Another countermeasure is fencing (described in Section 2.1.5), which locks out nodes when the status of a node is uncertain. There is no clear consensus on whether fencing must be implemented in HA K8s clusters. It could be superfluous, since K8s has advanced monitoring techniques for its nodes.

Worker nodes must be replicated in order to mitigate the effect of node failures. The number of replications required depends on several factors. Generally, more nodes should provide a higher availability, given that applications scale horizontally, and a functioning HA load balancer directs traffic to each of the elements. Some applications might need to be grouped on specific nodes. This leads to a situation where if group A of applications can only be scheduled on node A, and group B of applications can only be scheduled on node B, both node A and B have to be replicated a number of times to mitigate the risk of node component failures. Preferably, these replications will be scheduled on different physical machines, to ensure that not all replicated nodes fail when a physical machine fails.

Mitigating the effect of pod failure is achieved by replicating pods. The preferred way to do this is by enabling containers to work in parallel on the application level. This way, a load balancer can direct requests to all live pods. When a pod dies, the load balancer can direct traffic to living pods seemingly unnoticed, given that the load balancer offers health checking functionality on the endpoints. The failed pod will then be rescheduled by the kubelet on the node.

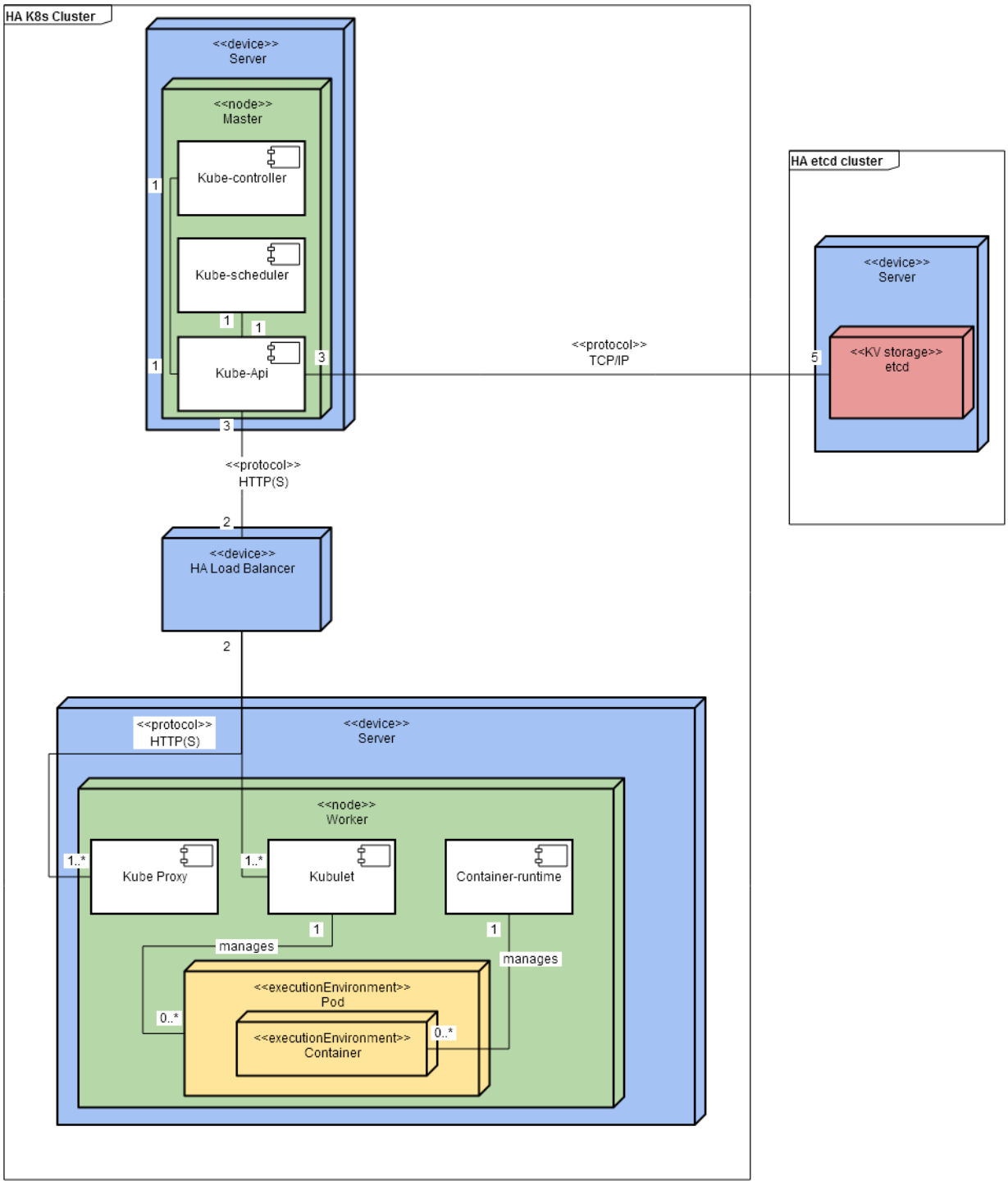

Figure 6: Overview of a single HA cluster

2.3 K8s Availability in Literature

The amount of literature available on K8s availability is small. The available literature has been assessed on its usefulness in the context of this research, which resulted in a set of five papers. Because the amount of literature is small, the search also included VMs cloud federation. We assumed that the

concepts and solutions proposed and discussed in these papers would provide relevant information to this research, but most found literature on this subject was tied to specific technologies or contexts. This literature was not generalizable to K8s environments, therefore it has not been included in this report.

Several investigations on K8s availability have been conducted. Vayghan et al. experimented with node and pod failures through administrative commands (deleting the pods) and by forcefully killing the process/VM running the pod or node [33]. The commands allow K8s to gracefully terminate the pod or node, whereas forcefully killing is more abrupt and requires K8s to detect the failure, instead of getting a notice of the termination. The results show that outage time is considerably longer when the pod/node is externally terminated.

Kanso et al. tested the responsiveness of K8s' HA mechanisms by testing *pod failure*, *host failure*, *Network failure* and *Master component failure*, using a self-defined testing strategy [29]. This strategy was designed to extract the failure recovery times of each failed component. Next to this, the researchers tested the overhead of different monitoring intervals. The researchers concluded that K8s' HA mechanisms are powerful, but there is room for improvement, since the default monitoring values are not sufficient to achieve HA. The researchers also found that changing the monitoring intervals does not have a large impact on the resource overhead. The research considers memory and CPU usage, excluding network overhead from the research. The researchers configured the monitoring intervals to its lowest possible value, therefore the outage time recorded was considerably lower than the times recorded by Vayghan et al. [33]. Vayghan et al. note that the kubelet posts the node's status every 10 seconds to the master [33]. The worker node is allowed to fail to deliver such a message up to four times, before being marked *NotReady*, leading to a mean reaction time to a node failure of 38 seconds. Kanso et al. found a mean detection time of 4.65 seconds in such an event by using the lowest possible monitoring interval [29]. Vayghan et al., in another research, also used the lowest possible monitoring interval [34] and came to similar results as Kanso et al. [29], suggesting that lowering this monitoring interval has a large impact on the reaction time.

Oliveira et al. implemented Raft (the consensus algorithm also used by etcd) as a container orchestrated by K8s [35]. Raft can be used to replicate machine states, providing a HA mechanism for when a machine fails. The research concluded that the Raft algorithm implemented in K8s provides an approximate of 17,4% lower performance than a bare-metal implementation. The researchers argued that this lower throughput is compensated by the benefits offered by containerization and using K8s. Therefore, Raft deployed as containerized instances on K8s clusters could prove to be a useful tool in container state replication.

Netto et al. created a protocol in K8s that uses shared memory, so that, State Machine Replication (SMR) is achieved more easily [36]. The authors mentioned that extra security layers will be needed in this implementation, therefore it is not applicable to our research. Some of the researchers have continued their work and created Koordinator [37], which implements SMR (by using Raft) by adding a layer of 'Coordination as a Service' to a K8s deployment. The authors argued that existing SMR libraries are hard to implement. By offering the SMR as a service layer, the applications do not have to include SMR-libraries, which should save implementation time and effort. Due to the scope of our research, SMR protocols will not be taken into account when considering HA clusters. In an actual real-world implementation of the HA cluster, SMR can be a valuable mechanism to increase availability, but applications which will support SMR will have to be adjusted to interact with the Coordination as a Service layer.

2.4 K8s Federation

Federation or multicluster management can be implemented to increase availability in the event of a critical failure that affects a cluster on a cluster-wide level. If multiple clusters are deployed, federation/multicluster management is required to ensure the set of clusters is working in a coherent way. Several options for multicluster management were found, although some lacked a lot of features due to their lower maturity. These technologies have been listed in Table 5, along with a category which describes the features the technology offers and the reason why it has not been considered further this research. KubeFed [38] was deemed to be the most promising technology for further research due to its community support and feature set. Hence, it is described in more detail.

Technology	Category	Reason for not considering
Multicluster-scheduler [39]	Multicluster control	Low maturity, low community support, too much unknown about actual workings and design
Crossplane [40]	Multicluster control	Documentation for bare-metal servers lacking, since the focus is on cloud-providers
Multi-cluster Ingress [41]	Cross-cluster load balancing	Only works with Google Cloud Load Balancer, therefore will not work in on-premise environment
Cilium ClusterMesh [42]	Cross-cluster communication	Only offers communication and no cross-cluster control plane, which results in having to implement such a control plane
Rancher Submariner [43]	Cross-cluster communication	Only offers communication and no cross-cluster control plane, which results in having to implement such a control plane
Istio Multicluster Service Mesh [44, 45]	Cross-cluster communication	Offers more functionality than ClusterMesh and Submariner, but still requires a separate K8s cross-cluster control plane for coherent multicluster operation

Table 5: Multicluster Technologies Not Further Investigated

KubeFed is a system developed by the K8s Special Interest Group (SIG) 'multicluster', which is the largest K8s SIG and attracts a lot of attention from the community. By implementing KubeFed, one interface to communicate with multiple clusters is created. One cluster acts as the master cluster, which creates the configurations for the other clusters. The master and worker cluster both contain a federation-api, which allows the master and worker cluster to communicate. Next to this, the master cluster contains a federation-controller, which acts as the control-plane for the federation. This controller generates configurations for the worker clusters in its federation, based on the configuration files created by the administrator. This way, member clusters can be replicated while still working in a coherent way. In theory, this offers better redundancy than a single cluster, but managing multiple worker clusters through a single master cluster does introduce a new SPOF, namely the master cluster itself, although methods exist to reduce the impact of a failure.

Although KubeFed is currently not suitable to support HA due to its low maturity, several strategies to ensure higher availability have been proposed or are already implemented. The first strategy is to allow multiple federation controllers to be deployed on the master cluster [46], this strategy is already implemented and leverages the same leader election capabilities as the kube-controller-manager and the kube-scheduler. These controllers follow a leader/follower pattern. In this configuration, only one master cluster is active and forms a SPOF. Implementing leader election across redundant master clusters is not a priority of the SIG [47], stating that it has an unfavourable cost/benefit analysis. The developers have stated that replication K8s API state across clusters is expensive, since this results in a constant overhead on the network, which would result in significant costs when utilizing the public cloud in different AZs. The public cloud is the focus of the developers, since it is the use case of the product for

most of the users. Next to this, reliable leader election across clusters is a problem that has not yet been solved [48].

Another strategy to increase availability is to implement pull reconciliation. In this setup, the federation plane does not directly communicate the configuration to the member clusters (push-based reconciliation) [49]. Instead, it posts the configuration to a separate storage, and the member clusters use agents to retrieve and apply the configuration. This way, when the master cluster goes down, the latest configuration can still be used by the member clusters. This also relies on the fact that when the federation control plane goes offline, the member clusters will continue servicing requests, in a similar fashion as non-federated clusters, where nodes will continue their workloads when the master node fails. For this behaviour to be effective, a load-balancer with health checking functions for endpoints must be used [50].

The last strategy discussed is using a back-up tool called Velero [51] to create snapshots of the master cluster. These snapshots can be used to start a new master cluster when the master cluster dies, through scripts and monitoring. This strategy is less of an HA strategy, since no redundancy is implemented, and more a disaster recovery system. However, these topics are closely related, and the back-up strategy could prove useful in achieving better service availability.

Completely eliminating SPOFs in a federated setup is not (yet) possible, since implementing leader election on a cluster level is not deemed to be a priority by the current developers, leaving the single master cluster as a SPOF. It is debatable whether a single cluster is a SPOF due to the distributed nature of K8s. The single HA cluster in practice does not contain a SPOF, since it is resilient to hardware and software failures on each level of the cluster, but the cluster itself is technically a SPOF, if for example a faulty configuration is given. Resolving this SPOF cannot be completely done by federation, since the master cluster is a single cluster, but federation offers several technologies which can help mitigate the impact of a master cluster failure. The lower level cluster federation technology (the federation controllers) can be replicated and makes use of leader election in order to increase redundancy. The back-up tool can be used to create snapshots of the master cluster, combining this with a script that creates a new master cluster when the current one dies, using the snapshot. To minimize the impact of the offline master cluster while the start-up script is running, the pull reconciliation strategy can be used. By saving the configuration to an external storage and allowing the member clusters to use this saved configuration, operations during the downtime of the master cluster can continue, even in case a member cluster has lost its configuration due to some outage in the cluster.

A system with the described HA capabilities (excluding the back-up software) is given in Figure 7. The depicted master cluster contains three master nodes, each running a federation controller, connected to a HA etcd cluster. The federation controllers push the generated configuration to a three times replicated VCS (version control system, in this case git). An arbitrary amount of worker clusters retrieve the configuration from this VCS. Each of the worker clusters is connected to a dedicated HA etcd cluster. In this representation, the master cluster only creates the configurations for the member clusters. The master cluster can also act as a member cluster and process workloads. For reference, Figure 8 provides a similar setup using push propagation. These representations are deliberately simplified and omit other components that are not KubeFed specific, like the kube-apiserver or the kubelet.

These strategies are, at the time of writing, not yet implemented. Leader election for the controller-manager will be included in the next release (v1beta1) [52]. Pull propagation is discussed in the SIG but no significant development efforts for this feature have been done due to prioritization of other tasks

[53]. Only the leader election for the controller-manager is implemented, but a comprehensive analysis of the effectiveness of this strategy (or a combination of the aforementioned strategies) has not been done yet [54]. To compare the availability of federated clusters to the other options considered in this analysis, experiments must be conducted.

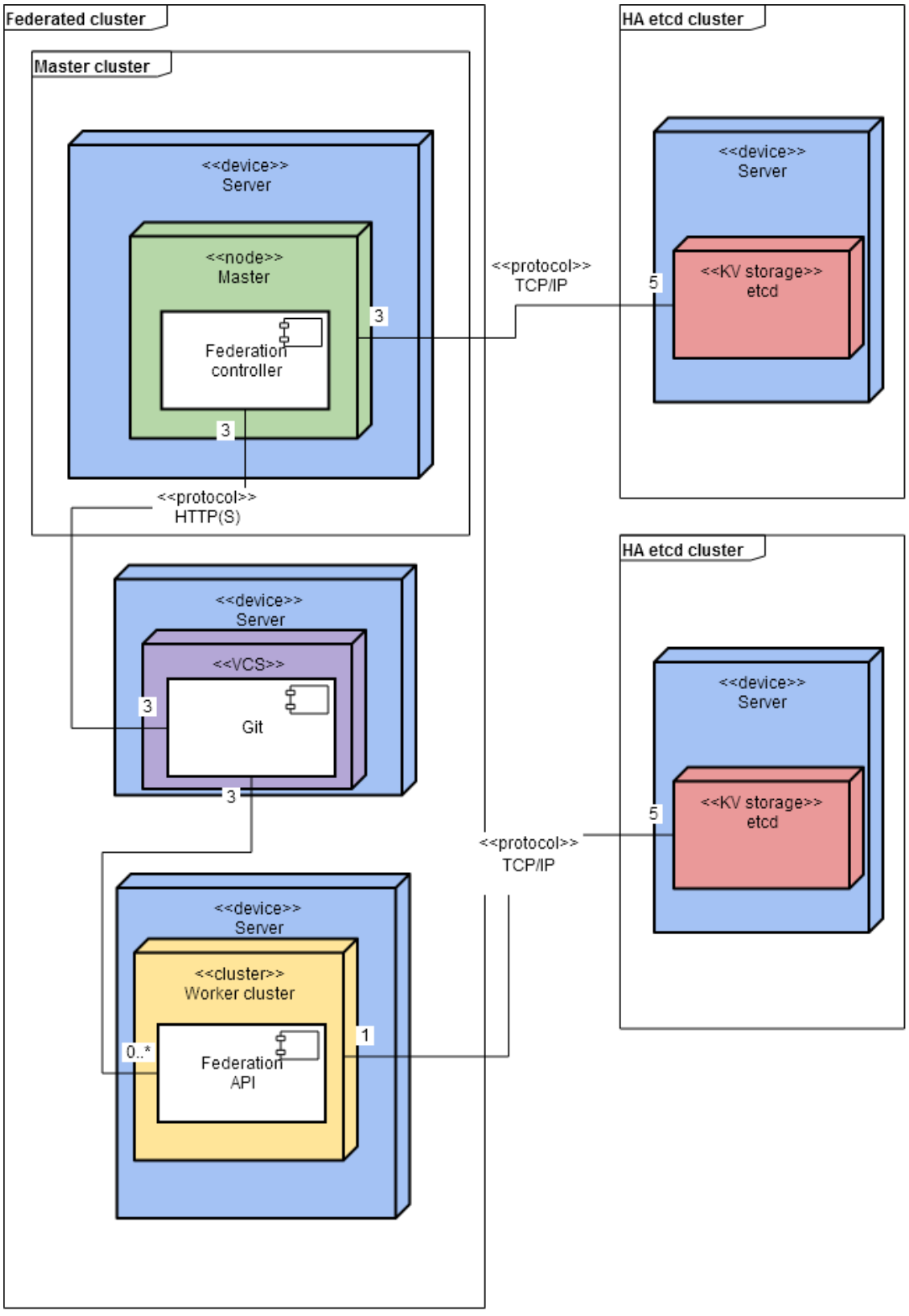

Figure 7: Federated cluster using pull reconciliation

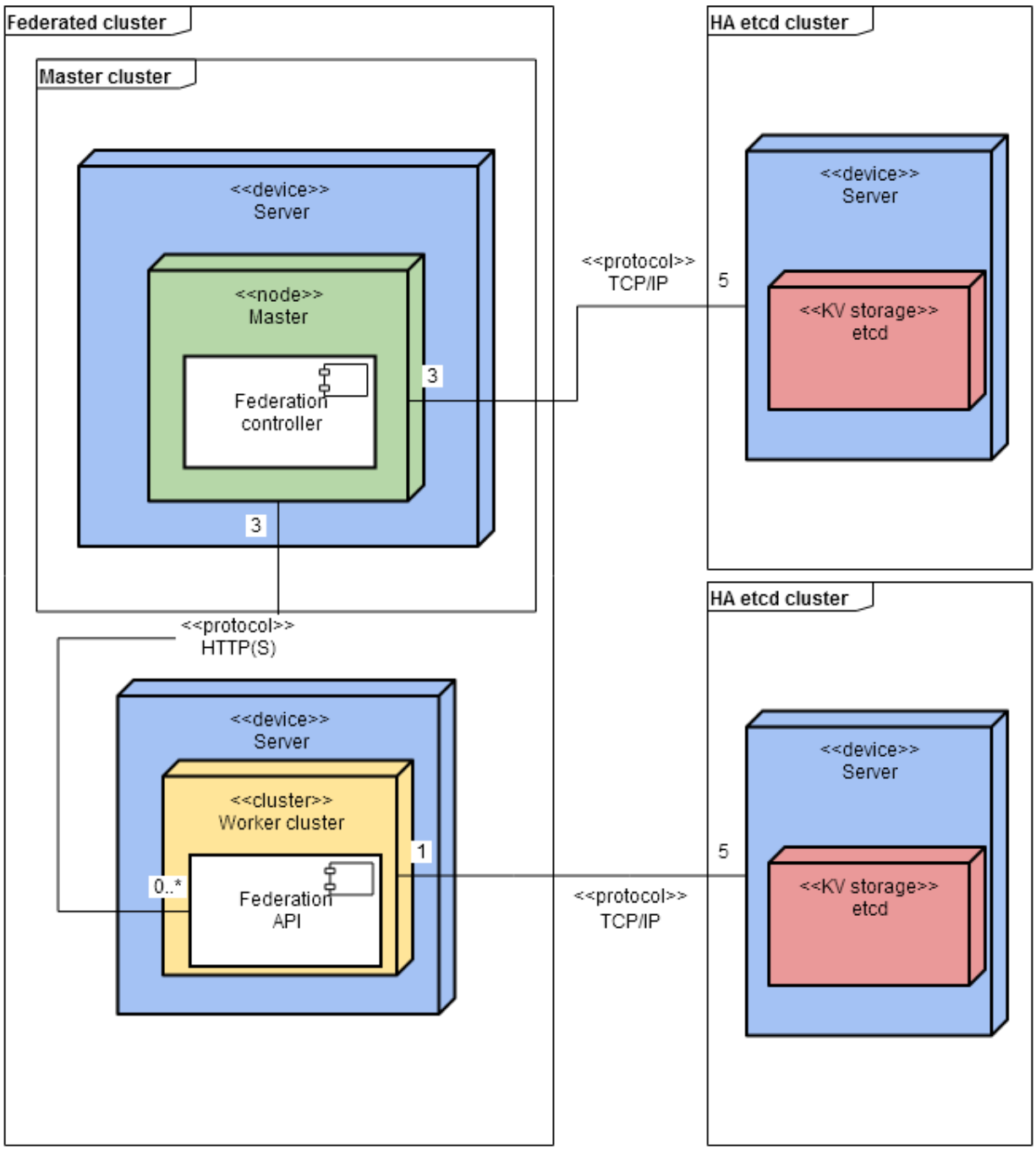

Figure 8: Federated cluster using push reconciliation

3 Problem Analysis

This chapter describes the relevant current challenges encountered with mission-critical systems in the defence domain. Some of these encountered challenges are similar to challenges found in other domains, although the defence domain often has stricter requirements, which requires more careful consideration to adequately address the encountered challenges. Considerations for implementing containerization in several business domains are discussed in this section and combined with specific considerations for the defence domain to provide an overview of the challenges that containerization can address. This results in an overview of why containerization is considered in the defence domain, therefore showing why further research into the specifics of containerization is valuable. The chapter continues with a description of a typical on-premise mission-critical system, which serves as the As-Is situation for this research. After the As-Is situation, the trade-offs which were made during the research have been described. These trade-offs were used in the requirements and reference architectures creation, the trade-offs are rooted in business requirements which are specific to the defence domain. The chapter ends with the requirements and assumptions for the To-Be situation. Most information used in this chapter was gathered through interviews with domain experts. Information on implementing container technology in other business domains was gathered through literature research. This chapter describes research activities conducted in the Relevance cycle.

3.1 Interviews

The conducted interviews were semi-structured, and the questions asked were adjusted for the expertise of the interviewee. This way the interviews gave insight into the business considerations for implementing containerization and related container technologies in the defence domain, as well as technical details on the current system and the To-Be situation. In total four employees with different expertise were interviewed (two Software Architects, a Cloud Architect and a Product owner). During the interviews, written notes were taken and no recording was made, due to the confidentiality of the discussed subject. The notes were processed into a comprehensive report after the interviews. This report was sent to the interviewee with the request to review the answers on how accurately these reflected their thoughts and opinions. This process is referred to as member checking. Afterwards the answers were adjusted according to the received feedback, and these answers were then used to preform the analysis for the business considerations, the current situation and for the future situation. The reports have not been included in this paper due to the confidential nature of the content.

3.2 Containerization in Business

Several investigations have been conducted on containerization in varying business and technical domains, with new investigations being published regularly, due to the technology's growing popularity. Goldschmidt and Hauck-Stattelmann created an architecture for industrial Programmable Logic Controllers using concepts from container solutions [55]. By using these concepts on a different technical domain, they showed the flexibility of the concepts of containerization for complex systems. Kho Lin et al. have successfully containerized a training application for the defence domain and deployed the containers with K8s in the public cloud [56]. Kugele et al. addressed the challenge of managing the complex modern automotive electric architecture through the use of containerization, showing how small scale, on-premise systems can benefit from containerization [3]. The containers in this system were not managed by an orchestration solution. Morris et al. showed the effectiveness of containers for the deployment of astronomy software, which demonstrates that containers can be utilized to support high-performance workloads [57]. Dragoni et al. migrated an on-premise, mission-critical monolith to a microservices architecture. The researchers made use of containerization, using Docker Swarm to orchestrate the containers, focusing on scalability and availability [4]. The researchers concluded that the

proposed microservices architecture did solve several problems the monolith was facing, including hardware and OS-dependency of the applications.

The K8s website shows at least 70 case-studies [58], including cases from high profile companies (e.g., eBay, SAP and Philips), describing containerized systems deployed by organizations to increase business value. This shows the versatility of the container/K8s stack for many industries. Most of these examples use the public cloud, or larger on-premise private clouds, to implement containerization and K8s. An example of these larger on-premise deployments is the system used by CERN (the European Organization for Nuclear Research) [59]. This system is used to process large amounts of data, with the possibility to 'burst' workloads to the public cloud if the computational demand is higher than the on-premise infrastructure can facilitate.

The discussed successful implementations of container technology show that containerization can be used in varying contexts and to tackle several challenges which are associated with monolithic, complex IT systems. Although some studies were found to be related, none fit the specific context in this research well. Kugele et al. containerized on-premise applications in cars but did not use an orchestration method [3]. Dragoni et al. [4] migrated a monolithic, on-premise and mission-critical system, but did this in another domain with less strict requirements (financial banking) and used a different orchestration method, which was already found to be less applicable to the defence domain, this was reported by research previously conducted at a defence organization. Next to this, the system had access to more installed hardware, compared to the on-premise context considered in this research. The lack of documented systems which do fit the context suggests that the artefact created in this research is not a routine design and will therefore contribute to the knowledge base, which is a requirement for Design Science, according to Hevner et al. [6].

3.3 Specific Challenges for the Defence Domain

The interviewees mentioned that there are mission-critical systems in the defence sector currently on offer that are delivered as one complete system (a monolith). This research considers a typical example of a mission-critical system, the challenges encountered regarding this system are deemed representable for challenges with mission-critical on-premise monoliths in the defence sector. The monolith renders the process of creating and delivering small incremental changes to the product impossible, since this requires a complete system recertification because of the nature of the product. The modern world has brought accelerated changes to the defence industry, which results in faster changing requirements. The current IT infrastructure is not able to adequately support a deployment model (e.g., Microservices Architecture) which fits the fast-changing requirements. It is possible that the delivered system could fit its intended purpose better if the adaptability is enhanced. According to agile software development principles [60], which has been the standard for developing software for well over a decade now [61], potential customer value (and in turn business value) is lost by not being able to easily deliver smaller, incremental changes. Not having to recertify the complete system saves time and effort in the testing and certification processes. These time savings can directly translate to lower overall development costs and shorter software delivery cycles. Whether this is possible in the defence domain is dependent on specific contractual agreements, therefore this will not be (fully) applicable to all projects.

The system under study in this research contains hardware and OS-dependent applications. This dependency decreases the maintainability of the system, which leads to increased costs for maintaining the applications. Next to this, this dependency can lead to the complete system being required to run a relatively older OS, which does not offer the same functionality as more modern OSs. Therefore, it inhibits the innovation of applications which could benefit from the functionality offered by more

modern OSs. Technology focused companies create a lot of business value from their innovative abilities, supporting these abilities results in creating more business value.

The business challenges of having to recertify the complete system and the added maintenance costs of the hardware and software dependent applications can be addressed by containerization. As described in Section 2.2.1, containerization allows for host independency of the applications [4] with comparable performance to running applications on bare-metal [20]. The isolation offered by containerization allows applications to be deployed on the available hardware while having no or limited impact on each other.

Although containerization addresses the aforementioned challenges, implementing containerization introduces a new set of challenges as well. One of these challenges is achieving HA with containerized applications through orchestration, which is the focus of this research. Mission-critical systems must be highly available, since this is a hard requirement posed by the clients. As mentioned in Section 2.3, only a small number of investigations have been dedicated towards K8s availability mechanisms. No implementations of K8s could be found in similar on-premise systems, and this is not surprising, given that mission-critical systems in the defence sector often have the hard requirement of confidentiality as well.

3.4 Representation of As-Is Situation

In this section an As-Is representation of a typical on-premise, mission-critical infrastructure is given. The purpose of this architecture is not to create a comprehensive architecture of these kinds of systems, since such architectures would be too complicated and extensive for the purpose of this research. Only the information necessary for creating an overview that can be used to compare the To-Be situation with the As-Is situation has been included in this report. This section combines the information from the interview with some documentation on the current system.

3.4.1 Hardware in the As-Is Architecture

The systems offered to customers consist of different configurations depending on a customer specific set of sensors and actuators, and the customer's wishes. This does not have impact on the basic infrastructure since it can be assumed that the basis for the systems is the same. The most relevant components in the system for this research are the ECCs (Environment Controlled Cabinet) and the MOCs (Multifunctional Operator Consoles). ECCs are filled with a varying number of servers, and these servers host parts of the system. Generally, two ECCs are installed on-premise in different equipment rooms, where these equipment rooms can be regarded as AZs, as described in Section 2.1.3. MOCs are Fat Clients installed in the Combat Information Centre (CIC). The MOCs are used by personnel to interact with the various applications that run on the MOCs. Most application processing is currently done on the MOCs. The delivered systems vary between a single MOC and no ECC to tenths of MOCs and two ECCs. For this research, an environment with two ECCs and an arbitrary number of MOCs is considered. The MOCs and ECCs are connected through redundant communication paths and are connected with a customer-specific set of sensors and actuators through these communication paths.

3.4.2 Software in the As-Is Architecture

The current system consists of several software components, most of these components are managed by Logical Subsystems (LSS), which handles the start-up and management of the applications within the systems. LSS will be replaced in the To-Be situation by K8s and is included in the As-Is architecture to present how the To-Be situation differs from the As-Is situation. Applications intended for use on the MOCs can run on any of the available MOCs, the applications are loaded based on user authorization and the choice made by the operator. Most applications are stateless and save data on a Data Distribution

Service (DDS). The specific implementation of DDS used provides persistence of the distributed data, this way, applications have access to the required data. DDS is a middleware tool which allows for publish/subscribe communication between application as well, but in this research, we focus on the distributed data storage functionalities it provides. The DDS instances are monitored by a dedicated monitoring application, so that if the local daemon fails, this application forces a reset of this instance. Next to this, the DDS instances monitor other DDS instances. This creates a cluster of instances which monitor each other. DDS allows the applications to be started on a different machine without the loss of data, since it allows for data to be available on all the machines running DDS. Important applications are deployed redundantly to ensure availability. The system is monitored through a monitoring application, which keeps track of the status of each node in the system. This monitoring application is installed on every node. If a node fails, the monitoring application sends a notification to LSS. LSS contains configurations for different failure scenarios and starts a new application or adjusts the applications running on its node according to the notification message sent by the monitoring application.

Figure 9 shows an overview of a system deployment. The connection between the different components is redundant but represented as one connection to keep the figure simple. Each MOC and each server in the ECCs run an arbitrary number of applications, which are specific to either ECC or MOC environments. The MOCs and ECCs are connected to several sensors and actuators, and the applications communicate with these sensors and actuators. These applications have been kept general, since the specific details of these applications are not important for the reference architecture. The OS has not been specified, since it does not have impact on the reference architecture. Previous internal research at an organization active in the defence domain has found that K8s can run on a set of Linux distributions which are applicable in the domain. Functioning of K8s on these different OSs is essentially identical and does not impact the reference architecture in this research.

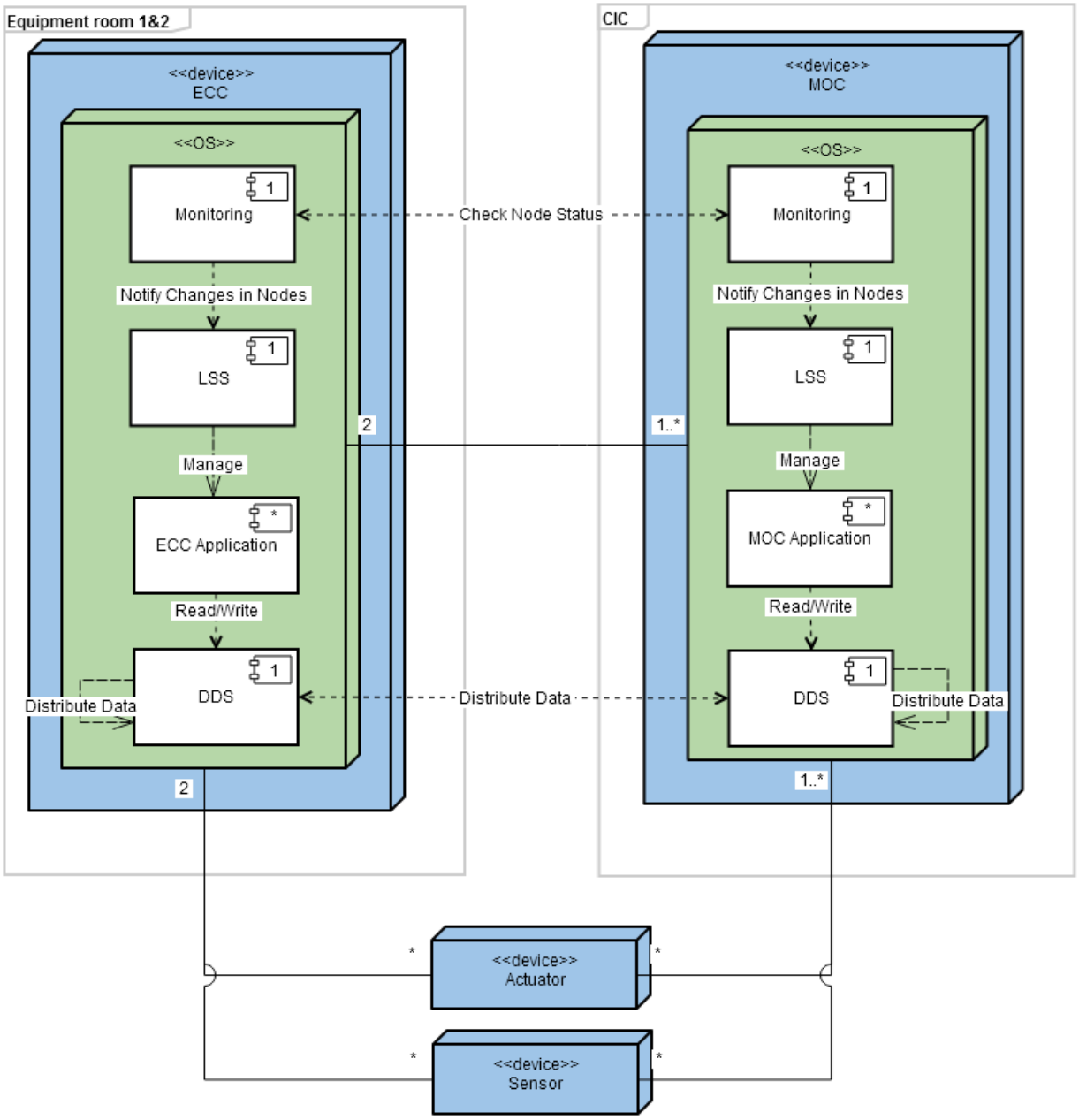

Figure 9: As-Is Architecture

3.5 Trade-offs

Several trade-offs have been identified in the process of creating the reference architectures. Not all these trade-offs were addressed in this research. Some of these trade-offs have already been considered in previous (internal) research and were mentioned by the interviewees, and some of the trade-offs are out of the scope of this research. These aspects have been mentioned in order to show these trade-offs and the relevant factors in these trade-offs. The results of this analysis can provide a starting point for further research on these trade-offs, the trade-offs that fall in the scope of this research are considered in the reference designs.

3.5.1 In-Scope Trade-Offs

One of the trade-offs is the timeout for scheduling a new pod on a functioning node when a node fails. This trade-off is tightly coupled with availability and will therefore be considered in greater detail. Increasing the amount of heartbeats sent by each component (in other words, lowering the monitoring interval) increases the overhead on the both the network and the available hardware. Kanso et al. found out that lowering the monitoring interval to its lowest configuration does not substantially increase resource utilization (memory & CPU), although the research does not consider network overhead [29]. Configuring K8s to react to potential failures very fast (two seconds) can lead to a false-positive in failure detection due to a network hiccup. These false positives can lead to an instable system. During the design process of our research, we assumed that high quality networking equipment is installed on-premise. This equipment should be able to support the network overhead of the higher amount of heartbeats and should reduce the chance of a network hiccup. An appropriate heartbeat interval and K8s reaction configuration has been chosen in the design of the reference architectures.

Making the system more redundant also increases the overhead by deploying an increased number of components. In the case of a multicluster system, this means an added number of management components (K8s master nodes and storage back-end nodes), which all have claims on the available hardware. These components are critical for the correct functioning of the cluster and should therefore have access to the computing power without having to compete with other workloads. Increasing the amount of hardware installed on-premise is an option, but this would also increase costs, which is not beneficial for the customer. It also introduces several other challenges including increased space requirements, which complicates the design process of the on-premise context. Next to this, adding redundant components also increases complexity and makes system management more challenging. Therefore, a sensible trade-off between the redundancy and the amount of hardware that can be dedicated towards this goal has been made during the evaluation of potential architectures.

Some of the considered open-source technologies have a low maturity and relatively low community support, but these technologies could potentially be used to address the availability challenges encountered when implementing container orchestration. Therefore, a trade-off has been sought on the (potential) functionality, the maturity and the likelihood of increased community support of each technology. In case the potential functionality of a specific technology is deemed to be highly valuable but its maturity and community support are low, we can recommend a system using this technology, with the added limitation that contribution to these technologies is needed in order to be applicable to the domain.

3.5.2 Out-Of-Scope Trade-Offs

Another trade-off considered for containerizing the applications is the performance impact. As mentioned before, containers do not need a full OS compared to VMs. Because of this, a considerably lower overhead and better performance is achieved [16, 17]. Mazehari et al. found out that containers

offer near bare-metal performance [20]. We assumed that the advantages of containerization for development and maintaining mission-critical systems (as discussed in Section 3.3) compensate for the small performance hit. Actual performance impact is not measured in this research.

Due to containers being deployed on shared hosts and being less isolated compared to VMs an inherent security problem arises. A number of studies addressed this issue, and with proper management a lot of challenges can be resolved [62, 63]. Another way in which isolation can be enhanced is by having multiple security levels. In the case of clusters, this would mean multiple clusters. For example, one cluster could contain all the high-security level application and one cluster the lower-security levels of applications. This vertical multicluster setup is not considered in this research, since it does not influence availability. The results of this research can be used to implement HA clusters on both security levels, essentially having two independent setups, would the security specifications require this.

Next to this, a trade-off for the latency needs to be considered. Since latency is also an important aspect for mission-critical systems. The added latency of introducing containers and K8s is largely dependent on the technology stack selected. Different networking plug-ins and service-mesh solutions offer different functionalities and have different associated latency impact. Because this requires more investigation, this trade-off is not considered in this research.

3.6 To-Be Situation

This section describes the requirements for the To-Be situation, as well as the assumptions made during the design of the reference architectures. The information needed for the requirements and assumptions was obtained through a combination of the interview results and consultation of internal documentation. The requirements have been verified by an expert as being appropriate for the mission-critical system and the focus of this research. The requirements are summarized in Table 6.

3.6.1 Requirements

The new system must not contain any SPOF.

No SPOF can be present in the system, therefore every part of the system has to be redundant. The redundant components have to be distributed across the available AZs as to not render an AZ a SPOF. Software using quorum cannot have a majority of its members deployed in one AZ, otherwise this AZ will contain a SPOF. Quorum cannot be used if an even amount of AZs is available due to this requirement.

The new system should make use of K8s.

K8s has become the de facto standard for container orchestration, since many high-profile companies use this technology (see Section 3.2). The experts indicated that K8s is the most promising technology for the purpose of container orchestration in the domain. Therefore, the reference architectures should make use of K8s as a basis for orchestration of containers.

Critical parts of the system must be failed over in case of a failure within thirty seconds.

Failover of critical applications may under no circumstances take longer than 30 seconds. The aim is to reduce this further. For example, pod failures should be completely resolved in four seconds and node failures in ten seconds. Some applications are too critical for this and are designed in a specific redundant way. These applications have zero downtime, as long as one instance is running.

The system should be technology agnostic (apart from K8s).

The system should not depend on specific technologies, and the used technologies should be interchangeable for other similar technologies. This avoids vendor lock-in and ensures the technology stack can be maintained for long periods of time (>30 years), as is common in mission-critical systems. The reference architectures' diagrams include both the more abstract term for the role of the component as well as an example of a popular software product that offers the required functions, in order to show what each component is supposed to do.

The new system should monitor the applications and components in a short interval without being susceptible to false positives.

Lowering the monitoring intervals and the number of times a component is allowed to fail delivering a heartbeat leads to higher availability, because failures are detected earlier. Lowering the intervals too much can lead to false positives, which can eventually lead to components wrongly being excluded if a network hiccup occurs or if K8s is not able to process all heartbeats, which reduces service availability. The interval times have been defined in such a way that these intervals can satisfy the allowed failover time without introducing a considerable risk of false positives.

The utilized open-source technologies must have a license which allows for classified usage in mission-critical systems.

Some open-source licenses require developers to open-source their products that are derivative work of another technology. This is not an option for defence systems, due to the nature of the products, the customers and the intended uses of these systems. Therefore, the licenses under which technologies are open-sourced have been checked on compatibility with confidentiality requirements typical to the defence domain.

Nr.	Requirements
Req1	The new system must not contain any SPOF.
Req2	The new system should make use of K8s.
Req3	Critical parts of the system must be failed over in case of a failure within thirty seconds.
Req4	The system should be technology agnostic (apart from K8s).
Req5	The new system should monitor the applications and components in a short interval without being susceptible to false-positives due to network hiccups.
Req6	The utilized open-source technologies must have a license which allows for classified usage in mission-critical systems.

Table 6: Requirements for Reference Architectures

3.6.2 Assumptions

Several assumptions were made during the definition of the reference architectures. These assumptions have been summarized in Table 7.

K8s master nodes can be deployed across the servers on the two ECCs and an arbitrary third computing instance.

Introducing K8s and Cloud Native principles into mission-critical on-premise systems gives vendors the opportunity (and in some cases forces the vendors) to rethink the current hardware setups. Currently MOCs are Fat Clients that could host a third master node alongside the other workloads it has to process,

but this can change in the future. Several more computing instances that could host a third master node are distributed across the on-premise context. Since the choice for hardware changes is dependent on several different factors outside the scope of this research, we abstract from where this third master node should reside. For a some of the reference architecture we assumed that an arbitrary third computing node is available. In the actual system this third computing node could either be a MOC installed in the CIC or another computing unit installed in a different AZ. Other reference architectures have been dedicated to achieving HA with only two AZs. Using only two AZs increases the flexibility of the design and requires less space in a context where physical space is limited and valuable. Hence, it would be beneficial to find K8s deployments which achieve HA with only two AZs.

Organizations are willing to contribute to the open-source community.

Vendors of mission-critical systems are generally high-tech organizations with a lot of technical knowledge. These organizations are often willing to contribute to open source-projects, next to using open-source technology to strengthen their products and business offerings. In case a technology (or the concept behind the technology) presented in the architectures is considered of a high potential value, whilst the maturity is low, we can advise to actively contribute to this technology, in order to speed up the maturity and render the technology usable for the defence domain.

The current storage solution will be replaced by a quorum based alternative if possible.

As mentioned in Section 3.4, an important factor in achieving HA currently is programming applications stateless. This way, applications can be easily scaled and restarted on the available nodes. Although the applications are stateless, these still need access to consistent data in order to function correctly. To achieve this, several storage solutions can be used. Currently an implementation of DDS is used but depending on the future implementation of the systems another kind of distributed data technology can be used. Usually, distributed storage in K8s is managed in a similar way to how etcd is managed, using a quorum algorithm and replication of data across all instances to achieve HA. Managing state and storage in K8s clusters can be a challenging task and finding the right technology and implementation is a large subject. Therefore, a complete analysis of the storage solutions is not made in this research, and the storage solutions proposed are presented as indicative of the possibilities for each architecture. In complex systems oftentimes a message-oriented middleware (MOM) or similar middleware for communication between services is deployed. Whether or not this is needed in a K8s environment is beyond the scope of this research, since this subject is further complicated by the fact that service mesh solutions (e.g., Istio [64]) are often used in combination with K8s. Service mesh solutions allow for better management of communication between microservices. Whether or not a middleware is still needed in combination with a service mesh solution is out of scope of this research.

For the purpose of this research it is assumed that quorum based storage is preferred when possible (i.e., when three AZs are available). In case this is not possible (i.e., when two AZs are available), it is assumed that the current combination of DDS and storage solution is used.

The quorum based storage solution included in the reference architectures is GlusterFS [65], since previous internal research has found this solution to be the most appropriate distributed storage solution for mission-critical systems. GlusterFS is a distributed file system that can distribute data across large clusters. Several operating modes for GlusterFS exist, but for the purpose of this research only the replicated method is considered. In this method, data is replicated across all available nodes containing a GlusterFS instance, which ensures availability of the data if one of the instances fails. GlusterFS requires

three nodes to work in HA. A two node cluster is not sufficient since it cannot operate as a single instance in a two node cluster, due to a quorum being required. GlusterFS does support a quorum device (as described in Section 2.1.5) in the form of an arbiter [66], making it easier to achieve quorum with limited hardware. GlusterFS can be deployed and managed by K8s as a pod by using Heketi [67]. Heketi is a management interface for GlusterFS that allows cloud services (like K8s) to manage GlusterFS instances. For the purpose of this research GlusterFS has been chosen as the storage solution, but other storage solutions could have been chosen (e.g., Ceph(FS) [68]), this includes the current DDS and storage solution.

The complete system will use K8s.

To decrease complexity and increase maintainability of the mission-critical systems offered, vendors aim to standardize most of the utilized technologies. We assumed that the design should be able to support the complete mission-critical system. Since all (stateless) applications will be containerized and must be managed, no specific subset of applications is considered. Introducing subsets of clusters for specific sets of applications is deemed to be unviable because of the amount of overhead it introduces, especially when considering that these specific subsets also have to be deployed redundantly. A uniform cluster design is therefore preferred. K8s offers some functionality that allows applications to be grouped. If it is deemed necessary, more systems using the same architecture could be deployed for different security domains. This is out of the scope of the research, but shows that our reference architectures can serve as a basis, where further configuration can be done in order to achieve certain characteristics.

Load balancing and its associated failover are done through tested and proven technologies, of which several mature solutions exist.

Load balancing has been a widely used concept within server infrastructures for a long time now. For example, HAProxy was released in 2001 [69] and Nginx was released in 2004 [70]. Load balancing is required in a multi-master K8s setup, because the worker nodes need to be able to reach all master nodes and redirect traffic to a functioning master node if one master node has failed. This research did not measure the reaction times of the load balancers. Similarly, the reaction times of IP failover software is not tested either. Keepalived was released in 2000 [71] and is used by Red Hat [72] in their offered products, which indicates that the technology is stable and has received support. IP failover software is needed for multiple load balancers to work in an active/passive mode. If no multiple load balancers are deployed, this load balancer becomes a SPOF. Reported failover times that are available, both publicly [73, 74] and in literature [75, 76, 77, 78], are not tested again in this research, because of the wide-spread usage and the aforementioned maturity giving credibility and validity to these reports.

Deployed, active K8s clusters in the defence domain are immutable.

Rolling updates are common for a lot of businesses, which means that some replicated applications are updated to a new version, whereas other replications are still on the older version. This way the replications running the older version are still redundantly available (two instances are always online) and able to service requests in case the newly deployed version encounters some unexpected error. In the defence domain the system is redeployed during set maintenance intervals and not during operation. Therefore, applications and cluster components do not need to be replicated to facilitate this behaviour. We assume that the configuration is given during such a maintenance interval and that this configuration is tested and works, but between maintenance intervals the configuration is not changed.

Possible configuration errors are left out of scope in this research. In other words, the system is considered to be immutable during operation.

Nr.	Assumption
A1	K8s master nodes can be deployed across the servers on the Two ECC and an arbitrary third computing instance.
A2	Organizations are willing to contribute to the open-source community.
A3	The current storage solution will be replaced by a quorum based alternative if possible.
A4	The complete system will use K8s.
A5	Load balancing and its associated failover are done through tested and proven technologies, of which several mature solutions exist.
A6	Deployed, active K8s clusters in the defence domain are immutable.

Table 7: Assumptions for the Reference Architectures

4 Designing the Reference Architectures

This chapter describes the designed reference architectures that fulfil the specified requirements. The architectures are represented by using diagrams and have been designed iteratively. Feedback was provided by a domain expert during these iterations. This feedback has been used to clarify the diagrams and adjust the architectures to better fit the domain. The diagrams were abstracted from details to keep the architectures readable. Section 2.2.2 can be consulted for detailed information on the details of K8s components. The diagrams representing the architectures are according to the agile principle Just Barely Good Enough [79] to convey the most important aspects of each architecture without cluttering the diagram.

It is common practice in the Cloud Native community to represent architectures using ad hoc notations. In order to make this research reproducible and make its results understandable for a large audience, a systematic standard for representing the architectures has been used. The modelling language chosen is UML [30], specifically the deployment diagrams. These diagrams show the relation between hardware and software components, which is useful for this research. The relations between software components have also been given for clarity, although this is not strictly according to the UML standard for deployment diagrams. Otherwise, multiple diagrams for each architecture had to be made, which could complicate understanding the reference architectures, whereas this small addition of relations between software components voids the need for these extra diagrams. These relations are denoted as dotted lines in the diagrams.

This chapter starts with a description of the specific challenges mentioned by interviewees for implementing K8s in the research context, and how these challenges are addressed in this research. After this the five designed reference architectures are discussed. The chapter ends with the selection of the most promising reference architectures. This selection was based on questionnaire results conducted with several domain experts. This chapter describes research activities conducted in the Design cycle, namely the creation and evaluation of the reference architectures.

4.1 Challenges

The main challenge that was mentioned during the interviews is achieving quorum in the on-premise context due to amount of available Availability Zones (AZs), since a quorum requires at least three AZs and the current on-premise environment consists of two AZs. Although it was assumed in this research that a third AZ could be incorporated in the on-premise context, it was still necessary to define reference architectures to be deployed across two AZs, in order to make an appropriate comparison between the (dis)advantages of the different designs for the on-premise context. As mentioned before, due to the physical space available and constraints of the on-premise context, it could be beneficial to deploy the system across two AZs. To address this problem, the interviewees mentioned that federation could help alleviate this challenge, by increasing redundancy by adding multiple clusters to the overall system.

Another challenge mentioned during the interviews is the speed of K8s reactions to failure. In case of a pod or node failure, K8s should failover the affected applications, but it does this after a certain timeout. The default timeout is too long for the failover requirements of mission-critical systems. Lowering this timeout is possible, but this increases the chance of a false-positive for marking the pod or node as failed. Not marking a failed node in a timely manner leads to an overall performance loss and reduced service availability, since the failed pod will continue to receive requests, whereas false positives in marking a node or pod failed also leads to performance loss and reduced service availability. Section 2.3 discussed the research done on K8s reaction times. These investigations found that K8s has a mean

reaction time of 4.65 seconds in case of a node failure. This is partly due to the node being allowed to fail four consecutive status updates. The amount of allowed failed consecutive updates can be lowered. For the reference architectures we assume that this is lowered to two consecutive status updates. Therefore, we assume a theorized mean reaction time of around 2.65 for the reference architectures. The first reaction of K8s after a worker node failure will be to mark the node as *NotReady*. After this, the pods on this node will not receive new requests. Due to a default timeout, the pods on the failed node are evicted 40 seconds after the node is marked *NotReady*. After this, new pods are created on a functioning node. This default timeout is too long for mission-critical applications, so we assume for this research that the timeout is reduced to two seconds, which we deem to be a safe and sensible threshold regarding possible false-positives. Due to the high-quality network equipment used in the on-premise context, it is safe to assume that network hiccups and other network performance problems are unlikely. Therefore, a low timeout and monitoring interval can be used. The total duration from failure to creation of new pods would then be around five seconds, the failed pod will not receive any requests after around 2.65 seconds, and new requests will be serviced by deployed redundant pods after this period. Pod failures have a lower mean detection time and are repaired by the worker node in which the pod resides, this is generally done within a second.

Creation of new pods takes time, and the K8s community has described Service Level Objectives (SLOs) to which these start-up times have to comply [80]. The SLO states that 99% of the pods should be created and watchable within five seconds, given that the container images are pre-pulled and that the pods are stateless. This is tested on standard K8s clusters with up to 5000 nodes, with 150000 pods deployed to these nodes [81]. This is deemed sufficient for the mission-critical, on-premise use-case. The SLO does not consider the actual application start-up times. According to the K8s maintainers this is very specific to each application. Therefore, it will not be taken into the scope of this research. Next to this, some applications in the mission-critical system are of utmost importance, so they are programmed in such a way that a workload will finish as long as one of the redundant instances is active. For these applications, it is more important for K8s to quickly notice a pod is not active so requests will not be sent to this pod than starting a new pod within seconds. No SLO has been drafted for stateful pods yet. Given that we have theorized that the detection and eviction time combined is around five seconds (2.65 seconds for detection and two seconds for eviction), and 99% of pod creation times takes less than five seconds [80], we assumed a total of ten seconds from failure to the new pod functioning in case of a node failure.

Another component of which the reaction times should be assessed is the load balancer (in this research HAProxy [69]), which is deployed between the master and the worker nodes. In case one of the master nodes fails, the load balancer should be able to redirect traffic to a functioning master node within a short amount of time, which can be specified to be within seconds. Load balancers can be deployed redundantly by deploying multiple (often two) instances of the load balancer. These load balancers share the same virtual IP through an application like Keepalived [82]. One load balancer acts as the master, the other load balancer acts as a standby. In case the master load balancer fails, Keepalived will switch over the IP to the backup load balancer within a short amount of time. Usually, Keepalived is configured to check the process it monitors every two seconds [83, 84, 85]. Given that Keepalived only has to failover a virtual IP and (depending on the application) run a script which promotes the standby to master, a failover time of less than five seconds can be assumed for these components. The reference architectures have been designed to be technologically agnostic. Keepalived and HAProxy are mentioned here as examples of technologies which could be used as a redundant load balancer, technologies with similar functionality exist. The failover software (Keepalived) is also used in case additional monitoring is

needed for specific applications (mainly in two node data storage situations), where again a failover time of five seconds can be assumed, since Keepalived only has to failover the virtual IP and run a relatively uncomplicated script.

In two AZs setups, a possibility of split-brain arises, as described in Section 2.1.5. A split-brain situation would have a lot of impact on the backend storage for the cluster, since incompatible data objects would lead to significant problems. Master nodes can both assume leadership, but if the backend storage does not have split-brain issues, the impact on data integrity and consistency is deemed low. Since the storage backend will only accept write requests sequentially, data objects cannot be written to simultaneously. Therefore, potential split-brain issues in the master nodes are regarded as less impactful. In case of a two-node storage solution using Keepalived to configure a virtual IP, a split-brain situation would result in both nodes claiming the virtual IP and assuming master status. To reduce this chance, the Virtual Redundancy Router Protocol (VRRP), that is used by Keepalived to verify the health of other members can be configured to use the same network interface as the virtual IP. This reduces the chance that both instances claim the virtual IP due to network outage. Since, if Keepalived can claim the virtual IP over a network interface, it is also very likely capable of sending and receiving VRRP packets over the same network interface. Next to this, the redundant network paths installed reduces the likelihood of a split-brain situation considerably. If these measures are not deemed enough to prevent split-brain, an implementation of STONITH or fencing (Section 2.1.5) can also be applied. Implementation of these mechanisms has been left out of the scope of this research, since it is not clear whether the chance of a split-brain occurring is high enough to be included. During the interviews, domain experts mentioned that the redundant network paths currently used in the on-premise context render a split-brain situation highly unlikely. We assume that the described precautions and the use of redundant network paths render the chance of a split-brain scenario negligible.

4.2 Reference Architectures

This section discusses information applicable to all five reference architectures. We assumed that each of the architectures use the configurations described in Section 4.1. We also assumed that the failover times as mentioned in that section are applicable to all the reference architectures, since K8s is a scalable technology and failover mechanisms used by K8s should work in a very similar way for each architecture. Actual prototypes must be created to confirm the assumed failover times. The architectures are represented by a diagram.

Each of the presented architectures contain Environment Controlled Cabinets (ECCs) which are filled with servers running both master and worker nodes. In the diagram the worker nodes are denoted as one worker node per ECC. In the actual real-world implementation, this can be a multitude of worker nodes, depending on the number of servers available in each ECC. Usually, in UML deployment diagram model redundant components which are identical instances are denoted as one entity (type), with a number denoting the number of instances of the specific type that are present. Combining the instances has deliberately not been done in our diagrams, since some of the architectures have slight differences in the components deployed, for example, ECC1 contains different software to ECC2. Consequently, this has also been done for architectures where ECCs do contain identical components, to make the architectures more comparable. Otherwise, the architectures with identical deployments on the ECCs will appear considerably less complex, due to these architectures containing almost half the visual elements, while it consists of a similar number of components. All physical communication paths are represented as a single connection between the physical components to retain readability. Each architecture is accompanied by an overview of the minimum amount of system resources that should be dedicated towards each architecture. The presented minimum amount of system requirements allows

for a comparison between the expected system requirements for each architecture, but these requirements are the actual bare-minimum. For an actual deployment more system resources will have to be dedicated to cluster [86]. The included <<executionEnvironment>> in the diagram can be bare-metal machines or virtual machines, but this choice has no significant impact on the architectures in this research. Each of the <<devices>> in the diagrams represents a separate physical component. A storage solution for each option has also been included, but this is not the focus of this research and should therefore only be regarded as a suggestion. The Multifunctional Operator Consoles (MOCs) have been included in the clusters as nodes in a way that these can be removed from the cluster without impacting the fundamentals of the designs, allowing for flexibility to consider what the MOCs role will be in the future. The K8s control plane (master nodes) will be considered HA when two master nodes are deployed. As described in Section 2.2.3, most HA K8s deployments contain three master nodes, but since rolling updates will not be done during operation, as described in Section 3.6.2, a third master node is not necessary.

The architectures have been created with two variables, namely the amount of AZs available (two or three) and whether federation should be used. These variables were focused on after consultation with the domain experts, who indicated that federation could be a solution to the availability challenge. The domain experts also indicated that the on-premise context currently consists of two AZs, although redesigning the on-premise context is possible a viable solution to achieve HA using two AZs is preferred.

The presented reference architectures should support running the applications currently managed by the infrastructure described in the As-Is architecture (Section 3.4). The current As-Is system will (largely) be replaced by a K8s-based system. To achieve this a number of steps have to be taken, since the applications have to be containerized and additional aspects like inter-pod communication and security have to be considered. The presented reference architectures provide insight in how to address one of the challenges encountered with K8s (namely availability). The other parts (e.g., security, communication) also have to be added to the presented architectures, in order for these architectures to become a suitable replacement for the As-Is system.

4.3 RA1: Single Cluster (Three AZs)

The first reference architecture considered is a single cluster using three etcd nodes. A representation of this reference architecture is given by Figure 10. Two of the etcd instances reside on the ECCs. The third etcd instance is placed on a third node, called the quorum node, in a third AZ. This third AZ is present in the on-premise context in the hypothetical situation that the on-premise context is redesigned.

The components that require a quorum (Distributed Storage, etcd) in this system have been distributed across the two ECCs and the quorum node. The Distributed Storage instance in the quorum node consists of a quorum device (in GlusterFS' terms, an arbiter), which is not run as a K8s pod. An arbiter requires considerably less resources than having a full additional member. To run this arbiter as a pod, the quorum node should also host a K8s node, which would increase the overhead. Next to this, the quorum node does not contain a third K8s master node, since the master components do not use quorum for leader election but use a lease system which is described in more detail in Section 2.2.2. Keeping the overhead low allows the quorum node to be a smaller and less powerful computing device. This adds flexibility to the implementation of the system, since it would simplify the redesign of the on-premise context, because a smaller device is easier to fit in the limited physical space.

This setup is similar to a standard single cluster HA setup as described in Section 2.2.3, which makes the setup relatively simple for creating a stable cluster.

Theoretical minimal requirements

2x master node = 4x (v)CPU, 4GB RAM [87]

3x etcd nodes = 6x (v)CPU, 12GB RAM [88, 89]

2x load balancer = 2x (v)CPU, 2GB RAM [90]

Total: 12 (v)CPU, 18GB RAM

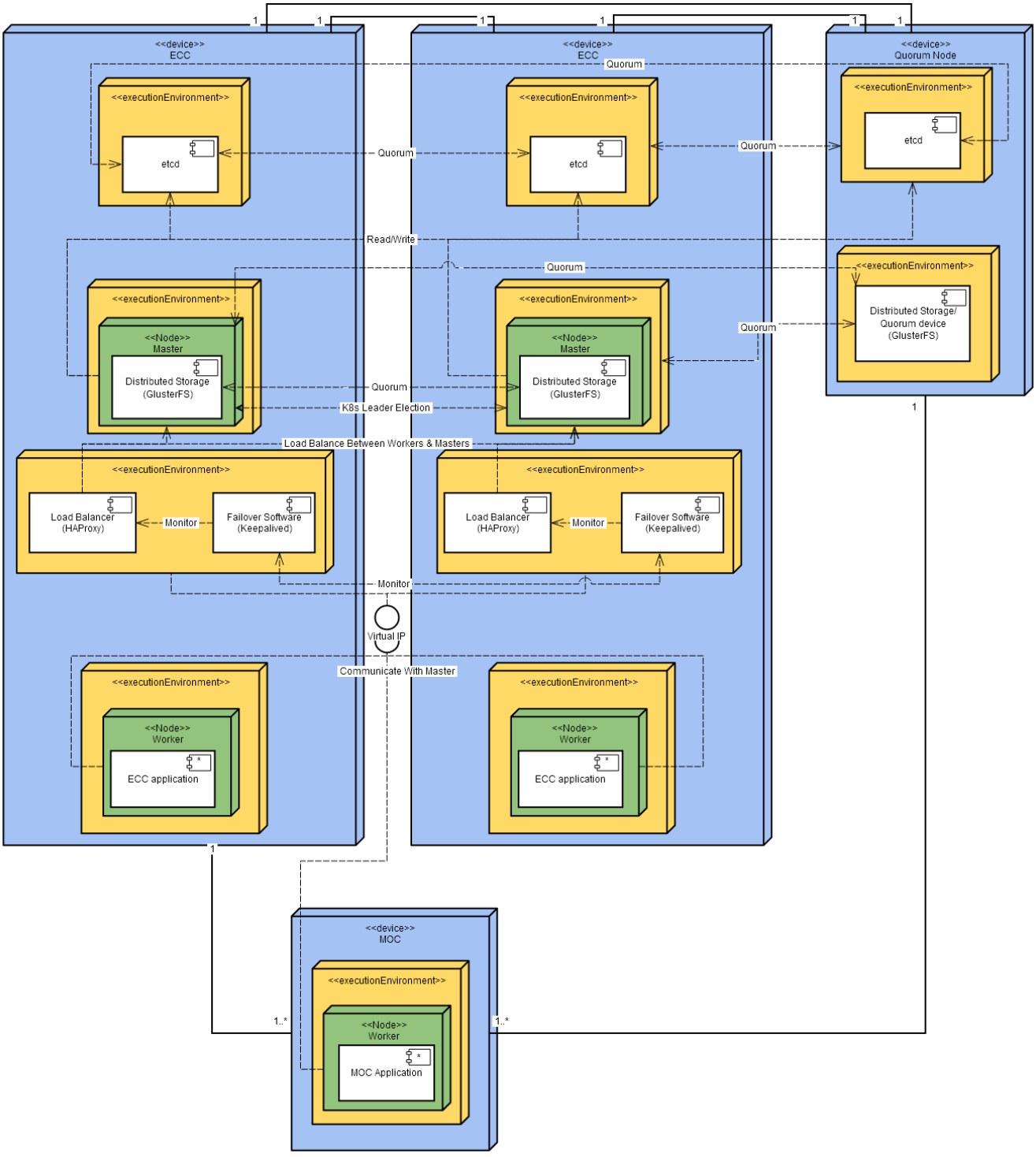

Figure 10: Single Cluster Three AZs

4.4 RA2: Single Cluster (Two AZs, Kine)

This architecture does not make use of the standard Key-Value store for K8s (etcd), but uses software called Kine [91]. Kine was found after a suggestion from a domain expert. Kine is an etcd-shim. A shim intercepts API calls and translates the contents to an API call which is compatible with another environment or software product. Kine translates the read/write commands the K8s-apiserver would normally send to etcd to read/write commands for SQL-based storage solutions. PostgreSQL [92], MySQL [93] and SQLite [94] are currently supported by Kine. The Special Interest Group for the K8s-apiserver has shown interest in adding this to their apiserver-builder [95], which would allow for a custom apiserver to easily utilize Kine. This could lead to Kine being incorporated into upstream K8s, meaning that it will be supported by the large K8s community. This opens a number of possibilities, since SQL-based storage solutions are deployed differently than quorum based storage solutions like etcd, since MySQL and PostgreSQL do not need a quorum and allow for a Master/Slave setup [96, 97]. In such setups, the master has read/write access to the data objects, whereas the slave only has read access. Data is synchronized between the master and the slave, and in case of a master failure, the slave can take over service with up-to-date data. There is a chance that a small amount of data is lost if a failure happens after the master has committed a write operation but before the write operation was synchronized with the slave. Since K8s is an intent-based system, we assume that it can easily recover of this small amount of lost data. Some of the SQL-based technologies support synchronous Master/Slave setups. In these setups the write command will only be committed by the Master if it is sure that the Slave has correctly committed the write command issued before. This way, data on the backup are always up-to-date. It does however complicate the deployment and correct management of the SQL-based technologies. We recommend considering such a setup if the risk of losing this data is seen as too high or impactful. The SQL-based storage solutions offer varying levels of failover capabilities as build-in features. For the storage solutions to be interchangeable, a separate health checking/failover application must be deployed. This way, the choice of storage solution is not limited by its build-in load balancing and failover capabilities. This increases the interchangeability of the storage backend of the K8s cluster.

A Master/Slave setup for the storage allows K8s to be deployed across two nodes without containing a SPOF. Figure 11 depicts how such a system could be deployed. The SQL storage instances (in this case PostgreSQL) are monitored by a failover application (Keepalived) and both servers reside behind a virtual IP through Keepalived. In case the Master fails, the Slave is promoted to the new master through a Keepalived script and the virtual IP is assigned to the new Master. The kube-apiserver is shown because this component communicates through the shim with the backend storage, which is different from a standard K8s deployment.

The design does require some different approaches compared to deploying a standard HA cluster. Since only two AZs are present, quorum for other system components can not be achieved. Therefore, GlusterFS has not been included in this model, since it requires at least three nodes. To our knowledge, other distributed filesystems also have this requirement. Therefore, the currently used DDS is this architecture also provides the storage for this architecture, since it does not require quorum and has proven to satisfy the strict availability requirements. DDS does not support being run and managed by K8s (yet) and will require its own dedicated monitoring and solution, like it has now in the current architecture. These additional components have not been modelled to keep the model simple.

Theoretical minimal requirements

2x master node = 4x (v)CPU, 4GB RAM [87]

2x SQL-based Storage server = 4x (v)CPU, 4GB RAM (Only guidelines for the system requirements are given by SQL-based storage vendors, this is an estimation based on the amount of connections it has to serve)

2x load balancer = 2x (v)CPU, 2GB RAM [90]

Total: 10 (v)CPU, 10GB RAM

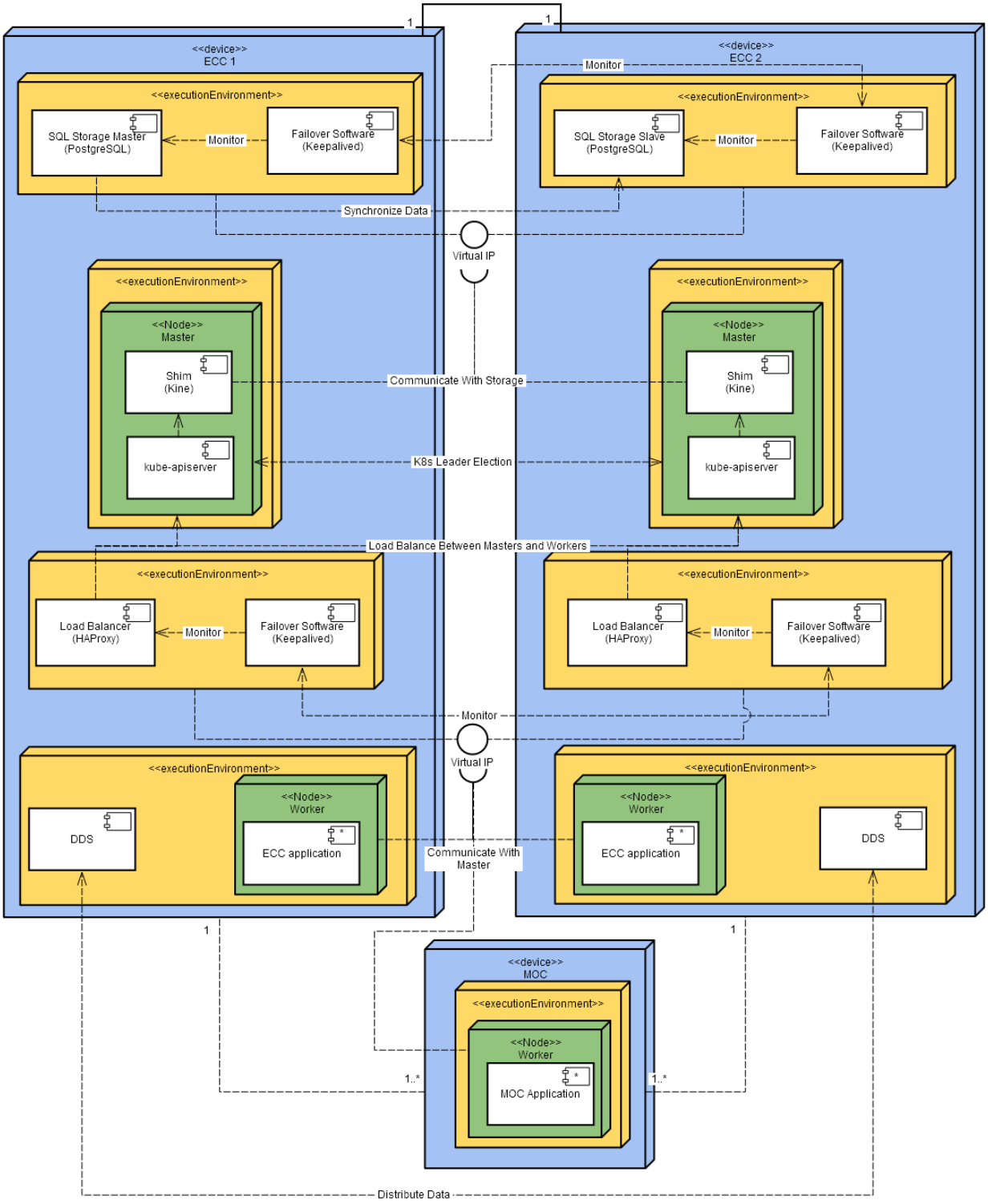

Figure 11: Single Cluster Using Shim

4.5 RA3: Single Cluster (Two AZs, Etcd Learners)

This architecture deploys K8s across two AZs without a SPOF by using a new etcd feature called 'learners' [98]. This feature allows an etcd cluster to have additional non-voting members that do not affect the quorum. The learner was introduced to reduce the strain on the leader of adding a new member, because a lot of data must be transferred from the leader to a newly added member. A learner can receive this information at a reduced speed, so that the leader can service requests while the learner catches up. A learner does not affect the quorum and cannot vote, it is in read-only mode and can help service read requests. Currently the learner has to be manually promoted to become a full member, but in the future, it is planned to support automatic promotion. An option to use learners as hot-standby failovers in case cluster availability is affected is also planned. In the hot-standby mode, the learner will continue to be a learner indefinitely, until a failure in the cluster requires it to become a full member. Figure 12 shows a diagram of the architecture using etcd learners. In this setup an active etcd instance is deployed on each ECC. Each active instance has an associated learner instance, which is constantly kept up-to-date by the leader. In case of failure in one of the servers running the instances, the quorum is lost, since the quorum of a two node cluster is two. As a reaction, the failover standby member is added to the cluster, this creates a new two node quorum across a two (or three, depending on the specific functioning of the failover) node cluster. This automatic promotion feature is not yet implemented in the current version of etcd. The hot standby behaviour is not yet implemented for etcd, the same behaviour can be achieved by adding a failover mechanism like the mechanism described in RA2 (Keepalived). This failover mechanism sends the command to promote the learner in order to restore the quorum.

Like the architecture described in Section 4.4 the two AZs do not provide the possibility of forming a quorum. Therefore, distributed storage systems which use quorum like GlusterFS are not applicable. Therefore, this design also uses DDS for data storage, which is already used in the current system.

One uncertainty of this architecture is that the precise behaviour of the hot-standby etcd learners is not yet known. Discussion within the community should be followed in order to assess the applicability of this method in the future. Contributions towards this feature could speed up the development process of this feature, and tailor the feature more towards the functionality as described in this reference architecture.

Theoretical minimal requirements

2x master node = 4x (v)CPU, 4GB RAM [87]

4x etcd nodes = 8x (v)CPU, 16GB RAM [88, 89]

2x load balancer = 2x (v)CPU, 2GB RAM [90]

Total: 14 (v)CPU, 22GB RAM

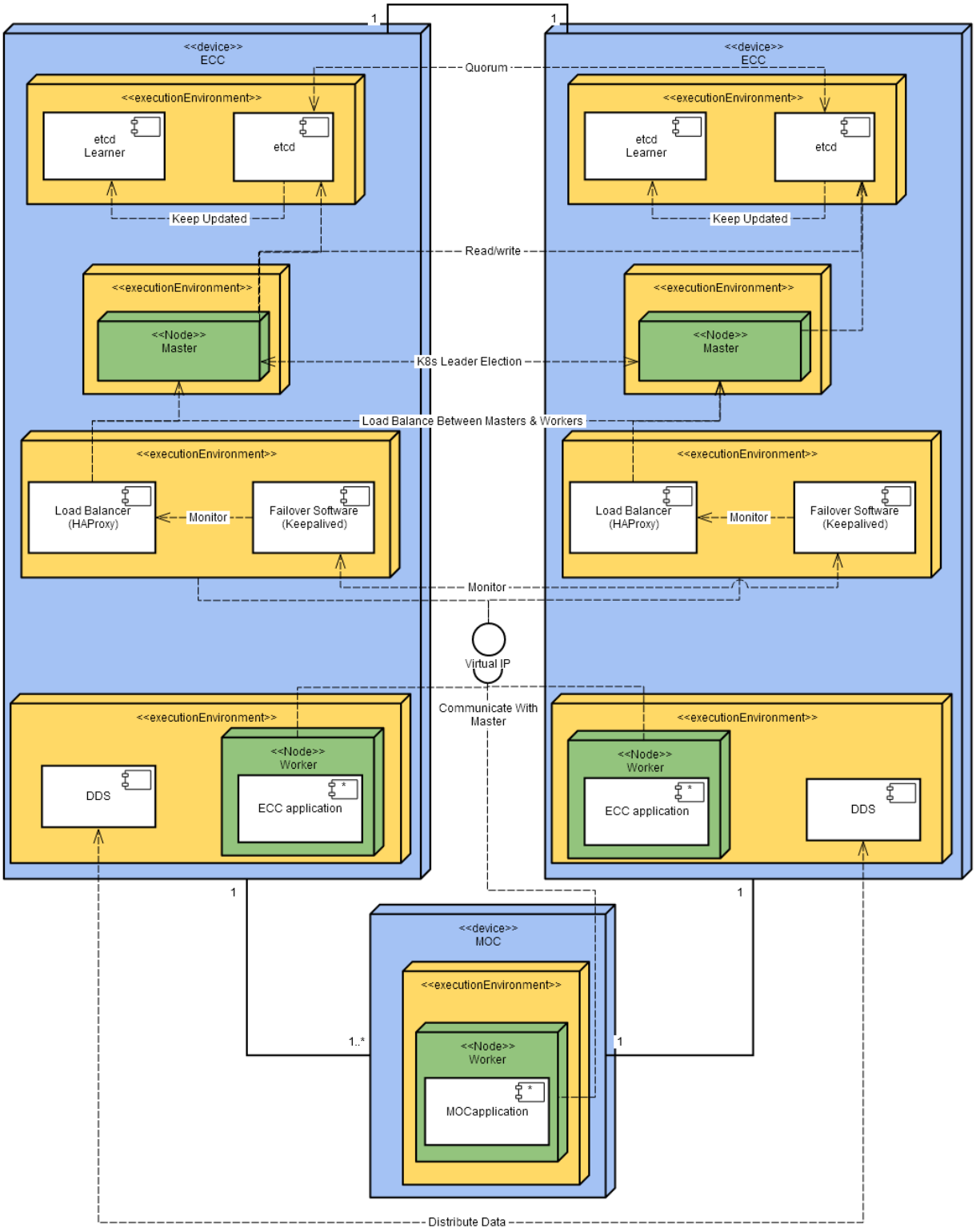

Figure 12: Single Cluster using Learners

4.6 RA4: Federated Clusters (Two AZs)

This architecture is depicted in Figure 13 and uses the KubeFed project described in Section 2.4. The system deploys a total of three clusters across the two ECCs. It contains one master cluster that generates the configurations for two worker clusters. The connected MOCs are either connected to worker cluster 1 or worker cluster 2. Distributing the connection of these MOCs across the two clusters ensures that not all worker nodes on the MOCs are immutable (the current workloads will continue but cannot be changed) if one of the worker clusters is down.

The load balancer that connects the worker nodes with the master nodes of the worker cluster has been kept abstract because the same load balancer as the one connecting the worker clusters with the master cluster can be used. Both the worker clusters have their own dedicated etcd cluster of three nodes to make sure that quorum is only lost if there is a failure on the ECC level.

A challenge that remains is the etcd cluster that services the master cluster, which currently forms a SPOF since it is only deployed on two AZs. To address this challenge, one of the two aforementioned solutions could be used, either the etcd shim or the etcd learners. The presented architecture has abstracted this part of the architecture. The evaluation of RA2 and RA3 should give insight into which solution (shim or learners) is the most appropriate. Since this is uncertain at the moment, it would result in having to design two nearly identical reference architectures that only differ in the back-end storage. Whereas the most appropriate back-end storage for two AZs deployments is implicitly found in the evaluation of RA2 and RA3. As mentioned in Section 2.4, the master clusters in Kubefed do not support leader election across multiple clusters. In case this is implemented in the future, one master cluster can be installed in each AZ, given that the leader election on these clusters works according to the same principles as leader election in master node components. Quorum-based storage (like GlusterFS) is not possible in this environment containing only two AZs, therefore the current DDS is included in the model as the storage solution.

If the master cluster goes down, the worker clusters are still able to serve workloads and restore pods in case a pod fails, but no new configuration can be created for these clusters.

Theoretical minimal requirements

6x master node = 12x (v)CPU, 12GB RAM [87]

8x etcd nodes (without a failover strategy for Master Cluster etcd) = 16x (v)CPU, 32GB RAM [88, 89]

2x load balancer = 2x (v)CPU, 2GB RAM [90]

Total: 30 (v)CPU, 46GB RAM

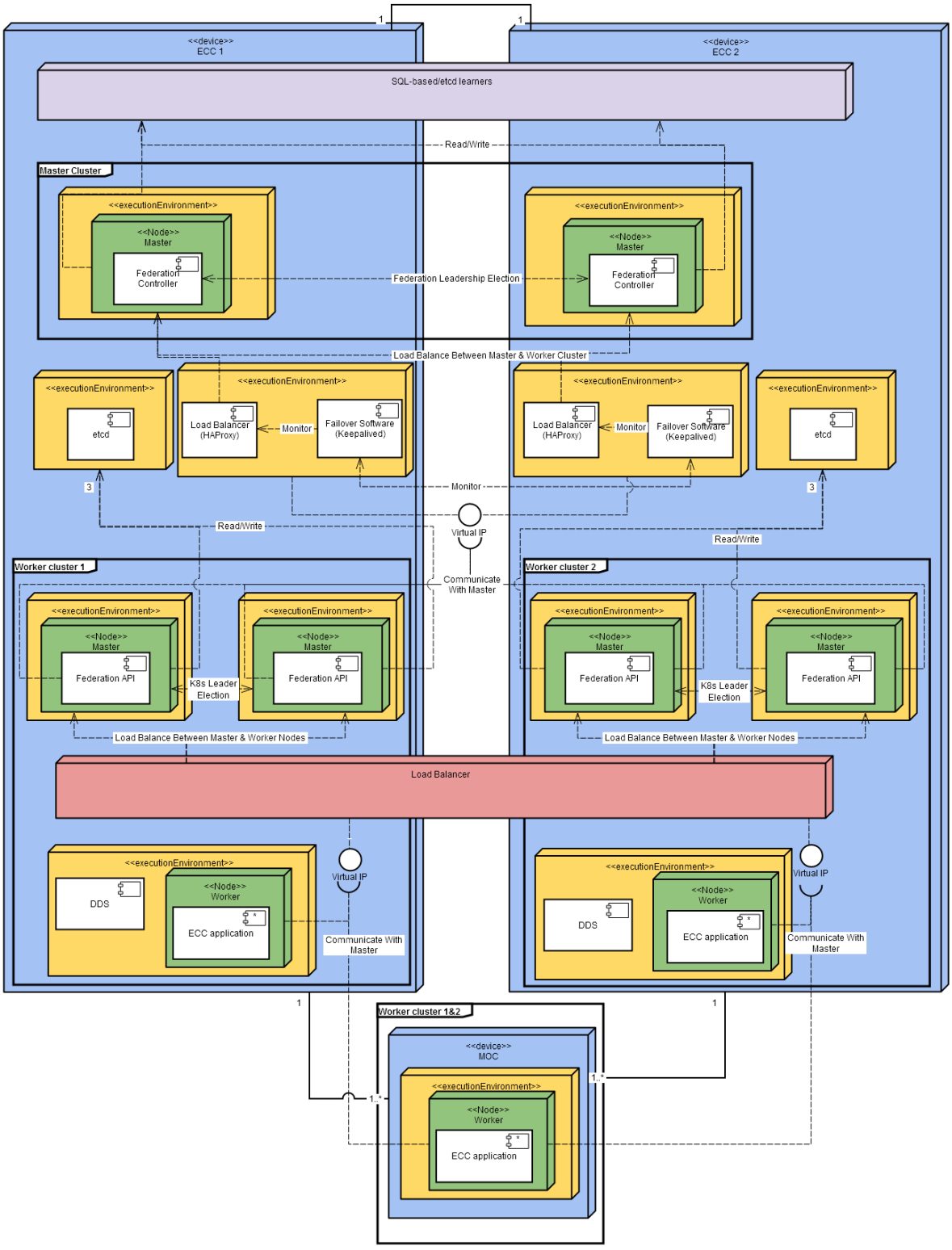

Figure 13: Federated Cluster on Two AZs

4.7 RA5: Federated Clusters (Three AZs)

This architecture also consists of federated clusters but uses three AZs for the etcd cluster that services the master cluster. The architecture is depicted in Figure 14. The architecture has similarities with RA4 but allows for quorum to be formed by using the quorum node described in Section 4.3. Like the architecture described in Section 4.3, it allows for a relatively standard (federated) K8s deployment. Because of the quorum node, a quorum based storage solution can be deployed. In the diagram the connection for reading and writing data from the ECC/MOC applications to the distributed storage is not shown in order to maintain readability of the overview.

Theoretical minimal requirements

6x master node = 12x (v)CPU, 12GB RAM [87]

9x etcd nodes = 18x (v)CPU, 36GB RAM [88, 89]

2x load balancer = 2x (v)CPU, 2GB RAM [90]

Total: 32 (v)CPU, 50GB RAM

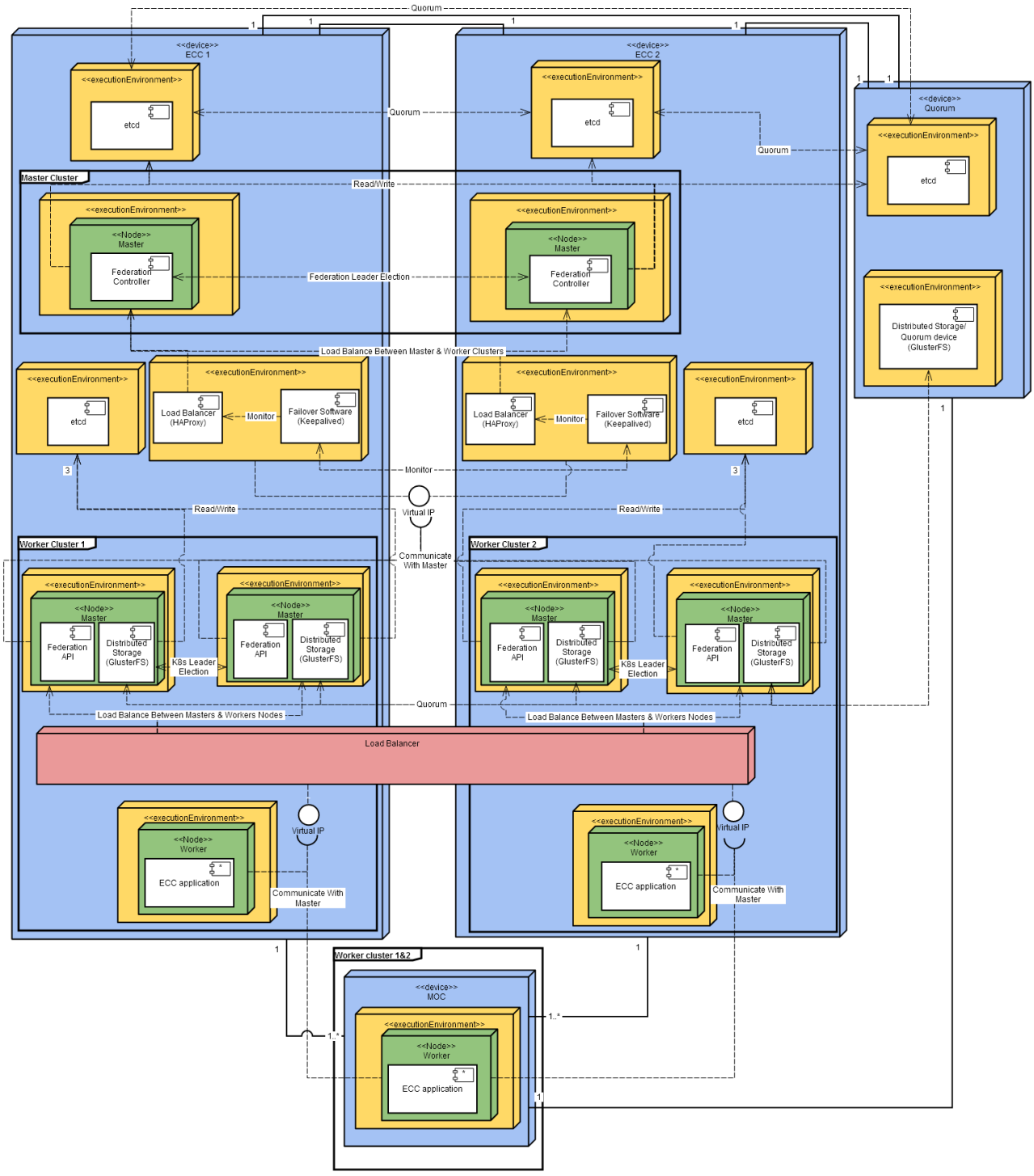

Figure 14: Federated Cluster across Three AZs

4.8 Comparison

Due to time limitation, prototyping all five reference architectures was outside the scope of our research. Therefore, we selected two reference architectures to be prototyped. This selection has been done based on the answers to a questionnaire with experts within the domain. Due to scheduling

challenges we digitally provided a concise document to the experts containing the reference architectures and a summary of the relevant information, combined with the questionnaire. The questionnaire used a four-step Likert scale ranging from 'Strongly Disagree' to 'Strongly Agree'. A fifth option of 'No Opinion' was provided as well, since a four-step Likert scale forces the respondent to give an opinion, and adding a neutral midpoint as is typical in a five-step Likert scale can result in a misuse of this neutral midpoint. The options were translated to values ranging from -2 for 'Strongly Disagree' and +2 for 'Strongly Agree'. 'No Opinion' had an associated value of 0. The respondents were asked to assign a weight to each question ranging from 0 to 3, corresponding with the perceived importance. Because of the small number of respondents (four), it was not meaningful to apply statistical methods to the values. The values from the answers were multiplied by the associated weight for each question. The values from all respondents were combined in this way to create a score for the most appropriate architectures. All questions were formulated in a positive way, in order to avoid confusion and make reverse scoring unnecessary. Each question had an associated comments field that could be used to give additional remarks on the characteristics of each architecture.

4.8.1 Characteristics

The characteristics on which the designs have been evaluated are derived from the ISO/IEC 25010 [99] standard for system & software quality. Figure 15 shows the ISO/IEC 25010 characteristics. The most important characteristics as underlined in the interviews have been chosen for the questionnaire. The questions mainly focused on reliability and whether the proposed designs could be applied in the implementation of mission-critical on-premise systems.

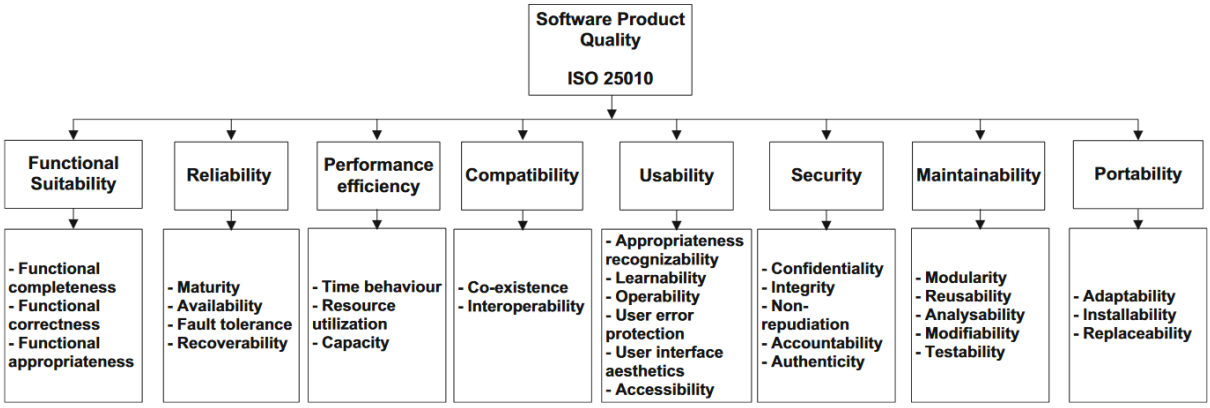

Figure 15: Quality model ISO/IEC 25010 [100]

Table 8 maps the questions of our questionnaire to the corresponding ISO 25010 characteristics. Questions 1-7 were asked for each specific design. Questions 5-7 were specifically on three separate systems that were known by the experts, to give a more tangible concept of what kind of environments the designs should be able to operate in. Systems #1 and #2 have very high redundancy requirements, whereas system #3 has lower redundancy requirements. This has been done to evaluate whether the designs would be adaptable enough to support different systems with differing requirements. Questions 8-10 were asked once in the questionnaire, since we assumed that the defined failover times are the same for each architecture. The questionnaire also contained a question about the maintainability of the system over a long period of time (>30 years). This question was deemed to be unanswerable by the respondents and is therefore not included in the report nor as further assessment of the reference architecture.

Nr.	Question	ISO 25010 characteristic
1	The system is simple	Maintainability/Analysability
2	The system is technology agnostic (apart from K8s)	Portability/Replaceability
3	The required system resources are acceptable	Performance efficiency/ Resource utilization
4	The system is likely to satisfy the redundancy requirements	Reliability/Fault Tolerance
5	The system would be able to support system #1	Functional Suitability/Functional appropriateness
6	The system would be able to support system #2	Functional Suitability/Functional appropriateness
7	The system would be able to support system #3	Functional Suitability/Functional appropriateness
8	The specified recovery times (monitoring intervals + failover times) for Node failures are sufficient for the mission-critical, on-premise systems	Reliability/Recoverability
9	The specified recovery times (monitoring intervals + failover times) for Pod failures are sufficient for mission-critical, on-premise systems	Reliability/Recoverability
10	The specified recovery times (monitoring intervals + failover times) for Load Balancer failures are sufficient for mission-critical, on-premise systems	Reliability/Recoverability

Table 8: Questions as ISO 25010 Characteristics

4.8.2 Selection

Based on the questionnaire, we ranked the reference architectures according to the experts' preferences. In total, four experts responded (a Software Architect, two Infrastructure Architects and a Cloud Architect). The respondents found the federated designs too complex for the on-premise context. The best regarded design was RA1, mainly because the experts knew this design was close to the standard way of deploying K8s HA and has tested real-world examples. The other two single cluster designs (RA2 and RA3) had comparable scores, which were higher than the federated options. RA3 is based on not yet implemented features and is therefore less suitable to prototype than RA2, which has some experimental support. Hence, we chose to prototype RA1 and RA2 for further evaluation.

5 Testing & Validating

In this chapter, our systematic testing strategy for K8s cluster architectures is proposed and validated. The strategy is used to validate RA1 and RA2 by testing these architectures on the non-functional requirements (Req3 & Req5). Scholarly researches on K8s environments have used different strategies to test these environments. These strategies are discussed and used as a basis for our testing strategy in this chapter. Our strategy is then discussed and validated by applying it on a relatively standard HA cluster (RA1), which shows that it produces correct and useful results.

First, this chapter describes the process for creating the prototypes, since a RA1 prototype is needed to validate our testing strategy. Secondly, the testing strategies available in the literature are discussed. Thirdly, our testing strategy is described. Fourth, we validate our testing strategy and give the results of testing RA1. Lastly, the RA2 is tested and the architectures are validated with respect to the requirements. This chapter mainly describes research activities conducted in the Design cycle (prototyping & creating and evaluating a testing methodology), with some input from literature research conducted in the Rigor cycle.

5.1 Deployment Tool

Deploying K8s clusters is often done by using one of the available deployment tools. For RA1, Kubespray [101] has been used, because of its ease of use and completeness. Kubespray can deploy a functioning HA cluster in a reproducible way. By default, Kubespray deploys reverse proxies on each worker node. Reverse proxies offer the same functionality as a load balancer would normally offer, although less configurable and feature-rich, it allows the worker node to communicate with multiple master nodes. Therefore, a load balancer is unnecessary, which simplifies the prototype and reduces the potential impact of the external load balancer misconfigurations. This allows us to focus on availability mechanisms specific to K8s.

To prototype RA2, K3s [102] was used. K3s is a fully compliant K8s cluster which that stripped a lot of alpha and legacy features present in upstream K8s, and combines most K8s components into one binary. The result is a considerably smaller footprint imposed by the cluster on the available hardware. The main reason for using K3s for this prototype is the built-in deployment of Kine. Version 0.10.2 supports a HA deployment across two nodes, with the master nodes connecting to a Postgres backend. This adequately represents the design for RA2.

The used VMs made use of networked system times from a public service which offers time synchronization, to make sure that timestamps on each VM and on the testing machine were consistent. The system times of each VM are likely to have some (small) inconsistencies, since perfectly synchronized times are unrealistic. We consider this uncertainty as being unavoidable and the impact on results as acceptable, with the uncertainty being in the order of milliseconds.

5.2 Installing and Configuring the Clusters

The prototyped clusters have been created according to the HA mechanisms described in Section 2.2.3. The prototype was installed on several VMs running on a single PC. All VMs ran CentOS 7 and K8s 1.16. To make sure every test is done on the same environment, snapshots of each VM have been made. By creating and restoring snapshots every test is done on a cluster with the same starting point. This way, the cluster has no knowledge of the test (and failure injection) done before.

Both prototypes for RA1 and RA2 consist of two master and two worker nodes. The worker nodes ran two instances of NPM (a web server) containing a hello-world application. This application is stateless

and serves short lived connections. A stateless application is chosen because it is easier to install than stateful applications. In addition, most applications in the As-Is situation are currently stateless.

In the default configuration, the kubelet posts the node's status every ten seconds and is allowed to fail this status update up to four times, leading to a mean reaction time of around 38 seconds [33]. On top of this, the controller waits another 300 seconds before evicting the unavailable pods from its endpoints and scheduling new pods onto other nodes. This means a total outage of these pods of around 340 seconds, which is too high in a HA setup. In our configuration, the kubelet posts the node's status every second (the lowest setting). Next to this, it is allowed up to two consecutive fails before being marked *NotReady.* The master starts evicting pods two seconds after the node is marked *NotReady*. Lowering the amount of allowed consecutive fails to one is a possibility, but this increases the chance of a false positive. We assume that the high-end networking hardware installed on-premise is able to perform reliably with these monitoring intervals. This should render the chance of a false positive negligible.

One of the availability mechanisms of K8s is the leader election of the master components, for example, implemented by the kube-controller-manager. The kube-scheduler uses the same mechanism. Therefore, focusing on the kube-controller-manager should provide sufficient data on the availability impact of master component failures. By default, the standby instances of the kube-controller-manager are configured to acquire the leadership fifteen seconds after the current leader has renewed its lease. In this standard configuration the current leader refreshes its lease every ten seconds. This is deemed too high in a mission-critical setting. Therefore, we configured the non-leader kube-controller-manager to acquire the leadership three seconds after the leader has renewed the lease. The leader renews its lease every two seconds, which should lead to the leader keeping its lease as long as it is active. The controller-manager also has a configuration that handles how often the non-leader tries to gain leadership once the lease has expired, which is configured to one second. A hard requirement by the kube-controller-manager is that the interval for the non-leader retrying to gain leadership is lower than the interval for the leader renewing its lease. Another requirement is that the interval for renewing the lease by the master is lower than the lease duration. At the time of writing, the lease durations are programmed in seconds in the source code of the leader election component [103], milliseconds are not supported, with one seconds being the minimum accepted value. Therefore, one, two and three second intervals are the lowest possible settings for leader election.

For RA1, three independent etcd instances have been deployed in three VMs. Figure 16 shows a deployment diagram of the RA1 prototype.

For RA2, two instances of PostgreSQL have been deployed on two VMs in a Master/Slave configuration. The PostgreSQL instances shared a Virtual IP through Keepalived. Figure 17 shows a deployment diagram of the RA2 prototype.

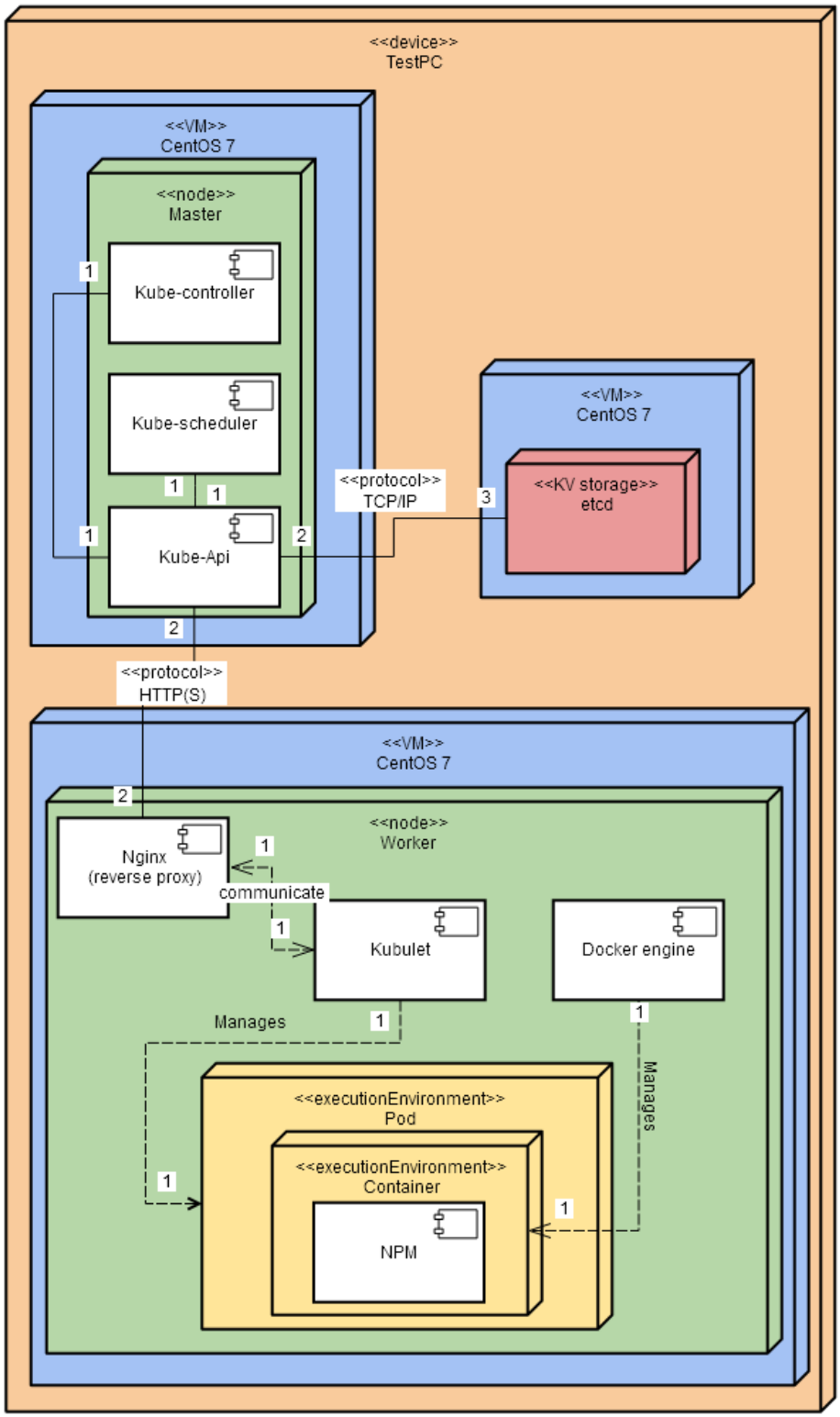

Figure 16: Test setup RA1

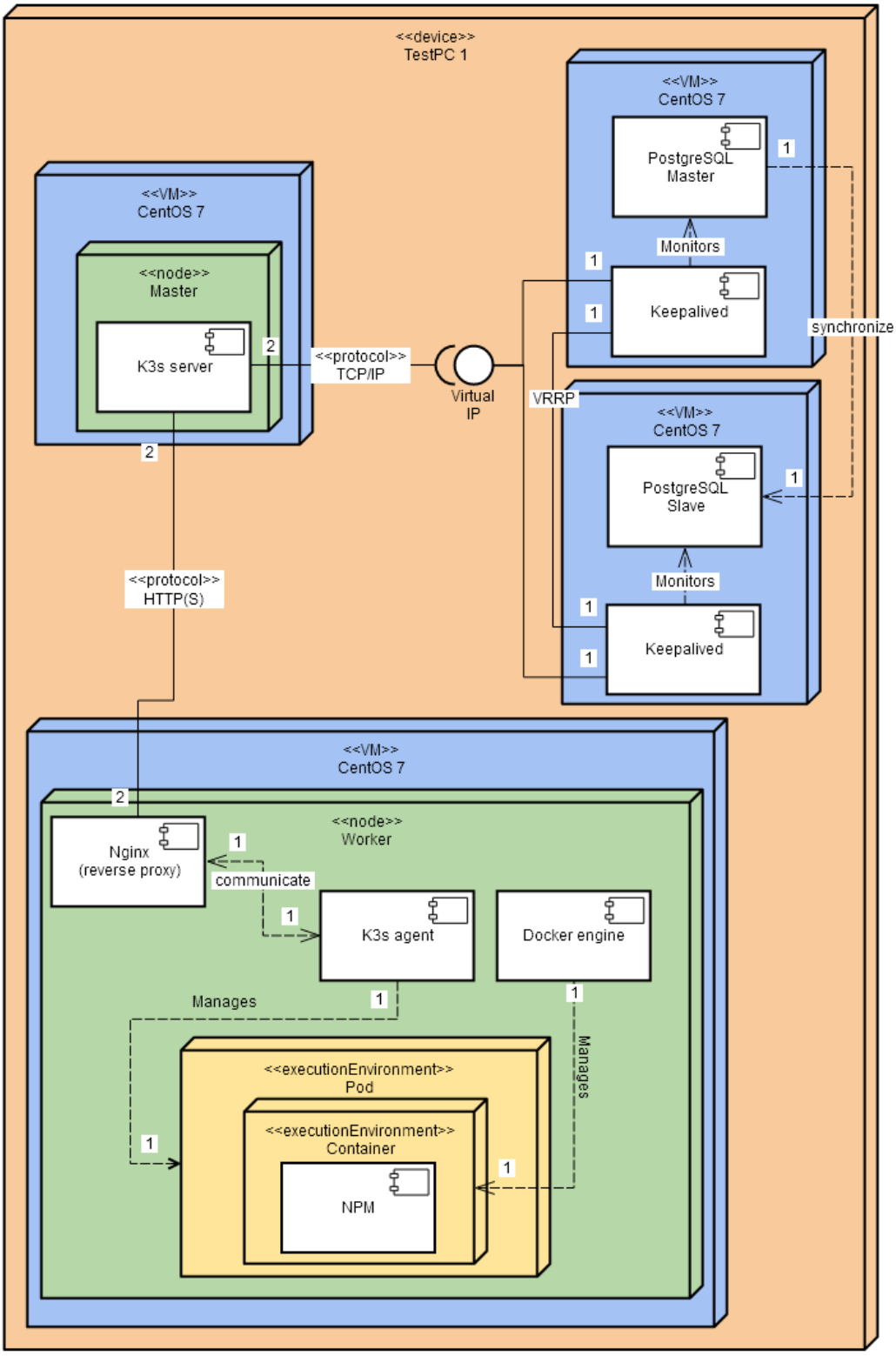

Figure 17: Test Setup RA2

5.3 Testing Strategies in Literature

We studied three strategies for testing clusters on non-functional characteristics (mainly availability) from academic literature. Vayghan et al. [33, 34] and Kanso et al. [29] tested the availability achievable with K8s under different circumstances and configurations. These strategies have been used as a basis for our testing strategy.

Kanso et al. [29] deployed three master nodes with three replicas of etcd (as is common HA practice). Three worker nodes were deployed as well. The master nodes resided behind a virtual IP, letting the worker nodes connect to the master nodes via a single IP. Although not specifically described as such, the virtual IP was used in combination with a load balancer. This allowed each worker node to communicate with the three master nodes. The authors defined a testing strategy with four steps: 1. sanitize the environment by restoring the state of the cluster to a base snapshot. 2. when the sanitized environment is running, inject a fault at the level under study (e.g., pod or node). 3. extract the event's timestamps of the logs at different levels (e.g., Docker, etcd, K8s). 4. analyse these timestamps and compare these results to the timestamp of the injected failure and draw appropriate conclusions. The authors defined four main failure categories: Network, Pod, K8s (master component) and Host failures. Each of these failure categories have been defined in more specific failure components that are shown in Figure 18.

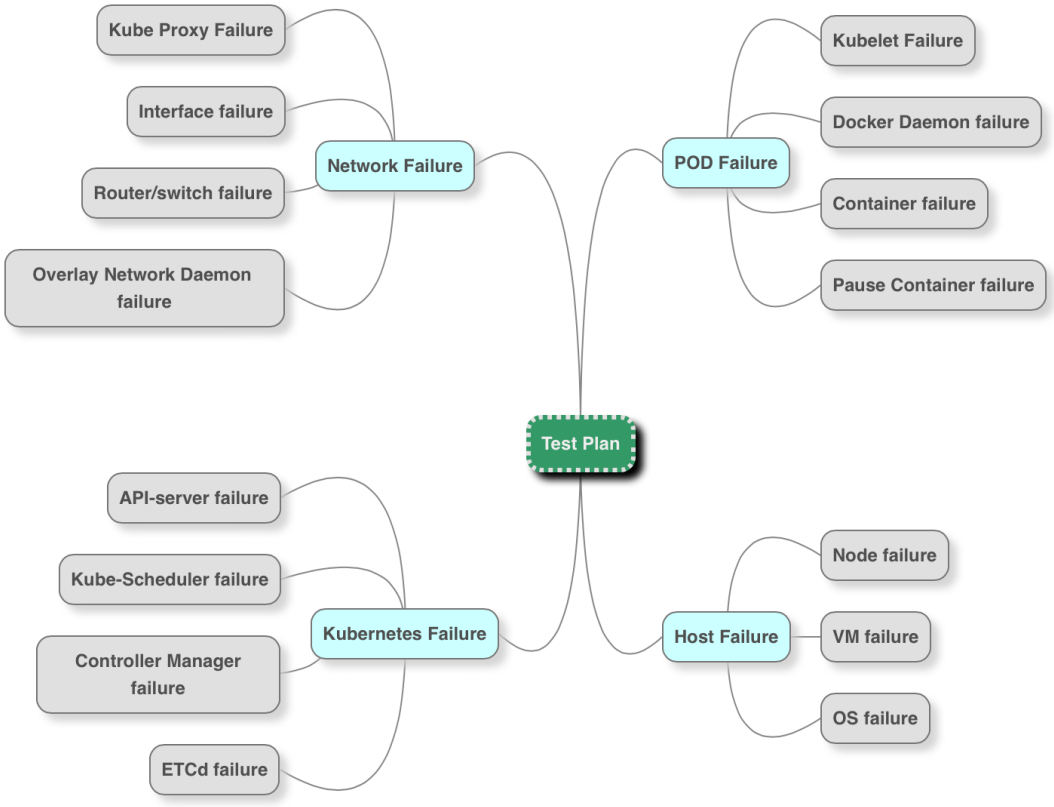

Figure 18: Test plan overview [29]

Next to this, Kanso et al. [29] have defined different events that can be observed after a failure has been injected. These events serve as reference points for calculating the reaction times. The events are listed in Table 9.

Event	Description
Failure detected by the kubelet	Duration between failure and the failure being detected by the kubelet
Container created	Duration it takes for the container runtime to create a new container after the failure was injected
Container active	Duration it takes for the container to become active after failure injection
Failure reported to master	Duration it takes for the master to realize a failure has occurred, since the failure injection
Recovery reported to the master	Duration it takes for the master to realize the container has successfully recovered since the failure injection

Table 9: Events during recovery

Vayghan et al. [34] deployed only a single master node and kept master node failure out of the scope of their research. Next to the single master cluster, two worker clusters were deployed. For accurate comparison of the timestamps of different events, the VM times were kept up-to-date through networked time synchronization. The experiments consisted of three failure scenarios: container failure, pod failure and worker node failure. The deployed and tested pod contained the VLC video streamer. Every time the pod went live, VLC started streaming a video file. Restarting the pod meant that the video started from the beginning again, i.e., the state was not replicated. The authors experimented with differing monitoring intervals. Like Kanso et al. [29], the authors defined a number of reaction metrics, which are listed in Table 10.

Metrics	Description
Reaction Time	Time between the failure event injected and the first reaction of K8s
Repair Time	Time between the first reaction and the repair of the failed pod
Recovery Time	Time between the first reaction and when the service is available again
Outage Time	Duration of time interval in which the service was not available (sum of the reaction time and the recovery time)

Table 10: Metrics used by [34]

Vayghan et al.'s earlier research [33] conducted on K8s availability does not mention the master node setup, but mentions K8s deployment on three VMs. Since master node failure is not considered in the investigation, we assume that only one master node is deployed. Like Vayghan et al.'s later research [34], the VM system times are synchronized by networked time. Monitoring intervals are set to the default value. This means that the kubelet reports its status every ten seconds and is allowed to fail to deliver this status update up to four times before the master nodes marks the concerned node as failed. The failure scenarios that were considered are node and pod failure.

All three investigations described above have tested the duration of events in the K8s system after a forceful termination of either a pod's process or a running VM. Vayghan et al. compared the reaction time of K8s to a forceful termination outside of the K8s system to the reaction times when using K8s administration commands [33]. The researchers concluded that the forceful termination results in much longer outage time than termination through administrative (K8s deletion) commands. This is likely

because K8s does not get the chance to gracefully stop the workload, when forcefully terminating the workload. Forceful termination (either through killing pod processes or killing complete nodes through powering down a VM) offers a much more realistic scenario. We concluded that forceful termination is the most appropriate way of testing the K8s availability mechanisms.

5.4 Creating Our Strategy

In this section, our testing strategy for assessing the prototypes is defined. To generate results which are close to the real-life context, a similar strategy to the ones discussed in Section 5.3 has been defined. The resulting strategy consists of a cycle of four steps, namely starting the cluster by using snapshots, injecting a failure, recovering the logs and powering down the cluster. An overview of the cycle is given in Figure 19. Between starting the VMs and injecting the failure a waiting time of 75 seconds was present, to allow the cluster to completely start and stabilize. Between injecting the failure and retrieving the logs a waiting time of 35 seconds was present. This allowed K8s to react to the failure. The cycle ran thirty times for each test scenario, to make sure the results were consistent and to reduce the impact of potential anomalies on the results.

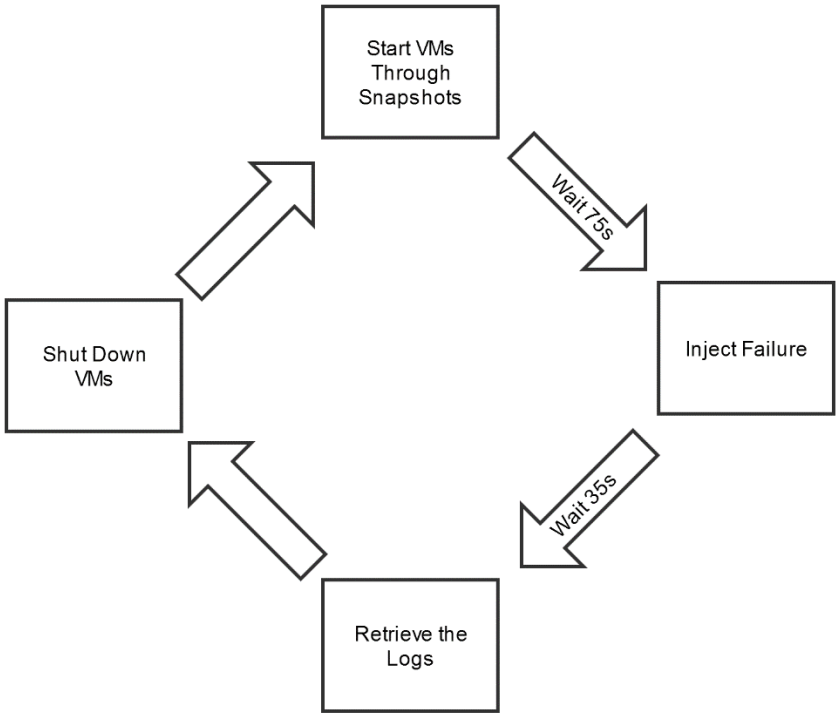

Figure 19: Overview of Testing Strategy Steps

5.4.1 Failure Scenarios

The prototypes have been tested on three failure scenarios. The first scenario is the pod failure scenario, in which the cluster's reaction time has been tested by forcefully terminating a pod process. The node failure scenario has been simulated by abruptly powering down the VMs that were running the specific node. The master node component (kube-controller-manager) failure scenario was simulated by forcefully terminating the master node that contains the leader of the tested component at that time. Table 11 offers an overview of the failure scenarios considered in this research. The table also maps the failure scenarios to the possible failure scenarios described in Section 2.1.2. All three failure scenarios

described cover more than one of the possible failure categories described in Section 2.1.2, since the failure scenarios can be caused by multiple failure categories.

Not all failure scenarios listed by Kanso et al. [29] have been used in these failure scenarios, since the reaction of K8s to the different scenarios is the same in a number of cases. K8s does not distinguish between a node, VM or OS failure, which all result in the kubelet not posting the node status. Next to this, kubelet failure and network failure also result in the kubelet not posting its status to the master nodes. K8s' reaction to these situations is identical, the leader kube-controller-manager notices that the node has failed to post its status an X amount of times (this is configurable) and marks it *NotReady*. Whether the failure of posting the status is due to the kubelet failure or due to a network failure does not make a difference for the kube-controller-manager. Consequently, it does not make a difference for the way in which K8s reacts to the failure. Furthermore, testing of the network failures will most likely not result in useful results, since the network components used in testing are completely virtualized. Testing these kinds of network setups is very different from the reality of mission-critical systems. Therefore, this has been left out of the scope.

Failure scenario	Failure category according to Section 2.1.2	Simulated by	Considered functional	Considered repaired
Master component failure (kube-controller-manager, kube-scheduler)	Container /application /network /hardware /VM /VMM	Terminating the node's VM which hosts the master component that is currently the leader	When leader election has finished and the other component has taken over the leadership	When failed components are marked as such
Host Failure (Node failure)	VMM /hardware /network	Terminating the tested nodes	When the non-functioning pods are deleted from the endpoints (no requests will be redirected to these pods)	When all (non-system) pods are failed over to functioning nodes in the case of worker nodes, in master nodes the new leader has to be elected
Pod failure	Container /application	Terminating a specific pod's process	When the non-functioning pods are deleted from the endpoints (no requests will be redirected to these pods)	When the new pod is considered in working state by the kube-controller-manager

Table 11: Failure scenarios

5.4.2 Getting Results

To get the duration of the service outage, we applied a similar method as the ones applied in the studies presented in Section 5.3. By comparing timestamps of the inserted faults with timestamps in the different technologies' logs, conclusions can be drawn on the duration of the service unavailability. K8s does not provide a centralized log to collect all the data needed for the research. Next to this, events are often accompanied by their age, so no specific timestamp is given. Therefore, some logs are created by

watching the kube-apiserver through the respective kubectl commands, adding the timestamp of the observation and saving this output to a log file. Table 12 lists the places from which data is extracted for each event after a failure. Specific monitoring tools like Prometheus [104] are not used, since the research involves non-standard deployments of K8s. For example, there are reports of Prometheus not working correctly with K3s [105, 106]. To make sure that the testing methods can be used on both designs, a more low-level way of monitoring and extracting data was used. We found out that the combined binary of K3s created combined logs. Therefore, the data sources were different for the K3s environment. These logs were also less verbose. The required changes to the data sources for the prototype using K3s has been included in Table 12.

Event	Notified by	Data source
Host Failure	Kube-controller-manager	Watch the node list, combined with timestamps, saving to a log file
Pod Failure	Kubelet, Kube-controller-manager	Kubelet: System log Kube-controller-manager: watch the pod list, combined with timestamps, saving to a log file (for RA2 the K3s binary logs contain the combined logs)
Pod Started	Kubelet, Kube-controller-manager	Kubelet: System log Kube-controller-manager: watch the pod list, combined with timestamps, saving to a log file (for RA2 the K3s binary logs contain the combined logs)
Leader election	Container	Controller logs (for RA2 the K3s binary logs contain the combined logs)

Table 12: Log data for each event after a failure

5.5 Validating the Testing Strategy

To validate the testing strategy the prototype of RA1 has been used. This prototype is comparable to the clusters used in the aforementioned researches. Showing that our testing strategy creates reproducible results comparable to the results from the aforementioned strategies shows the validity of our testing strategy, rendering it usable for the reference architecture. Because RA1 is one of the preferred reference architectures, it also validates the non-functional requirements of one of the preferred architectures. The generated data fits one of the availability metrics defined in the taxonomy mentioned in Section 2.1. This metric is the 'detection interval metric', which is described as 'the maximum time interval necessary to detect a failure'. For each test this metric will also be included, which is the highest, legitimate, value for which the master notices the failure.

5.5.1 Test Results

All test results are presented as the time between issuing the kill command and the time the described event was reported, in seconds. These results do not reflect how much service availability (as described in Section 2.1.1) can be achieved with K8s. Since the clusters do not have a long runtime and do not service any real-world workloads.

Pod Failure

The first scenario tested is the pod failures in which the tested pod process was forcefully terminated. Table 13 shows the results of these tests. It shows that the median repair time is 1.577 seconds and that the median time for system to be functional is 0.505 seconds. During one of the tests the Kubelet took longer than one second to detect the pod failure, namely 1.039 seconds. Two of the test results showed complete recovery times (the last event) of more than two seconds (2.089 and 2.018). The detection interval in this test scenario was 1.065 seconds, this is longer than the aforementioned 1.039 which was the longest time it took for the Kubelet to notice the failure. The detection interval time is the time it took for the master to notice the failure, since the Kubelet has to communicate the failure to the master, this takes some additional time.

Event	Mean (s)	Median (s)	Std. Dev (s)
Kubelet noticed pod died	0.518	0.505	0.268
Master noticed pod died	0.551	0.545	0.270
Kubelet noticed container was started	1.570	1.572	0.271
Master noticed container was started	1.582	1.577	0.266

Table 13: Reaction times after pod failure RA1

Node Failure

In this scenario, a worker node VM was forcefully terminated. Table 14 shows the results of these tests. It shows that the mean and median repair times are within ten seconds and that the system is functional in seven seconds. In five instances it took longer than ten seconds for the new pod to be started after the failure, but in only one instance this was above eleven seconds. The detection interval in this test scenario was 5.036 seconds.

Event	Mean (s)	Median (s)	Std. Dev (s)
Master marked node *NotReady*	2.716	2.449	1.064
Master started evicting pod	6.905	6.700	1.063

Master sets status of new pod *ContainerCreating*	6.985	6.771	1.062
Time kubelet notified pod was started	8.376	8.259	1.028
Time master marked pod as running	8.573	8.377	1.288

Table 14: Reaction times after node failure RA1

Kube-controller-manager Failure

In this scenario, the master node that ran the leader instance of the kube-controller-manager was forcefully terminated. Table 15 shows the results of these tests. It shows that the median time for the system to become functional was 3.974 seconds and a median time of 19.284 seconds before the system can be considered repaired. In one of the test instances it took the leader election longer than eight seconds to start, this is considerably higher than the other results and is regarded as an outlier and therefore excluded. The leadership election performed as expected, but marking the terminated node did take a long time (around twenty seconds). This can be due to the fact that the new leader has to process a lot of things (the logs show a lot of events) and marking the node *NotReady* happens after these consecutive events, or because there is some hidden configuration left that defined a timeout which we could not find. The time it took for the non-leader instance to notice the current leader had failed was not saved to the logs. Therefore, this time could not be used as the detection interval metric, which would have been the most appropriate value for this metric. In this test scenario the detection interval metric is denoted as the 'New Leader Elected' time, which is 5.421 seconds.

Event	Mean (s)	Median (s)	Std. Dev (s)
New leader elected	3.744	3.974	0.934
Failed node marked *NotReady*	19.238	19.284	0.990

Table 15: Reaction times after Kube-controller-manager failure RA1

5.5.2 Comparing Results

In this section, the generated results are compared to the results from the researches on K8s availability [29, 33, 34]. The reaction times for pod failure found in this research are comparable to the results reported by the aforementioned researches. The node failure times are not in line with the reported results in the other research that tested node failure [34]. The results differ because the configurations used in this research are more strict (less failed postings are allowed), next to this, the eviction timeout is set in this research to two seconds, whereas Vayghan et al. [34] configured a zero second eviction timeout. The configuration differences explain the differences in results, if similar configurations are used it is likely that the results will be more comparable. The referenced papers did not consider kube-controller-manager failure, so no comparison of those results could be made.

Due to the generated results being comparable to the reported results in the other investigations, it can be concluded that the used testing strategy is capable of generating valuable and reproducible data. Therefore, the strategy can be used to evaluate the prototype of RA2. The validation of the strategy resulted in the reaction times of RA1, which is one of the reference architectures. Hence, the results have been used to validate one of the reference architectures as well.

5.6 Testing RA2

The test results for RA2 are presented the same way as the test results for RA1 in order to easily compare the results. RA2 is prototyped by using K3s, this deploys a combined binary for the necessary

K8s components on each node depending on the role of the node. The deployed binary for the worker nodes mainly consists of the Kubelet. It differs in the way logging is done compared to the environment deployed by Kubespray. Therefore, the container creation and container start message as found in the kubelet log in RA1 could not be found in the logs generated in these test cases. This does not have a large influence on the results of these tests, since the most important events in the event chain after a failure are the reactions of the master. When the master has noticed the pod has been restored it adds this to the service endpoints, after this the service is reachable for requests. Therefore, the service can only be regarded as being restored after the master has added the new pod to the service endpoints. The presented results, like the results for RA1, include the 'detection interval metric'.

5.6.1 Pod Failure

The found reaction times for the pod failures with RA2 were around 0.5 seconds longer as the reaction times found with the same test for RA1 and have been presented in Table 16. The median time for the system to become functional again was 1.013 seconds. The median time for completely restoring the service was 2.070 seconds. The difference between the test results in both tests is likely due to the way in which the cluster is deployed. Further configuration could bring the results closer to those of RA1. One result was unusable due the master noticing the pod had died before the kill command was issued, hence, this result has been left out. We believe that there are two likely explanations for this error, either the pod had crashed because of an unrelated reason just before the kill command, or because the VMs and host machine were not able to synchronize the system times because of a network hiccup. The detection interval for this test scenario was 1.806 seconds.

Event	Mean (s)	Median (s)	Std. Dev (s)
Kubelet noticed pod died	-	-	-
Master noticed pod died	1.017	1.013	0.371
Kubelet noticed container was started	-	-	-
Master noticed container was started	2.095	2.070	0.411

Table 16: Reaction time after pod failure RA2

5.6.2 Node Failure

The found reaction times were very similar to the ones found in the node failure test scenario of RA1 and are presented in Table 17. Failure detection did take a bit longer for this prototype compared to RA1, but overall the repair time was similar. The median time for the system to become functional again was 3.305 seconds. The median time for completely restoring the service was 8.734 seconds. In two of the thirty test runs it took the master longer than ten seconds to completely restore service with a new running pod, namely 10.26 and 10.578 seconds. All test results were usable. The detection interval for this test scenario was 4.826 seconds.

Event	Mean (s)	Median (s)	Std. Dev (s)
Master marked node *NotReady*	3.333	3.305	0.916
Master started evicting pod	7.376	7.385	0.940
Master sets status of new pod *ContainerCreating*	7.535	7.622	0.980
Time kubelet notified pod was started	-	-	-

Time master marked pod as running	8.697	8.734	0.976

Table 17: Reaction times after node failure RA2

5.6.3 Kube-controller-manager Failure

This test scenario found similar reaction times as the kube-controller-manager test scenario for RA1 and is presented in Table 18. The median times reported for the system becoming functional again and for the service being restored are respectively 3.978 and 17.769 seconds. Like the kube-controller-manager test for RA1, the initial detection of the leader failure could not be found in the logs, therefore the 'New Leader Elected' event has been used as the detection interval. The detection interval for this test scenario was 5.686 seconds.

Event	Mean (s)	Median (s)	Std. Dev (s)
New leader elected	3.910	3.978	0.730
Failed node marked *NotReady*	17.734	17.769	0.714

Table 18: Reaction times after kube-controller-manager failure RA2

5.7 Analysis of the Selected Reference Architectures

The prototypes are further analysed according to the requirements as specified in Section 3.6.1. For easy reference, Table 6 has been copied here in Table 19.

Nr.	Requirements
Req1	The new system must not contain any SPOF.
Req2	The new system should make use of K8s.
Req3	Critical parts of the system must be failed over in case of a failure within thirty seconds.
Req4	The system should be technology agnostic (apart from K8s).
Req5	The new system should monitor the applications and components in a short interval without being susceptible to false-positives due to network hiccups.
Req6	The utilized open-source technologies must have a license which allows for classified usage in mission-critical systems.

Table 19: Requirements for Reference Architectures (Copy)

RA1 and RA2 have been designed without a SPOF, although one of the questionnaire respondents mentioned that single cluster solutions do inherently contain a SPOF. We believe that single decentralized clusters (which HA K8s clusters are) will not form a SPOF in this specific context. Running multiple clusters allows operators to perform rolling updates to cluster configurations while ensuring service availability. If for some reason, a faulty configuration is pushed to one cluster, the other cluster will continue servicing requests and the operator will know the configuration is faulty. This use-case is not relevant for this research's context, since the system will not be using rolling updates. When the system is active, it will keep running the same configurations. Since no new configurations will be pushed to the cluster and the system is decentralized by nature, a cluster-level failure is deemed highly unlikely, given that the used configuration is extensively tested. Therefore, both RAs satisfy RQ1. Next to this, both RAs made use of K8s (K3s is a simplified version of K8s), therefore satisfying RQ2.

Since the failover times are all within thirty seconds, both RAs satisfy RQ3, although the failure marking of the kube-controller-manager scenario is rather high. During the research it was not found why the node was marked around fifteen seconds after the new leader was successfully elected. Further research and configuration of the clusters is needed to reduce this time, which is outside the scope of this

research, because of time restraints. The respondents to the questionnaire indicated that a much lower failover time is preferred, with thirty seconds being the absolute maximum amount of downtime allowed in case of a failure, although this mainly applies to application or node failure, not necessarily controller failure. Both RAs reported a reduced failover time, although the kube-controller-manager scenario should be reduced further.

RA2 does currently not satisfy RQ4 since it relies on Kine. We, in addition to the questionnaire respondents, do not deem this to be a very impactful non-conformity, since there is a chance that Kine will be introduced in the mainline K8s, making it part of K8s. Next to this, another technology could be developed to fulfil this role, if the value proposition of running the system on two AZs is deemed high enough.

During the tests for both RAs no cases of pod evictions due to false positives were found, so both RAs satisfy RQ5. This does however not mean that no false positives will be present when implementing the architecture in a real-world scenario. Since the tests were done on a single development machine network latency was practically non-existent. The tests did show that K8s is capable of processing all the heartbeats correctly within the specified intervals. This is valuable, since the nodes had only access to the bare minimum (virtualized) hardware which is recommended for test clusters. In a real-world implementation the nodes will have access to more sophisticated hardware, which makes it likely that in this context K8s will also be able to handle the heartbeats. Next to this, the on-premise context contains high-end networking equipment, although this context will not have the same near-zero latency as the test setup. The used intervals are deemed to be suitable for this networking equipment and the relatively small physical distance the messages have to traverse.

Both K8s and K3s/Kine are released under the Apache 2.0 license, which allows for commercial usage and does not require the complete system to be open-sourced. Other technologies used (e.g. Keepalived & PostgreSQL) are replaceable by alternatives if the license does not allow usage in the target context. Therefore, both prototypes satisfy RQ6.

Requirement	Satisfied by RA1	Satisfied by RA2	Validated By
RQ1	X	X	Expert feedback, technical documentation
RQ2	X	X	Not externally validated
RQ3	X	X	Testing
RQ4	X		Questionnaire
RQ5	X	X	Testing
RQ6	X	X	Technical documentation

Table 20: Requirements satisfied by RA1 & RA2

6 Conclusions

Mission-critical systems in the defence domain have considerably stricter requirements compared to systems, even mission-critical ones, in other domains. One of these requirements is the relatively long lifetime (>30 years) of systems in the defence domain. Maintaining systems over such long periods of time is costly. Some of the applications which make up these systems are hardware and OS-dependent. This dependence complicates maintenance further. A novel, but increasingly popular, way of addressing this challenge is by containerizing the applications in the system.

Introducing containerization to the defence domain, like the introduction of most new technologies, has some inherent challenges. The technologies which support containerized applications (often referred to as Cloud Native technologies) are newer and less mature than the technologies currently utilized. Therefore, these technologies require careful consideration and additional research to be applicable in the defence domain. One of the most important technologies for effectively managing containers are orchestration solutions, of which Kubernetes (K8s) is the most popular. K8s is also used in this research.

One of the aspects which creates uncertainty is the way these Cloud Native technologies commonly achieve High Availability (HA). This is different compared to the current way of achieving HA in the defence domain. Achieving HA with Cloud Native technologies is generally done by utilizing the vast amount of distributed computing power in the public cloud. Vendors of public cloud solution have geographically dispersed data centres to offer these services. This makes it easier for the customer to reach their availability goals. Using the public cloud is generally not an option in the defence domain, since the deployed systems often have to function in remote areas, without reliable external network communication. Knowledge on achieving HA in these on-premise systems with Cloud Native technologies is lacking. Most of the information addressing these technologies is focused on using either public cloud solutions or large enterprise-grade private cloud solutions.

During this research, we created architectures by conducting design science. These architectures satisfy the availability requirements of mission-critical, on-premise systems in the defence domain.

6.1 Research Questions

This research set out to find a fitting architecture for a containerized cluster in an on-premise mission-critical context in the defence domain. A lack of academic literature on containerized workloads in this context was found, as well as a lack of practical knowledge in the defence domain on the availability mechanisms of this novel technology. The main research question posed was:

How can container orchestration be deployed in an on-premise context so that it satisfies the availability requirements of mission-critical systems in the defence sector?

This main research question was answered through several subquestions. The first subquestion was aimed at creating a theoretical basis on HA for this research:

RQ1. What is often meant by High Availability (HA) and how is it generally achieved?
- **a. What are the implications of HA in the cloud environment?**
- **b. What are mechanisms generally implemented to achieve HA in cloud environment?**

Through literature research a comprehensive view on HA in the cloud environment has been created. Availability is described in literature as: ‘the degree to which a system is functioning and is accessible to deliver its services during a given interval time’. HA is commonly achieved when a system is functioning and accessible for 99.999% of the time, although the defence domain imposes stricter availability

requirements. HA is generally achieved through replication of services and components. To make sure that these replicated services are reachable additional technologies like load balancers are required. The information gathered for this research question was used to correctly understand and apply mechanisms for achieving HA and categorizing several failure scenarios for our testing strategy.

The second subquestion was created to gain insight into the theoretical basis of K8s cluster federation:

RQ2. What is Kubernetes (K8s) cluster federation?
- **a. What is K8s and what are its related concepts and technologies?**
- **b. What is the state of the scientific knowledge on availability with K8s?**
- **c. What are methods to achieve HA with K8s, including multicluster solutions?**

By using technical documentation an overview of the most important K8s components has been created. This information is supplemented by information on how these components influence overall availability and what kinds of mechanisms are needed to achieve HA with K8s. The amount of scientific literature found on K8s availability was low but did provide an indication of the K8s availability mechanisms strength. This shows that K8s is able to recover within seconds from several failure scenarios.

An analysis of multicluster solutions has been done and summarized in this report. One of these solutions, KubeFed, has been discussed in more detail because of its potential relevancy to solving the technical challenge encountered in this research. The technical overview of K8s and federation has been used to create the in-depth understanding required for designing the reference architectures.

To create insight into the specific context of this research the third research question was drafted:

RQ3. What are the business considerations for implementing Cloud Native technologies in mission-critical systems?
- **a. What are current applications of Cloud Native technologies in the business domain?**
- **b. What are the business challenges for companies creating mission-critical systems and how can Cloud Native technologies help in addressing these challenges?**
- **c. What trade-offs should be considered when developing the system?**

Literature research combined with conducted interviews created insight into what challenges are generally addressed by using Cloud Native technologies, and what specific challenges are expected to be addressed by Cloud Native technologies in the defence domain. In general, it can be concluded that these technologies fit modern ways of designing software architectures (e.g., microservices) well, which makes it attractive for a wide range of business domains. The most important factors for implementing Cloud Native principles in the defence domain where the advantages offered by containerization. Current systems contain hardware and OS-dependent applications. This dependency decreases maintainability and inhibits innovation. Containerization has been used in other business domains to effectively tackle this challenge. Containers have to be managed properly through supporting technologies in order to be effective, this requires several other Cloud Native technologies. The usage of these supporting technologies is currently under research at organizations operating in the defence domain.

Trade-offs made for the reference architectures have been carefully considered and documented. One of the trade-offs considered the performance impact containerized applications encountered compared to bare-metal deployment, this impact was found to be manageable. Another trade-off considered was the increased overhead of running more redundant components, this was found to be impactful when

designing large federated systems. Next to this it was also important to consider when to include technologies in the reference architectures which had a low maturity. We argue that a low maturity technology with high potential value can be used in the reference architectures.

To create an understanding of the current situation for mission-critical systems the fourth research question has been created:

RQ4. What is currently a typical architecture for mission-critical systems and how are challenges regarding HA currently addressed?

A total of four interviews have been conducted for answering this research question. The interviews gave insights in the As-Is situation of a mission-critical on-premise system in the defence domain. The information from these interviews has been used to create a representation of a typical mission-critical system, as well as a description on how HA is currently achieved.

Currently HA is achieved by programming applications in a stateless way and replicating the data these applications need across the available machines. This way, applications can easily be restarted on a different machine, while still having access to the relevant data. To make sure that the correct applications are started at the right time a combination of monitoring and management software is used.

RQ5. What are the proposed reference architectures that satisfy the requirements of mission-critical systems?

- **a. What are the requirements of the on-premise container clusters, especially in relation to availability?**

The requirements for the reference architectures have been created by using the information gathered through interviews. The list of requirements has been kept concise in order stay within the focus of the research. These requirements, in combination with the trade-offs discussed for RQ3 have been used to create the reference architectures. The reference architectures consist of mechanisms and technology discussed in the answers to RQ1 and RQ2. In total five reference architectures were created which (mostly and theoretically) satisfied the requirements.

RQ6. Do the proposed reference architectures satisfy the requirements?

- **a. How to evaluate and validate the reference architectures?**
- **b. Which architecture fits the specified requirements?**

The reference architectures have been evaluated in a twofold approach. First, the five reference architectures were evaluated based on the answers to questionnaires with domain experts. This evaluation resulted in a selection of the most fitting reference architectures. The respondents thought the federated solutions were too complex for the use-case. Therefore, only the single cluster solutions were left in the selection. This selection has been used to conduct the second step of the evaluation, namely the creation and testing of non-functional requirements of two prototyped reference architectures. To test the prototypes, we created a testing strategy based on strategies available in scientific literature. Our testing strategy was validated by applying it on one of the prototypes which closely resembled a standard HA K8s cluster design. Both the testing strategy and the first reference architectures were evaluated through this step, after this the second architecture was evaluated using the testing strategy. The testing results gave insight into the conformity of the tested reference architectures on two requirements. Both reference architectures satisfied these tested requirements.

After this, the chosen reference architectures have been validated according to the list of requirements. One of the architectures satisfied all the requirements. The other architecture did not satisfy one of the requirements. We argued that the impact of this non-conformity to one of the requirements is manageable and mitigatable. The value of this specific architecture's advantages can outweigh the non-conformity to this requirement.

6.2 Contributions

This research has created several contributions to both practice and literature, the most important contributions are discussed in this section.

During the literature research several implementations of K8s systems were found, but none was found fitting the specific context of this research. For academics, this research has generated knowledge on what the important aspects are in designing HA K8s clusters for on-premise environments. Next to this, the strength of the availability mechanisms employed by K8s has been verified in a different environment.

For practice, two cluster designs containing a combination of software components have been created. These designs can be used as a basis for the mission-critical systems offered by vendors in the defence domain. Next to this, we have created insight into which parts of the cluster designs need additional engineering to better satisfy the requirements. The designs gave insights which help in the creation of fitting clusters.

6.3 Limitations

Some limitations were encountered during this research. The first limitation was due the specific domain in which this research was conducted. Due to the confidentiality requirements often found in the defence domain it was not possible to use confidential information without subjecting this report to the same confidentiality treatment. Therefore, the results of this research have not led to a definitive 'ready-to-go' design of a cluster, since testing with actual applications was not done. Further configuration is needed to use these designs in the intended domain.

The reference architectures have only been evaluated with experts in one organization. The reference architectures were mainly focused on a specific system. Hence, there is a real possibility of the results to be biased towards the needs of this system. Doing the validation at a different organization was not possible due to the confidential nature of the industry.

The prototyped clusters contained only a single application, whereas an actual implementation of such a cluster will host tenths (if not hundreds) of applications in many pods. Next to this, the tested application did not service any workloads during testing. This was done to keep the scope of the research manageable. Actual implementation with large numbers of pods could result in different behaviour or an unacceptable overhead imposed by the advanced monitoring. During the research several sources were found which indicated that the influence of a larger number of pods on the behaviour and expected overhead is manageable. Whether or not this is also true for the specific prototypes and the domain these are applied in was not tested. The prototypes created in this research were not tested on server grade hardware, but on a single development machine. We theorize that, because of the scalable design of K8s, this should not have a large impact on the suitability of the proposed reference architectures.

The number of respondents to the questionnaire and number of interviewees was quite low (both being four), due to the specific knowledge required to be regarded as expert on the subject.

Next to the low number of available experts, we also found a low amount of usable scientific literature. The found literature did not consider implementations of smaller scale K8s systems. We needed to bridge this gap, in addition to creating the reference architectures in such a way fitting to the mission-critical, on-premise nature of our context.

Some of the considered technologies had a very low maturity, which made it challenging to adequately address these technologies. In some cases, there was a lack in documentation, this was mitigated by using publicly available discussions between developers of these technologies. The possible future features (e.g., the hot-standby etcd learners) of a number of technologies are derived from these discussions, which offered interesting contributions to the understanding of the technology, but it is not conclusive information on the actual workings of the future features.

6.4 Future Research

Wieringa [7] describes the process of scaling up research, the goal of this process is to 'test an increasingly realistic model of the artefact under increasingly realistic conditions of practice'. The prototypes in this research have been tested on a single machine. The prototypes hosted a low amount of application that did not serve any real workloads. In order to create a higher certainty that the system will behave as intended a future research could deploy the prototypes on larger systems, with hardware which closer resembles the actual hardware used in the on-premise environment.

Next to this, the applicability of our architectures on on-premise systems in other domains could also be considered. For example, Kugele et al. [3], who containerized applications in automotive vehicles, offers such a domain. In this investigation the researchers did not consider an orchestration technology. It is likely that an orchestration technology which supports HA will be required in the future for these kinds of containerized applications in the automotive industry, especially when cars are becoming increasingly more autonomous. This increasing autonomy will most likely increase the availability requirements (because of the required safety) and increase complexity, due to the number of sensors and effectors that have to be managed. Other contexts such as aviation which also run on-premise systems can be relevant as well, if containerization is considered in these domains.

6.5 Recommendations

This research has created two designs which could potentially be used to support mission-critical on-premise systems in the defence domain. Both designs provide similar failover times and theoretically sufficient failover mechanism. The designs differ in one major aspect. The amount of required Availability Zones (AZs) to achieve HA is different for both designs, the first design (RA1) is a more standard HA K8s deployment using three AZs. The second design (RA2) uses only two AZs to achieve HA.

The amount of required AZs is important. The mission-critical system considered in this research currently uses two AZs. The nature of the on-premise context makes it challenging and costly to design it to support three AZs. Therefore, a careful consideration has to be done on which design would be the most appropriate, this section will give some recommendations.

Deploying RA1 should be a relatively uncomplicated process, since it is similar to widely deployed standard HA K8s clusters. As mentioned before, it does require a redesign of the on-premise context to support three AZs. RA1 utilizes the de facto standard for back-end storage for K8s (etcd), because the on-premise redesign allows for a quorum to be formed. Etcd is most likely to be the de facto standard for the foreseeable future. Using a de facto standard has the advantage of the relatively large community support this technology enjoys. The redesign allows for usage of other technologies using quorums, such

as specific storage technologies for the applications. The ability to form quorum also opens the possibility to redesign current applications in the system to support majority voting. To simplify this process a similar technology such as Koordinator ('Coordination as a Service') as described in Section 2.3 can be utilized, this allows applications to support quorum based leader election without having to implement such an algorithm for each application. We recommend choosing RA1 when redesigning of the on-premise context is not considered too costly.

Deploying RA2 does not require a redesign of the current on-premise context. It does however require some more unconventional cluster configuration. Etcd is replaced by a SQL-based storage technology (in this research PostgreSQL). To achieve this, an additional shim is needed, this shim translates read/write operations which would normally be send to etcd to read/write operations for SQL-based storage solutions. HA of this storage solution (and consequently the cluster) is not achieved through a quorum, but through a Master/Slave deployment of two SQL-based instances. Although the on-premise context does not have to be redesigned for RA2, it does introduce several new challenges. One of these challenges is the maturity of the shim (Kine in this research), additional development is required to render Kine a mature technology. Another (in-house) shim could be created, but this does also introduce additional development costs. An additional challenge for systems relying on SQL storage was mentioned by a questionnaire respondent, namely the challenge of managing the PostgreSQL instances in a Master/Slave configuration. PostgreSQL has been chosen because it offers more advanced Master/Slave functionalities than other found technologies. Configuring two PostgreSQL instances in a Master/Slave way is initially easy. One of the culprits of such systems is what happens if the Master fails, is restarted, and wants to rejoin the Master/Slave relation. The initial Slave has become the new Master and the old Master cannot rejoin as a Master, but needs to be reconfigured to be a Slave. PostgreSQL does not have a built-in feature for this, but open-source tools like Replication Manager for PostgreSQL Clusters (repmgr) [107] and Pgpool-II [108] exist to simplify this process. Still, failures of either the Master or the Slave have to be adequately detected (through tools like Keepalived or the ClusterLabs stack [109]). These tools are considered mature and suitable for the context. Configuring these tools requires careful consideration and is more complex than using the availability advantages inherent to quorum based storage. This design does not allow for other quorum based applications (storage or otherwise). RA2 is recommended when redesigning the on-premise context is deemed too costly or would complicate other parts of the on-premise context design too much.

We recommend potential implementers of our designs to consider the value that quorum based applications (other than etcd) in the cluster could bring. If storage for applications using quorum bears a lot of advantages, it could be enough reason to choose RA1 and redesign the on-premise context to support RA1. If sufficient expertise is in-house to correctly configure the additional components of RA2 and the costs of redesigning the on-premise context is deemed too high, it is recommended to contribute towards a more stable and mature shim. When this shim is regarded to be stable enough RA2 could be implemented in the on-premise context.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson and A. Rabkin, "Above the Clouds: A Berkeley View of Cloud Computing," 2009.

[2] "Cloud Native Computing Foundation," [Online]. Available: https://www.cncf.io/. [Accessed 20 June 2019].

[3] S. Kugele, D. Hettler and J. Peter, "Data-Centric Communication and Containerization for Future Automotive Software Architectures," in *2018 IEEE International Conference on Software Architecture (ICSA)*, Seattle, WA, USA, 2018.

[4] N. Dragoni, S. Dustdar, S. Larsen and M. Mazzara, "Microservices: Migration of a Mission Critical System," *CoRR,* vol. abs/1704.04173, 2017.

[5] "Kubernetes," Kubernetes, [Online]. Available: https://kubernetes.io/. [Accessed 18 June 2019].

[6] A. R. Hevner, S. T. March, J. Park and S. Ram, "Design Science in Information Systems Research," *MIS Quarterly,* vol. 28, no. 1, pp. 75-105, 2004.

[7] R. F. Wieringa, Design science methodology for information systems and software engineering, London: Springer, 2014.

[8] K. Peffers, T. Tuunanen, C. E. Gengler, M. Rossi, W. Hui, V. Virtanen and J. Bragge, "The design science research process: A model for producing and presenting information systems research," in *Proceedings of the First International Conference on Design Science Research in Information Systems and Technology (DESRIST 2006)*, Claremont, CA, USA, 2006.

[9] A. R. Hevner, "A Three Cycle View of Design Science Research," *Scandinavian Journal of Information Systems,* vol. 19, no. 2, 2007.

[10] M. Nabi, M. Toeroe and F. Khendek, "Availability in the cloud: State of the art," *Journal of Network and Computer Applications,* vol. 60, pp. 54-67, 2016.

[11] M. Toeroe and F. Tam, Service availability: principles and practice, John Wiley & Sons, 2012.

[12] A. Robertson, "Resource fencing using STONITH," August 2001. [Online]. Available: http://mirrors.sinuspl.net/www.linux-ha.org/heartbeat/ResourceFencing_Stonith.pdf. [Accessed 30 July 2019].

[13] K. Dooley, in *Designing Large-scale LANs*, O'Reilly, 2002, p. 31.

[14] "Fencing," 2010 January 2010. [Online]. Available: http://linux-ha.org/wiki/Fencing. [Accessed 30 April 2019].

[15] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing,* pp. 81-84, 2014.

[16] A. M. Joy, "Performance comparison between Linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015.

[17] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon and B.-J. Kim, "Performance comparison analysis of linux container and virtual machine for building cloud.," in *Advanced Science and Technology Letters*, 2014.

[18] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," *IEEE Fifth International Conference on Cloud Computing,* pp. 423-430, 2012.

[19] K. Agarwal, B. Jain and D. E. Porter, "Containing the Hype," in *APSys '15 Proceedings of the 6th Asia-Pacific Workshop on Systems*, Tokyo, 2015.

[20] S. Mazaheri, Y. Chen, E. Hojati and A. Sill, "Cloud benchmarking in bare-metal, virtualized, and containerized execution environments," in *2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS)*, Beijing, 2016.

[21] Kubernetes, "Concepts," [Online]. Available: https://kubernetes.io/docs/concepts/. [Accessed 25 April 2019].

[22] Kubernetes, "Kubernetes Components," Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/components/. [Accessed 25 April 2019].

[23] "Set up High-Availability Kubernetes Masters," [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/highly-available-master/. [Accessed 30 April 2019].

[24] A. Chen and D. Tornow, "Kubernetes High Availability," Medium, 19 October 2018. [Online]. Available: https://medium.com/@dominik.tornow/kubernetes-high-availability-d2c9cbbdd864. [Accessed 25 April 2019].

[25] "Kubernetes," [Online]. Available: https://github.com/kubernetes/kubernetes. [Accessed 21 November 2019].

[26] "Operating etcd clusters for Kubernetes," Kubernetes, [Online]. Available: https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/. [Accessed 8 May 2019].

[27] "Raft," [Online]. Available: https://raft.github.io/. [Accessed 6 Januari 2020].

[28] J. Jackson, "Kubernetes High Availability: No Single Point of Failure," The New Stack, 10 April 2018. [Online]. Available: https://thenewstack.io/kubernetes-high-availability-no-single-point-of-failure/. [Accessed 25 April 2019].

[29] A. Kanso, H. Huang and A. Gherbi, "Can Linux Containers Clustering Solutions offer High Availability?," in *IEEE Workshop on Containers (WoC)*, 2016.

[30] Object Management Group, March 2015. [Online]. Available: https://www.omg.org/spec/UML/2.5/PDF. [Accessed 25 July 2019].

[31] M. Bi, "Deep dive into Kubernetes Simple Leader Election," Medium, 16 January 2019. [Online]. Available: https://medium.com/michaelbi-22303/deep-dive-into-kubernetes-simple-leader-election-3712a8be3a99. [Accessed 6 May 2019].

[32] "Split Brain," Linux-ha, 29 January 2010. [Online]. Available: http://linux-ha.org/wiki/Split_Brain. [Accessed 30 April 2019].

[33] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned," in *IEEE International Conference on Cloud Computing, CLOUD*, 2018.

[34] L. A. Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Kubernetes as an Availability Manager for Microservice Applications," *Journal of Network and Computer Applications,* 2019.

[35] C. Oliveira, L. C. Lung, H. Netto and L. Rech, "Evaluating Raft in Docker on Kubernetes," in *International Conference on Systems Science*, 2017.

[36] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz and L. M. Sa de Souza, "State machine replication in containers managed by Kubernetes," *Journal of Systems Architecture,* pp. 53-59, 2017.

[37] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech and C. P. Oliveira, "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*, Natal, 2018.

[38] "Kubernetes Cluster Federation," [Online]. Available: https://github.com/kubernetes-sigs/kubefed. [Accessed 04 November 2019].

[39] "multicluster-scheduler," admiraltyio, [Online]. Available: https://github.com/admiraltyio/multicluster-scheduler. [Accessed 1 May 2019].

[40] B. Tabbara, I. Chekrygin and J. Watts, "Introducing Crossplane," Crossplane.io, 4 December 2018. [Online]. Available: https://docs.google.com/document/d/1whncqdUeU2cATGEJhHvzXWC9xdK29Er45NJeoemxebo/edit?usp=sharing. [Accessed 6 May 2019].

[41] "GoogleCloudPlatform/k8s-multicluster-ingress," [Online]. Available: https://github.com/GoogleCloudPlatform/k8s-multicluster-ingress. [Accessed 6 May 2019].

[42] "Deep Dive into Cilium Multi-cluster," Cilium, 18 March 2019. [Online]. Available: https://cilium.io/blog/2019/03/12/clustermesh/. [Accessed 06 May 2019].

[43] "rancher/submariner," rancher, [Online]. Available: https://github.com/rancher/submariner. [Accessed 28 May 2019].

[44] "How Istio works," IBM, [Online]. Available: https://developer.ibm.com/courses-center/how-istio-works-dwc024/. [Accessed 23 May 2019].

[45] "What is Istio," Istio, [Online]. Available: https://istio.io/docs/concepts/what-is-istio/. [Accessed 23 May 2019].

[46] "Enable LeaderElect for federation controller #402," 7 November 2018. [Online]. Available: https://github.com/kubernetes-sigs/federation-v2/issues/402. [Accessed 30 April 2019].

[47] "federation controller disaster recovery Support #636," 7 03 2019. [Online]. Available: https://github.com/kubernetes-sigs/federation-v2/issues/636. [Accessed 30 April 2019].

[48] "Preserve local cluster changes to status, annotations and labels #633," [Online]. Available: https://github.com/kubernetes-sigs/kubefed/issues/633. [Accessed 17 May 2019].

[49] "Kubernetes Federation v2 Prototype," August 2018. [Online]. Available: https://github.com/gyliu513/federation-v2/tree/pull. [Accessed 30 April 2019].

[50] "question: disaster recovery strategy with federation v2 #450," 26 11 2018. [Online]. Available: https://github.com/kubernetes-sigs/federation-v2/issues/450. [Accessed 30 04 2019].

[51] "Velero," 2019. [Online]. Available: https://github.com/heptio/velero. [Accessed 30 April 2019].

[52] "v1beta1 Milestone," [Online]. Available: https://github.com/kubernetes-sigs/federation-v2/milestone/4. [Accessed 30 April 2019].

[53] "Pull and Push mode in Kubernetes Federation V2," 2018. [Online]. Available: https://groups.google.com/forum/#!topic/kubernetes-sig-multicluster/nI28ORF-j5M. [Accessed 30 April 2019].

[54] "Add leader election to controller-manager," [Online]. Available: https://github.com/kubernetes-sigs/kubefed/pull/632. [Accessed 20 June 2019].

[55] T. Goldschmidt and S. Hauck-Stattelmann, "Software containers for industrial control.," in *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Limassol, Cyprus , 2016.

[56] S. Kho Lin, U. Altaf, G. Jayaputera, J. Li, D. Marques, D. Meggyesy, S. Sarwar, A. Novak, W. Voorsluys, R. Sinnott, V. Nguyen and K. Pash, "Auto-Scaling a Defence Application across the Cloud Using Docker and Kubernetes," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018.

[57] D. Morris, S. Voutsinas, N. Hambly and R. Mann, "Use of Docker for deployment and testing of

astronomy software," *Astronomy and Computing,* vol. 20, pp. 105-119, 2017.

[58] "Kubernetes User Case Studies," The Linux Foundation, [Online]. Available: https://kubernetes.io/case-studies/. [Accessed 21 October 2019].

[59] "CASE STUDY: CERN," The Linux Foundation, [Online]. Available: https://kubernetes.io/case-studies/cern/. [Accessed 21 October 2019].

[60] K. Beck and M. Beedle, "Manifesto for Agile Software Development," 2001.

[61] T. Dingsøyr, S. Nerur, V. Balijepally and N. B. Moe, "A decade of agile methodologies: Towards explaining agile software development," *Journal of Systems and Software,* vol. 85, no. 6, pp. 1213-1221, 2012.

[62] A. R. Manu, J. K. Patel and S. Akhtar, "A Study, Analysis and deep dive on Cloud PAAS security in terms of Docker Container security," in *2016 International Conference on Circuit, Power and Computing Technologies [ICCPCT]*, 2016.

[63] "Automated Cloud Infrastructure, Continuous Integration and Continuous Delivery using Docker with Robust Container Security," in *2019 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, 2019.

[64] "Istio," [Online]. Available: https://istio.io/. [Accessed 10 December 2019].

[65] "Gluster," [Online]. Available: https://www.gluster.org/. [Accessed 04 September 2019].

[66] "Split brain and the ways to deal with it," Gluster, [Online]. Available: https://docs.gluster.org/en/latest/Administrator%20Guide/Split%20brain%20and%20ways%20to%20deal%20with%20it/. [Accessed 4 September 2019].

[67] "Hekiti," [Online]. Available: https://github.com/heketi/heketi. [Accessed 10 September 2019].

[68] "Ceph," [Online]. Available: https://ceph.io/. [Accessed 11 September 2019].

[69] "HAProxy," [Online]. Available: www.haproxy.org. [Accessed 5 September 2019].

[70] [Online]. Available: http://nginx.org/en/CHANGES. [Accessed 28 October 2019].

[71] "Changelog," [Online]. Available: https://www.keepalived.org/changelog.html. [Accessed 28 October 2019].

[72] "Chapter 2. Keepalived Overview," Red Hat, [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/load_balancer_administration/ch-keepalived-overview-vsa. [Accessed 28 October 2019].

[73] R. van Elst, “Simple keepalived failover setup on Ubuntu 14.04,” 13 June 2014. [Online]. Available: https://raymii.org/s/tutorials/Keepalived-Simple-IP-failover-on-Ubuntu.html. [Accessed 10 December 2019].

[74] “Active-Passive HA for NGINX Plus on AWS Using Elastic IP Addresses,” Nginx, [Online]. Available: https://docs.nginx.com/nginx/deployment-guides/amazon-web-services/high-availability-keepalived/. [Accessed 10 December 2019].

[75] H. Handoko, S. M. Isa, S. Si and M. Kom., “High Availability Analysis with Database Cluster, Load Balancer and Virtual Router Redudancy Protocol,” in *2018 3rd International Conference on Computer and Communication Systems (ICCCS)*, Nagoya, Japan, 2018.

[76] B. Horowitz, C.-H. Yu and L. Deligiannidis, “Canarycurity: A Robust Security Surveillance System Built with Mobile and IoT Technologies,” in *Int'l Conf. Internet Computing and Internet of Things | ICOMP'19* , Athens, 2019.

[77] S. Brenner, B. Garbers and R. Kapitza, “Adaptive and Scalable High Availability for Infrastructure Clouds,” in *IFIP International Conference on Distributed Applications and Interoperable Systems.* , Berlin, 2014.

[78] E. Kassela, I. Konstantinou and N. Koziris, “A Generic Architecture for Scalable and Highly Available Content Serving Applications in the Cloud,” in *2015 IEEE 4th Symposium on Network Cloud Computing and Applications*, Munich, Germany, 2015.

[79] S. W. Ambler, “Just Barely Good Enough Models and Documents: An Agile Core Practice,” [Online]. Available: agilemodeling.com/essays/barelyGoodEnough.html. [Accessed 11 September 2019].

[80] “Pod startup latency SLI/SLO details,” [Online]. Available: https://github.com/kubernetes/community/blob/master/sig-scalability/slos/pod_startup_latency.md. [Accessed 6 September 2019].

[81] “Building large clusters,” Kubernetes, [Online]. Available: https://kubernetes.io/docs/setup/best-practices/cluster-large/. [Accessed 6 September 2019].

[82] “Keepalived,” [Online]. Available: https://www.keepalived.org/. [Accessed 5 September 2019].

[83] “How To Set Up Highly Available Web Servers with Keepalived and Floating IPs on Ubuntu 16.04,” Vexxhost, [Online]. Available: https://vexxhost.com/resources/tutorials/highly-available-web-servers-keepalived-floating-ips-ubuntu-16-04/. [Accessed 11 September 2019].

[84] I. Cicimov, “Kubernetes cluster step-by-step: Kube-apiserver with Keepalived and HAProxy for HA,” 16 June 2017. [Online]. Available: https://icicimov.github.io/blog/kubernetes/Kubernetes-cluster-step-by-step-Part5/. [Accessed 11 September 2019].

[85] J. Ellingwood, “How To Set Up Highly Available Web Servers with Keepalived and Floating IPs on Ubuntu 14.04,” 20 October 2015. [Online]. Available: https://www.digitalocean.com/community/tutorials/how-to-set-up-highly-available-web-servers-

with-keepalived-and-floating-ips-on-ubuntu-14-04. [Accessed 11 September 2019].

[86] "System and Environment Requirements," Red Hat, [Online]. Available: https://docs.openshift.com/container-platform/3.11/install/prerequisites.html. [Accessed 19 December 2019].

[87] "Creating Highly Available clusters with kubeadm," [Online]. Available: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/#before-you-begin. [Accessed 17 September 2019].

[88] "Hardware requirements and recommendations," [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SSBS6K_3.1.2/supported_system_config/hardware_reqs.html. [Accessed 17 September 2019].

[89] "Hardware recommendations," [Online]. Available: https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/hardware.md#hardware-recommendations. [Accessed 17 September 2019].

[90] "Operating System and Hardware Requirements," HAProxy, [Online]. Available: https://www.haproxy.com/documentation/hapee/1-5r2/getting-started/os-hardware/. [Accessed 17 September 2019].

[91] "Kine," Rancher, [Online]. Available: https://github.com/ibuildthecloud/kine/. [Accessed 11 September 2019].

[92] "PostgreSQL: The World's Most Advanced Open Source Relational Database," The PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/. [Accessed 11 September 2019].

[93] "MySQL," [Online]. Available: https://www.mysql.com/. [Accessed 11 September 2019].

[94] "SQLite," [Online]. Available: https://www.sqlite.org/index.html. [Accessed 12 September 2019].

[95] "[Feature] Support generic SQL-family storage backend," 13 June 2019. [Online]. Available: https://github.com/kubernetes-sigs/apiserver-builder-alpha/issues/362. [Accessed 11 September 2019].

[96] "Chapter 25. High Availability, Load Balancing, and Replication," The PostgreSQL Global Development Group, [Online]. Available: https://www.postgresql.org/docs/9.2/high-availability.html. [Accessed 11 September 2019].

[97] "Chapter 17 Replication," Oracle, [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/replication.html. [Accessed 11 September 2019].

[98] "etcd learner design," [Online]. Available: https://github.com/etcd-io/etcd/blob/master/Documentation/learning/design-learner.md. [Accessed 16 September 2019].

[99] "Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models," International Organizationfor Standardization, Geneva, 2011.

[100] M. Haoues, A. Sellami, H. Ben-Abdallah and L. Cheikhi, "A guideline for software architecture selection based on ISO 25010 quality related characteristics," *International Journal of System Assurance Engineering and Management,* vol. 8, no. 2, pp. 886-909, 2017.

[101] "Kubespray," [Online]. Available: https://github.com/kubernetes-sigs/kubespray. [Accessed 16 October 2019].

[102] "K3s," Rancher, [Online]. Available: https://k3s.io/. [Accessed 11 September 2019].

[103] "leaderelection.go," [Online]. Available: https://github.com/kubernetes/client-go/blob/master/tools/leaderelection/leaderelection.go. [Accessed 8 January 2020].

[104] "Prometheus," [Online]. Available: https://prometheus.io/. [Accessed 21 June 2019].

[105] "Best practice prometheus monitoring," [Online]. Available: https://github.com/rancher/k3s/issues/425. [Accessed 8 January 2020].

[106] "Prometheus scraping configuratio," [Online]. Available: https://github.com/rancher/k3s/issues/1244. [Accessed 8 January 2020].

[107] "Replication Manager for PostgreSQL clusters," [Online]. Available: https://repmgr.org/. [Accessed 4 December 2019].

[108] "Pgpool Wiki," [Online]. Available: https://www.pgpool.net/mediawiki/index.php/Main_Page. [Accessed 13 December 2019].

[109] "ClusterLabs," [Online]. Available: https://clusterlabs.org/. [Accessed 04 December 2019].

THALES

Goedkeuring Stage-/Afstudeerverslag van:

Max Riesewijk

Titel verslag: High availability Orchestration OF Linux containers

Opleidingsinstelling: Universiteit Twente

Stage-/Afstudeerperiode: 01-04-2019 / 31-01-2020

Vestiging/Afdeling: Hengelo Platform

Stagebegeleider Thales: Gezzit Binnenmars

Dit verslag (zowel de papieren als de elektronische versie) is door de begeleider van Thales Nederland B.V. gelezen en becommentarieerd. Hierbij heeft de begeleider de inhoud beoordeeld en gelet op de gevoeligheid daarvan, evenals die van daarin opgenomen gegevens zoals plattegronden, technische specificaties, commercieel vertrouwelijke informatie en organisatieschema's waarin namen staan vermeld. De begeleider heeft op basis daarvan het volgende besloten:

- ● Dit verslag is **openbaar (Open).** Een eventuele verdediging kan openbaar plaatsvinden en het verslag kan worden opgenomen in openbare bibliotheken en/of worden gepubliceerd in kennisbanken.

- ○ Dit verslag en/of een samenvatting hiervan, is **beperkt openbaar (Thales Group Internal)**. Het zal uitsluitend door docenten en indien nodig door leden van de examencommissie of visitatiecommissie worden gelezen en beoordeeld. De inhoud zal vertrouwelijk worden behandeld en niet worden verspreid door middel van publicatie of opname in openbare bibliotheken en/of kennisbanken. Digitale bestanden worden onmiddellijk na het afstuderen verwijderd van persoonlijke IT middelen, tenzij de stagiair expliciet toestemming heeft verkregen om deze bestanden (geheel of gedeeltelijk) te behouden. Een eventuele verdediging van de scriptie kan **beperkt openbaar** plaatsvinden. Uitsluitend familieleden tot en met de eerste graad, en docenten van de vakgroep .....*<naam vakgroep>* mogen de verdediging bijwonen.

- ○ Dit verslag en/of een samenvatting hiervan, is **niet openbaar (Thales Group Confidential)**. Het zal uitsluitend door de begeleider binnen de universiteit/hogeschool, eventueel door een tweede lezer en indien nodig door leden van de examencommissie of visitatiecommissie worden gelezen en beoordeeld. De inhoud zal vertrouwelijk worden behandeld en op geen enkele wijze worden verspreid. Het verslag wordt niet gepubliceerd of opgenomen in openbare bibliotheken en/of kennisbanken. Digitale bestanden worden onmiddellijk na het afstuderen verwijderd van persoonlijke IT middelen. Een eventuele verdediging van de scriptie dient **besloten** plaats te vinden, d.w.z. uitsluitend in aanwezigheid van stagiair, stagebegeleider(s) en beoordelaars. In voorkomende gevallen moet een aangepast verslag voor de opleidingsinstelling worden gemaakt.

THALES

Akkoord:

G. BINNENMARS
(Stagebegeleider Thales)

Akkoord:

(Opleidingsinstelling)

Hengelo - 27-01-2020
(plaats/datum)

(kopie security)

THALES