Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Investigating the Orchestration of Containerized Enterprise Content Management Workloads in Cloud Environments Using Open Source Cloud Technology Based on Kubernetes and Docker

Christoph Trybek

Course of Study:	Master of Science Informatik
Examiner:	Prof. Dr.-Ing. Bernhard Mitschang
Supervisor:	Dipl.-Phys. Cataldo Mega
Commenced:	February 1, 2021
Completed:	September 1, 2021

Abstract

Due to the mass adaption of the paperless office and mobile devices organizations have to deal with a growing amount of information. Additionally the continuous advancements of virtualization technologies and network bandwidth allowed numerous vendors to offer Enterprise Content Management applications on Cloud infrastructures. Those offerings enabled enterprises to exploit the potential of structured, semi-structured and unstructured information without maintaining the required infrastructure and staff. The following thesis aims to further improve the utilization of shared computing resources by integrating containerized ECM components into a Kubernetes cluster. The proposed system topology is then implemented prototypically to verify the introduced concept. The major challenge encountered during this thesis is the management of stateful database applications within a Kubernetes cluster.

Contents

1 Introduction **11**

2 Foundations **13**
2.1 Enterprise Content Management 13
2.2 Cloud Computing 14
2.3 Virtualization 14
2.4 Containerization 15
2.5 Microservices 15
2.6 Orchestrating Containers within a Microservice Architecture 17
2.7 OpenStack 18
2.8 Docker 20
2.9 Kubernetes 21

3 Related Work **25**
3.1 About the Design Changes Required for Enabling ECM Systems to Exploit Cloud Technology 25

4 Concept **27**
4.1 Implications of a Kubernetes Cluster 27
4.2 Aspired ECM System Topology 27

5 Prototype **31**
5.1 Infrastructure 31
5.2 Kubernetes Components 33
5.3 Initially Aspired System Topology 36
5.4 Refactored ECM System Topology 37
5.5 Implementation Details of the Kubernetes Components 39
5.6 Source Code 42

6 Conclusion and Outlook **45**

Bibliography **47**

List of Figures

2.1 *Virtual Machines* compared to Operation System Level and Application Level *Containers* [Kum17] 16
2.2 The Different Layers of Container Orchestration [JBB+19] 18

3.1 Proof of Concept by Shao [Sha20] 26

4.1 Data Allocation in Local Filesystems of Host Servers in a Cluster 28
4.2 Initial Cluster Topology 29

5.1 Workaround to Enable Incoming Traffic to the KiND Cluster 32
5.2 The Interactions of the Components Inside a Kubernetes Cluster [TKA20] 36
5.3 Initial Topology of the ECM System Inside a Kubernetes Cluster 37
5.4 Improved Topology of the ECM System Inside a Kubernetes Cluster 38

List of Listings

5.1 KiND Cluster Configuration File 32
5.2 Data Catalog Database *Service* Configuration File 39
5.3 Data Catalog Database *Endpoint* Configuration File 40
5.4 Resource Manager Application *Nodeport* Configuration File 40
5.5 Resource Manager Application *Deployment* Configuration File 43
5.6 Resource Manager Application *Deployment entrypoint.sh* script 44

1 Introduction

Large scale organisations have to deal with continuously increasing amounts of structured, semi-structured or unstructured information due to the progress in digitalization. Especially semi-structured data like business correspondences and multi media content have seen a rapid growth driven by trends like the paperless office and the large adaption of mobile devices. Additionally documents that support or result from essential business processes need to be stored in an audit-compliant manner for various regulatory reasons. To utilize the relevant information contained in documents, emails or media files throughout their whole life cycle, organizations rely on Enterprise Content Management Systems. Those systems are typically deployed as a monolithic applications on an on-premise infrastructure with a long-running update cycle which oftentimes requires a dedicated team within the IT department. To deliver the value creation that ECM systems generate to smaller enterprises which can not afford a large IT-Team or are unable to acquire the necessary talent various vendors launched Enterprise Content Management on Cloud infrastructures. This offer could only be facilitated through the continuous advancement of virtualization, containerization and orchestration technologies as well as the rapid growth of network bandwidth.

To further optimize the utilization of the infrastructure of ECM on Cloud components based on truly occurring workloads the following thesis aims to integrate a decomposed and containerized ECM application into a cluster running on a cloud environment orchestrated by Kubernetes. The conducted investigation focused on finding a feasible system topology that provides a stable and reliable environment for organizations to store their business critical data.

2 Foundations

The following chapter describes the necessary foundations of this work. It examines the concepts of enterprise content management, cloud computing, microservices, container virtualisation technologies and container orchestration systems.

2.1 Enterprise Content Management

Enterprise Content Management or short ECM is defined as a composition of strategies, processes, methods, tools and technologies that are required to manage structured, semi structured or unstructured information within or between organizations. The AIIM [AIIM13] and Grahlmann et al. [GHH+12] define the following essential functions of an ECM system:

Capture
Content can be accumulated by humans or applications like optical character recognition. This function handles the insertion of the gathered data into an ECM system. To make this information usable it requires pre-processing, categorization and indexation to create a structured format.

Manage
The management of content is concerned with its administration within the ECM system. It governs the related meta data as well as editing control and version control. Editing control describes the process of checking out a document, modify it and check it back into the system. The history modification is recorded in version control.

Store
Keeps content and documents available in a short-term timescale using storage technologies with a low latency like local data systems. This data is usually used on a daily basis therefore a fast and comfortable access is required.

Preserve
This function handles the long-term archiving of content which is not accessed frequently and is kept for regulatory and compliance reasons. It is important for the ECM system to store this data in a revision-safe manner.

Deliver
This feature takes care of the distribution of the *stored* or *preserved* content to a human operator. The information can be delivered actively through search and download or email as well as passively via internet or intranet.

The following Enterprise Content Management system components are required to enable the previously discussed functions that are considered within this thesis:

Data Catalog
Every piece of information within the ECM system is represented in this central component. It does not contain the content itself but the corresponding meta data and index in a relational database schema. The given structure of the schema can be extended depending on the business requirements of an enterprise. The *Data Catalog* is used to deliver fast search results to the end-user while enforcing defined content access policies.

Object Catalog
The *Object Catalog* contains all information about stored objects inside an ECM system that are needed for their retrieval like file size, logical path etc.

Resource Manager
This component manages the storage and distribution of content and interacts with the *Data Catalog* in regards of the associated meta data as well as the *Object Catalog* in relation to storage information. It handles the storage of content and all its revisions in a filesystem by utilizing its APIs.

Client Application
It serves the user web interface of the ECM system and provides all functionalities required by a human operator to utilize the previously discussed services.

2.2 Cloud Computing

The rapid advance in development of processing power and connection bandwidth facilitated the emergence of a new computing paradigm which is known as cloud computing or short cloud. This new possibility enabled companies to smoothly develop and manage own Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) platform offerings [Aa10]. SaaS is seen as the application layer on which a consumer can access a service hosted on the infrastructure of a vendor, with PaaS the customer has a direct access to the low level platform layer like operating systems and middleware abstraction. IaaS allows customers to utilize the infrastructure of the provider in an abstracted way and manner. The flexibility of cloud computing offerings enables other companies to launch software products without an extensive capacity planing step and resource allocation. It further reduces the burden of maintaining physical hardware components and hiring skilled professionals on-premise during the life cycle of a service. Additionally the pay per use billing and the minimal management principle allows service providers to adjust computing resources automatically to the demand of its applications [MG11].

2.3 Virtualization

To allow the introduced concepts the technology of Virtualization plays a key role. It is described as the abstraction of physical components into logical objects to obtain a greater utility of the resources. Virtualizing a computer by creating a *Virtual Machine* allows the access hardware resources like processors, memory, storage and network interfaces as logical objects. Those objects are managed and monitored through a software called *Hypervisor* which is a layer between the physical hardware and the logically abstracted virtual objects. The *Hypervisor* exposes only a subset of the available

physical resources to each *Virtual Machine* on a host server and acts as an I/O interface between those two layers. The various layers of a *Virtual Machine* are illustrated in Figure 2.1. Since hardware got more powerful and efficient the paradigm of "One Server One Application" led to underutilizing computing resources. The consolidation of computing resources through virtualization allowed to lower operation and maintenance costs, power consumption as well as the overall footprint of large data centers without mitigating the quality of the provided services. Another advantage compared to physical hardware is that a *VM* is basically a set of files and can be moved effortlessly between servers. Therefore a solely virtual environment enables organizations to rely on higher degrees of availability, flexibility and maintainability of their software systems [Por16].

2.4 Containerization

Operating *Virtual Machines* can still lead to underutilized computing resources especially for small applications. This is because each *VM* contains its own copy of a operation system as well as a virtual copy of the hardware resources. To minimize the described overhead of *Virtual Machines* the concept of *Containers* was introduced. A *Container* consists of a set of separated processes and all required dependencies of an application that can be operated independently of the host system. The containerization of applications is possible by leveraging various techniques like *cgroups*, *namespaces* and *rootfs* from the Linux kernel to create an isolated sandbox on a host machine. This allows for an abstraction on the operation system level in contrast to *VMs* which abstract the hardware level. Therefore it is necessary for all containerised applications to run the same operation system whereas virtual machines are able to support multiple operation systems side by side. There are two kinds of containers based on the level of shared host resources. Figure 2.1 illustrates the different types of *Containers* in contrast to a *Virtual Machine*.

Operation System Level Container
Encapsulates its own operation system while sharing the kernel with other *Containers* on the host.

Application Level Container
Incorporates all processes an application needs to run while sharing the operation system with other *Containers* on the host.

Containerization enables many advantages over traditional virtualization. Since *Containers* are much more lightweight than *Virtual Machines* it is possible to operate more containers on a computing resource. Additionally the initialisation is almost instant compared to the protracted booting process of a *Virtual Machine*. Further *Containers* are much more portable because all libraries and dependencies are encapsulated and can be operated regardless of the underlying operating system of the host machine [Kum17; Por16; Pou20a].

2.5 Microservices

The growing popularity of small applications that are packaged in a single container led to a new paradigm in software architecture. Within a typical *Microservice* based architecture there exist many standalone services which collaborate through one or more network endpoints to reach a

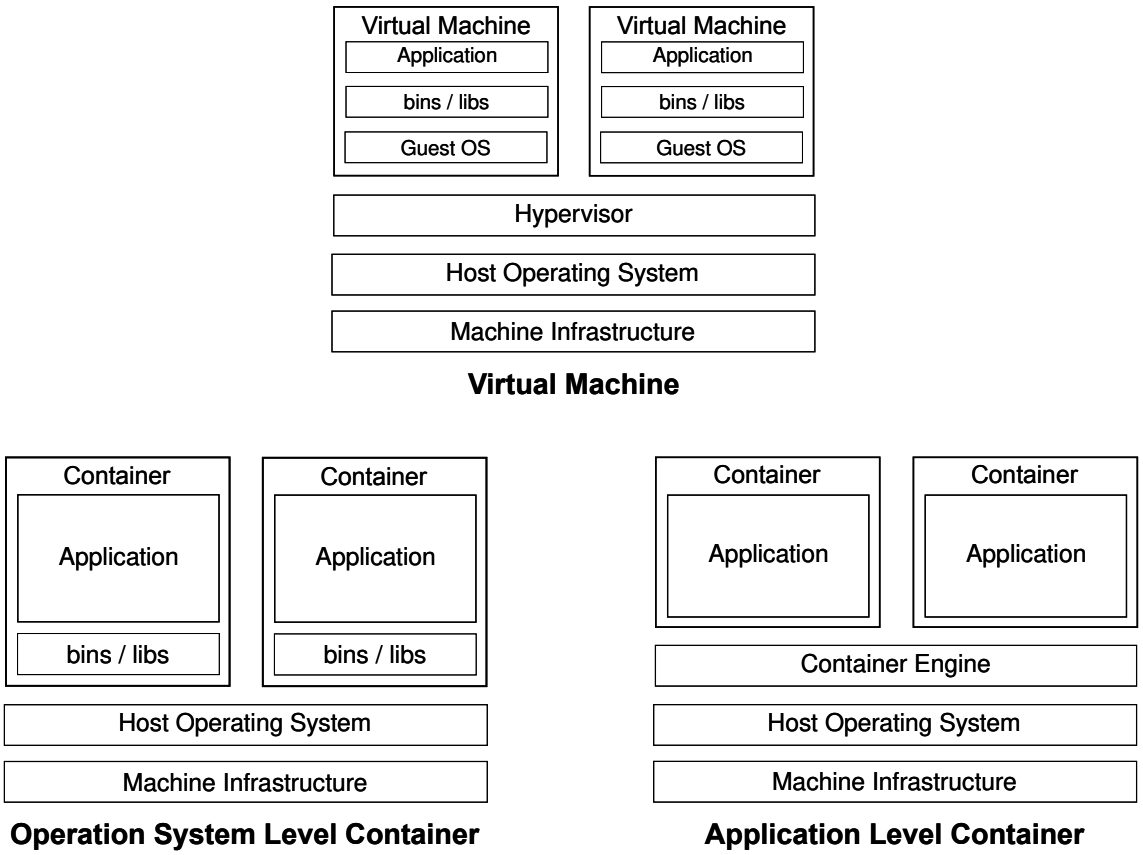


Figure 2.1: *Virtual Machines* compared to Operation System Level and Application Level *Containers* [Kum17]

defined business goal together. The key characteristic of those services is that each is independently deployable, modeled around a business domain and technology agnostic. That means that every *Microservice* can be implemented with its own technology, data structure and even programming language. Further while deploying changes of a *Microservice* into production it should not be necessary to interact with other services. To ensure this property all services need to be *loosely coupled* which means that each component of a system can be changed or swapped independently. This can be achieved through stable, explicit and well defined interfaces. Additionally the borders of each service are not defined by expert groups of certain fields like backend, frontend or databases but by business domain. That means that there is a team with multiple skills in a company which develops a service that handles all details to reach one business goal. For example a Team which has responsibility of a *Microservice* which handles the payments of a *SaaS* application. The team implements and maintains all service components that the customer interacts with while conducting the payment process. In this way the service contains a small part of user interface, a small part of application logic and a small part of data storage. This principle reduces changes in different layers of an application across multiple teams and therefore enables organizations to ship changes and updates faster. Another principle of *Microservices* is the ownership of the data it operates with and therefore no indirect access of information through sharing databases. That means that whenever a service needs data owned by another service it needs to explicitly ask the other service over a communication channel to pass on the desired information. That concept facilitates a service to

decide which data is shared with which external services. It further allows to hide implementation details which can change for arbitrary reasons behind exposed stable service interfaces. This brings the benefit of independently deployable *Microservices* which do not induce adjustments to multiple services whenever a service changes its internals.

One of the key advantages of a *Microservice* architecture is the independence of the various teams of a organization. It allows teams to choose the technology which is best for achieving one specific business goal and find the optimal composition of programming languages, frameworks, databases and many more. Further the independent deployability enables applications with a higher flexibility, availability, resiliency and scalability.

There are also potential pitfalls of using *Microservices*. One is that they depend on reliable network communication which is inherently slower than on a monolithic system. Additionally varying latencies, packet loss and random connectivity issues can make the behavior of a whole *Microservice* architecture unpredictable. Another organisational pitfall is switching to *Microservices* without having thought about the necessary groundwork. The most important considerations are containerisation of applications, continuous integration and continuous delivery. This fundamental automation prevents the organisation from ending up with many "mini monoliths"that need manual maintenance Nevertheless managing a large amount of *Microservices* across different cloud environments is still a complex task even when automated [New19; RDG20].

2.6 Orchestrating Containers within a Microservice Architecture

Enterprise-level applications utilizing the *Microservice* architecture are oftentimes made up of hundreds of containerized services that need to be orchestrated. To provide a real customer value while optimizing provider costs this cluster of *Containers* needs to be reliable and potentially globally available while making ideal use of computational resources. To achieve this goal an orchestration engine aggregates a set of server hosts with its network connections into a single resource pool called cluster. The engine autonomously deploys *Containers*, schedules them across the cluster while scaling its number proportionally to the currently accruing workload to ensure defined policies and service level agreements are met [Kha17; RDG20]. Figure 2.2 illustrates that the orchestrator serves as foundation for the application layer and typically consists of three layers which sit on top of the hardware, operating system and container runtime layers [JBB+19]:

Resource Management
This layer takes care of the low level resources like computation, data storage and network. Its goal is to maximize the utilization of those resources while preventing conflicts by competing *Containers*.

Scheduling
This layer is concerned with the efficient usage of cluster internal resources. It gets its service requirements through configuration files supplied by an administrator. It is then decided where the application *Containers* are placed in the cluster with respect to the number of needed replicas and co-location constraints to leverage the full potential of inter-process communication. It is further responsible to ensure that a *Container* is running and therefore

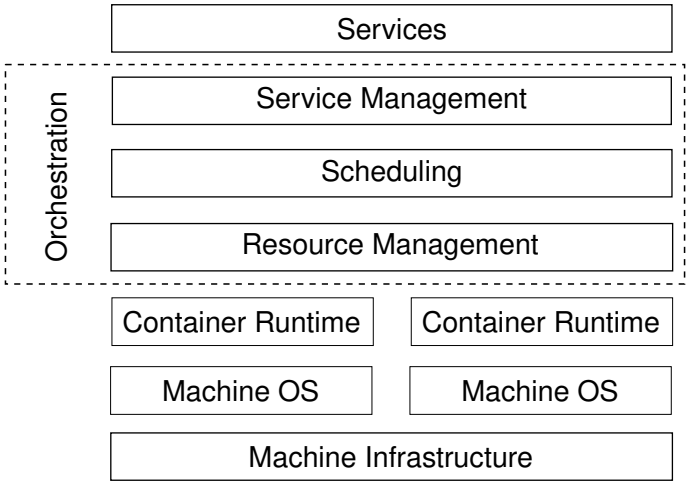


Figure 2.2: The Different Layers of Container Orchestration [JBB+19]

constantly checking their availability and if necessary restarting crashed *Containers*, move *Containers* from failed nodes as well as scaling up the number of *Containers* to deal with increasing workloads.

Service Management
The final layer allows administrators to define and manage the high level aspects of the underlying cluster like attaching meta data to *Containers* and divide incoming traffic to balance workloads. Additionally it allows *Container* isolation through the specification of *namespaces*. This allows the cluster to be used by multiple tenants without interference.

2.7 OpenStack

The following section describes *OpenStack* as Infrastructure as a Service solution which is needed as the groundwork for a scalable cloud environment. It is an open source *IaaS* platform written in python and developed by the National Aeronautics and Space Administration and RackSpace and published in 2010. The OpenStack architecture consists of three main components: compute, image and storage. There are many additional components in the OpenStack ecosystem which are out of scope for this thesis but some important will be mentioned briefly [ABG+15; RB14]:

Compute
The *Nova* component contains all tools to manage virtual machines on physical computing nodes and is used to administrate *IaaS Clouds*. It provides an external communication channel to applications or administrators through an API server. It handles the orchestration and life cycle of instances by creating and managing virtual servers without a strict dependency on a *Hypervisor*. Additionally it includes tools to manage networks as well as access control and serves as an essential building block for a basic *Cloud Computing* implementation.

Image
This component also known as *Glance* serves as a central catalog for *VM* images. It allows discovery, preservation and retrieval of images based on metadata.

Object
Swift is a redundant and scalable component to manage an object store. It provides users with storage capacity in a highly distributed architecture to prevent data loss through single points of failure.

Additional Components

Horizon: Is a dashboard to monitor, manage and provision services in *OpenStack*.

Keystone: Supplies an authentication service to apply tokens and policies to users and service interactions. It is an essential service to provide *Cloud* services to end users.

Cinder: Enables persistent volumes to virtual machines and was encompassed in *Nova* in previous releases. It uses the *Swift* component as backup for the supplied persistent volumes.

Neutron: Is a service which provides network connectivity and allows the configuration of advanced network topologies and policies.

2.7.1 Alternative Infrastructure-as-a-Service Technologies

The following alternative solutions for Infrastructure-as-a-Service were also examined and compared to OpenStack:

Apache Cloud Stack
This open source platform is implemented in Java, was originally released by Cloud.com in 2010 and was donated to the Apache Incubator in 2012 and has since then become a top-level project of the Apache Software Foundation. *Apache Cloud Stack* consists of three nodes a *Supervisor Node*, a *VM Creator Node* and a *StorageServer Node* which is for the most part identical with *OpenStack*. The main differentiator from *OpenStack* is that it is easier to configure since it is not made up of a collection of separately configurable components. This implies a lower flexibility when it comes to highly complex deployment scenarios. A further disadvantage which is crucial for this thesis is the lack of native *Docker* support [ACSA21; MS15].

OpenNebula
OpenNebula is a platform for managing distributed infrastructures mainly in large scale data centers and was released in 2008. It is primarily implemented in C++ as well as Ruby, utilizes the Linux native drivers concept and consists of three layers. The *Drivers Layer* communicates with the underlying operating system and abstracts the infrastructure of the host as a set of services. The middle layer is concerned with managing the *Virtual Machines* and their networks and is called *Core Layer*. The final *Tool Layer* contains interfaces that allow for user interaction as well as the scheduling of *VMs* [TONA21; VGM+16].

The primary reason to chose *OpenStack* is that the university department at which this thesis was conducted already operates an instance and has the experience to provide the required infrastructure in a stable and reliable manner.

2.8 Docker

The following section describes *Docker* as lightweight container virtualization technology of choice because its wide popularity and the many available images from large organizations. It is capable of creating, deploying and managing containerized applications and was created by Docker Inc. in 2013. To use *Docker* a *Image* is necessary. It is a file which contains all specifications needed to create a *Container*. Usually an *Image* contains another base *Image* with additional customization which is required to successfully operate an application. A *Container* describes instance based on an *Image* which can be started, stopped, moved or deleted. *Docker* consist of two main parts the *Runtime* and the *Daemon* sometimes called engine [Pou20a]:

Runtime
The overall runtime is responsible for setting up the environment like *cgroups* and *namespaces* as well as starting and stopping containers. It consists of high-level and low-level runtime which communicate with each other. The low-level one is called *runc*, is part of every container and forms the interface to the underlying operating system to start and stop the *Container*. The high-level *Runtime containerd* interferes with *runc* and manages the whole lifecycle of a *Container* as well as pulling *Images* and assembling network interfaces.

Daemon
The *Docker engine* also known as *dockerd* was created to abstract both levels of the *Runtime* and provide a swift interface. It communicates with *containerd* and enables *Image*, networking and volume management. *dockerd* also exposes the *Docker* API to establish a channel of interaction.

2.8.1 Alternative Container Virtualisation Technologies

The following alternative solutions for *Container* virtualisation technologies were also examined and compared to *Docker*:

Podman
Podman was designed to leverage Linux native components to a deamonless *Container* technology. That means that *Podman* is independent on a single *Deamon* process which may pose as a single point of failure. Instead it is directly interacting with the *runc* container runtime. It is able to run, build and deploy containerized applications or images that are compliant with the *Open Containers Initiative (OCI)* standards. Hence *Podman* is able to manage containers build in *Docker*. Additionally the processes controlled by *Podman* can be run as *root* or as *unprivileged user*. Whereas the *Docker deamon* requires to be executed as *root* which can lead to security implications [TPT19]. Since there is not a large developer community nor much literature available for *Podman* it was not further considered for this thesis.

LXC
LinuX Containers enhances the *cgroups* as well as the *namespaces* functionality of the Linux kernel to provide a contained environment to execute applications. It is maintained by Canonical Inc. which is also responsible for the *Ubuntu* Linux distribution and was released in 2008. In contrast to *Docker* which handles application level containers *LXC* manages

operation level containers [Kum17; LXC20]. The difference between both technologies is illustrated in Figure 2.1. Since *LXC* have no native support in Kubernetes they are not examined further.

FreeBSD Jails
Jails are one of the earliest attempts to realize things like process isolation dating back to the year 2000. To achieve this the BSD kernel feature *croot* was further customized to virtualize file access, system users and the networking subsystem. This allows multiple processes to utilize those virtualized resources while having only restricted access to a subset of the whole file system. Each *Jail* contains a set of users and a root which are limited to their own environment [BSD20]. Like *LXC* the concept of *FreeBDS Jails* can be classified as a operating system level container. Since it is only available on the *FreeBSD* distribution this technology is out of scope for this thesis.

Docker was chosen for this thesis since it has an extensive documentation, a large developers community and has been the de facto standard container runtime used by Kubernetes. Additionally the related work which is fundamental for this thesis was conducted using *Docker*.

2.9 Kubernetes

The following section describes *Kubernetes* which is a container orchestration system that allows the operation of scalable applications with changing topologies based on workload or traffic. It was primarily developed as an internal project at Google to manage large containerized applications like Gmail and was open sourced in 2015. *Kubernetes* is organized in a Master-Slave architecture with the master node controlling all Minion computing nodes and consists of the following componenents [Pou20b; TKA20]:

Master Node
The *Master Node* also known as *Control Plane* controls the operations inside the cluster. It is concerned to manage the distribution of *Pods* between the available *Minion Nodes*. To extend the robustness of the system it is possible to create multiple redundant *Master Nodes*. The *Control Plane* utilizes the following processes and components:

API Server: Serves as the main interaction point of the cluster. It receives *JSON* formatted configurations over a *REST* API and stores them in *etcd*.

etcd: Is a lightweight distributed *Key-Value-Database* which preserves the configured target state of the cluster.

Controller Manager: Is an independent component which runs all controller processes and communicates with the *API Server* regarding the status of the cluster. It observes whether all nodes are available and all pods were launched correctly. Further it populates the *Endpoints* object and thus connects *Services* and *Pods*.

Scheduler: Decides on which *Minion Nodes* a *Pod* is deployed and is constrained by specifications of quality of service, node locations and available resources. Additionally it is concerned with the management and overseeing of the workload directed to the *Minion Nodes*.

Minion Node
A *Minion Node* describes a single host machine that is used by *Kubernetes* to form a large scale cluster. Each node contains a container runtime as well as the following components:

Kubelet: Is responsible for the state of all *Pods* on a particular host machine and communicates its state to the *Control Manager* on the *Master Node*. *Kubelet* undertakes the restart of a failed *Pod* on the same machine. If the communication between *Kubelet* and *Control Plane* is interrupted the *Master Node* assumes a failed node and moves all its *Pods* onto an available host machine.

cAdvisor: Records the utilization of the resources of a *Minion Node* and can be accessed by external applications to provide dynamic scaling of the whole cluster.

Kube-Proxy: Manages the connections and open ports on a node

2.9.1 Alternative Container Orchestration Systems

The following alternative solutions for container orchestration systems were also examined and compared to *Kubernetes*:

Docker Swarm
Docker Swarm is an open source project by Docker Inc. to provide native cluster management support in *Docker*. It bundles several *Docker* nodes into one cluster to enable dynamic scaling as well as a failover mechanism through redundancy. Like *Kubernetes* it is organized in a master-slave architecture where the master is called *Manager* and accounts for the orchestration of *Containers*. The slave is known as *Agent* which operates the scheduled *Containers*. Compared to *Kubernetes* it is far less feature rich in both *Scheduling* and *Service Management Layers*. *Docker Swarm* lacks the support of readiness checking and rolling deployments which can lead to data loss and impacts on the availability of a service. Further it does not support *namespaces* and *load balancing* that means that no multi tenancy and no dynamic traffic distribution is available in a *Docker Swarm* cluster. Since *Kubernetes* supports all those features that are particularly important for an Enterprise Content Management System *Docker Swarm* is not further considered [JBB+19; Pou20a].

Apache Mesos
Hello Test *Apache Mesos* is an open source project developed by the University of California, Berkeley in 2011 which follows the master-slave architecture pattern. *Master Nodes* control *Slave Nodes* that contain *Frameworks* which execute *Tasks*. The *Master Node* decides based on specified *Policies* how many free resources are assigned to each *Framework*. A *Framework* contains a *Scheduler* which communicates with the *Master Node* to allocate computing resources and an *Executor* that completes *Tasks* on a *Slave Node*. Compared to *Kubernetes* it contains slightly more features in the *Resource Management* and *Service Management Layers* however the one major disadvantage is the rather complex setup and integration of applications. *Apache Mesos* requires *Marathon* to be installed on top of it to support containerized workloads. [HKZ+11; JBB+19]

OpenShift
OpenShift is Red Hat's own *Kubernetes* distribution which is fully compliant and extends it with features focused on improving the productivity of developers and operators. It

was originally released with an own runtime environment in 2011 and later rewritten to implement *Kubernetes* in 2015. The proprietary features contributed by Red Hat improve the native networking and provide support in the life cycle of deployed images [EKT21]. Since *OpenShift* is not entirely open source and the additonal functionality is not relevant for this thesis it was not further considered.

Kubernetes posed as an excellent choice since it is entirely open source, implements the right balance of required features and operational complexity. Further it has the largest market adaption among orchestration systems and therefore a large developer community and many learning resources.

3 Related Work

This chapter describes the previous work conducted at the University Stuttgart which delivers the essential components on which this thesis is build on.

3.1 About the Design Changes Required for Enabling ECM Systems to Exploit Cloud Technology

The main foundation for this thesis is the work from Shao [Sha20] which focused on the separation of monolithic Enterprise Content Management applications into isolated components. This split allows the separate ECM parts to be packaged and run within containers. Until now ECM systems were usually deployed as large monolithic applications on private bare metal servers or virtual machines. Since both of those approaches involve assumptions about the infrastructure which are not always true in cloud environments a new way of deployment is necessary.

First the applications are analyzed and decomposed based on their degree of coupling. Tightly coupled components stay in the same container and loosely coupled components are split into separate containers. Every container is composed with all essential dependencies and libraries to allow the operation as stand-alone service. Further the shared data storage of monolithic ECM components is separated from the application logic so each component can use its own databases and file systems. The cooperation between separated components happens over unified communication channels. This split allows to exploit the potential of a cost effective scalability, continuous integration as well as continuous delivery.

The proof of concept which was developed during the thesis of Shao consists of an Enterprise Content Management system and a container platform. As the ECM system the IBM Content Manager Enterprise Edition and as container virtualization technology Docker are selected. The IBM system is chosen because of the historical relationship between the Institute of Parallel and Distributed Systems at the University Stuttgart and the IBM laboratories in Böblingen. Docker is selected since it is the most popular container virtualization technology with a large community and therefore many predefined images. The ECM platform consists of four separate applications within Docker containers. *lsdbsrv* that contains the *Data Catalog* and *rmdbsrv* which incorporates the *Object Catalog*. Both containers are based on the public *ibmcom/db2:latest* Docker image provided by IBM. The other two components needed to be constructed manually based on the *centos:7* image since there were no public Docker images available. *wasrm* contains the *Resource Manager Application* as well as a HTTP server. *wasicn* includes the web client, the configuration database of the web client and a HTTP server. The user interacts with the system through the web client which then sends or retrieves data from the *Data Catalog*, *Resource Manager* and *Object*

Catalog. For the applications to be able to communicate with each other a virtual Docker network was created in the development environment. Figure 3.1 illustrates the resulting proof of concept with the separate components each inside its own container.

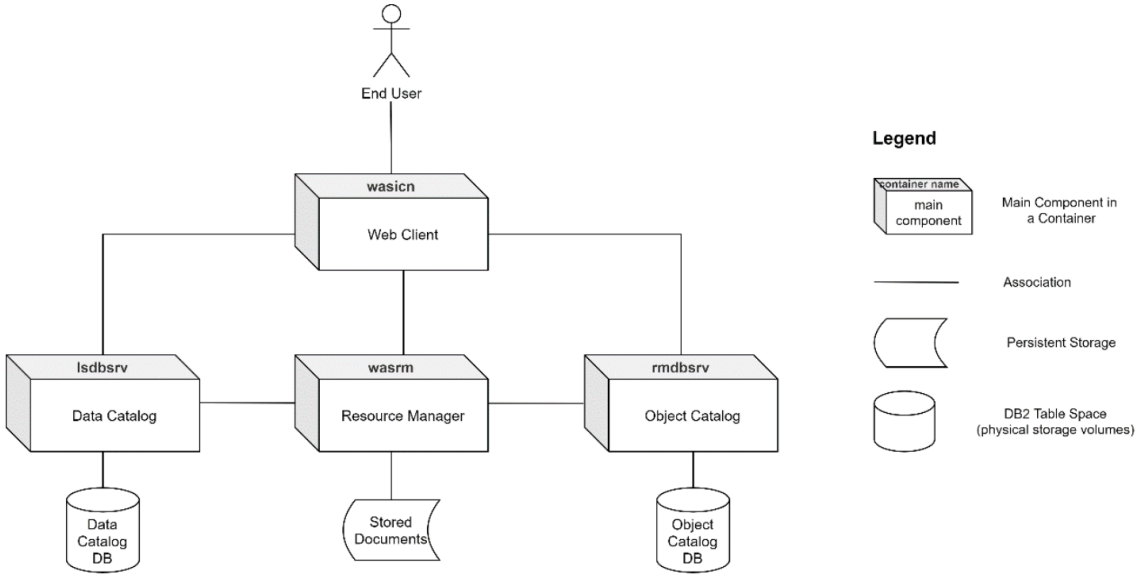


Figure 3.1: Proof of Concept by Shao [Sha20]

4 Concept

The following chapter describes the applied changes to a containerized Enterprise Content Management application introduced by [Sha20] to be operated within a Kubernetes cluster.

4.1 Implications of a Kubernetes Cluster

After its donation to the open source community by Google Kubernetes has become the most popular orchestration platform to operate and deploy containerized applications with minimum manual effort. The core concept of Kubernetes is the autonomous management of stateless applications which consist of identical, swappable and replaceable containers. Enterprise Content Management systems and other real world applications typically need some kind of stateful service which is concerned with persistent data storage [IH19].

After analyzing the dependencies of the containerized ECM applications created by Shao [Sha20] we found out that the components can be divided in two web applications and two database applications. Hence the web applications could be operated as stateless applications without further customization. The considered containers are *wasicn* with its web client component as well as *wasrm* with the encompassed resource manager component. The other components that are needed to construct a working ECM system are the data and the object catalog which both rely on databases to work properly.

4.2 Aspired ECM System Topology

The management of stateful applications like databases in a Kubernetes cluster is not trivial since it was designed to handle stateless workloads while keeping stateful components outside the managed cluster [IH19]. To pave the way to an operational ECM system we need to carefully examine how the state of the data as well as the object catalog can be preserved while being available to the cluster. Additionally the initial Docker images provided by [Sha20] relied on the application data and executables to be locally present on the host machine. There are two possibilities to achieve that each with its advantages and drawbacks:

Maintaining the State on the Host Node
To keep the state in filesystem of a host node it is crucial to upload the necessary data to that node on which the corresponding application will be operated. This approach implies that each potential node on which the corresponding application could be relocated by the *Scheduling Layer* of the cluster has to contain the needed data. The data allocation on the computing nodes is illustrated in Figure 4.1. It shows that only nodes that have matching

data can operate a certain application. In contrast to *Node 2*, *Node 1* and *Node 3* contain the necessary *Data* in their local filesystem and are able to host the application. Replicating the required data across all nodes of a large cluster leads to an overhead in storage allocation. By maintaining the data exclusively on dedicated nodes that operate only one specific application the flexibility and the availability of the whole system can be affected. In case of a failing node the *Scheduling Layer* can not relocate the application autonomously on another node without applying procedures that ensure that the required data is present on the particular node. The biggest advantage of this procedure is the minimization of the latency of read and write operations all required data is already present on the local node.

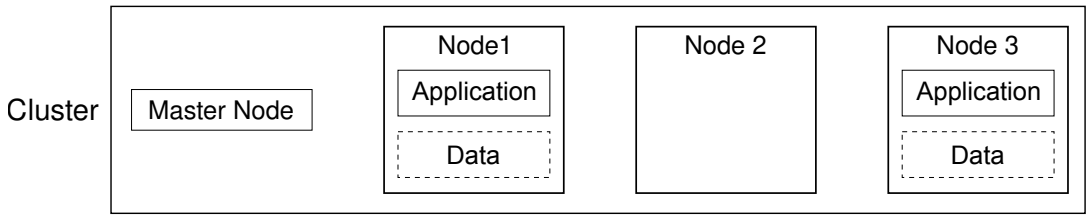


Figure 4.1: Data Allocation in Local Filesystems of Host Servers in a Cluster

Removing the State completely from the cluster
Relocating the state into an external storage service allows the *Scheduling Layer* to scale or move the web applications without assumptions about the configuration of the computing nodes that form the cluster. The external service can be located on a separate Virtual Machine or on a physically separated host. Since the storage service is now moved to an external infrastructure it needs its own maintenance effort which can not be supervised through the *Service Management Layer* of the cluster. Furthermore it can lead higher latencies of read and write operations due to the potential distance of the external infrastructure to the cluster.

Given the complexity of setting up and managing stateful services in Kubernetes this thesis is restricted to stateless services. Therefore the state is removed entirely from the cluster and an external Network File System server is utilized. Further a stateful data service requires a much more complex cluster design and is easier to maintain and scale on its own. Therefore the containerized applications can be scaled across the cluster independent of whether the volumes are mounted on a particular node or not. The following Figure 4.2 illustrates the initially aspired topology with all storage capacities transferred to the NFS server.

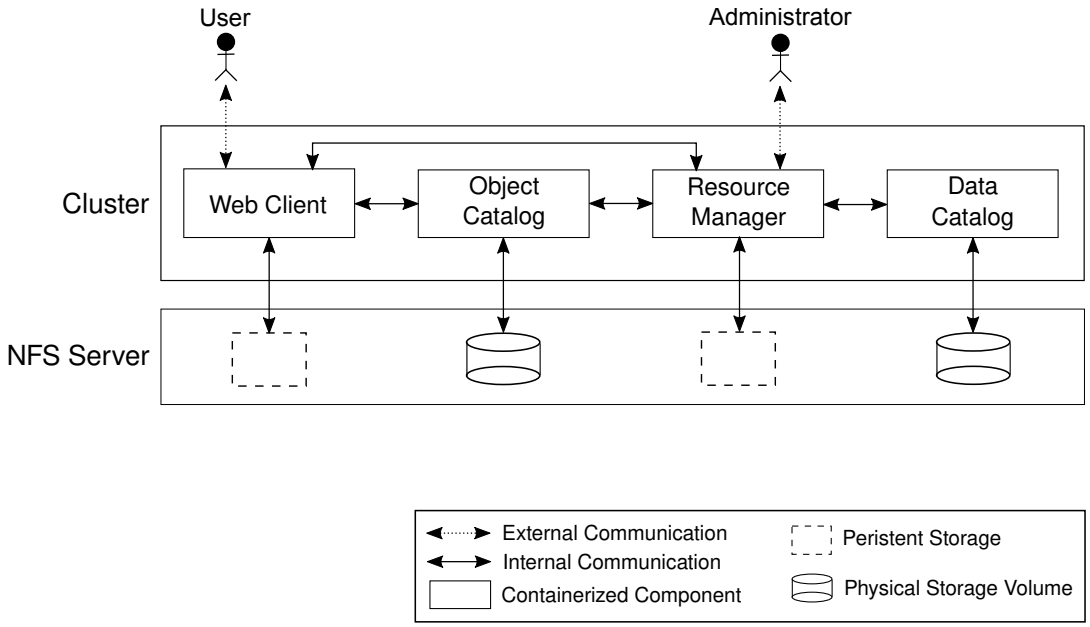


Figure 4.2: Initial Cluster Topology

5 Prototype

In this chapter describes the prototypical implementation of the changes to the approach by Shao [Sha20] from the previous chapter to allow the operation within a Kubernetes cluster.

5.1 Infrastructure

The following prototype was developed, deployed and tested on the infrastructure of the Institute of Parallel and Distributed Systems at the University Stuttgart. It consists of a Open Stack instance which manages the virtual machines running *CentOS 8*[1] that are used throughout this thesis. To further simplify the development process and minimize the effort of setting up and managing a set of virtual machines as Kubernetes nodes this thesis relies on the KiND project.

5.1.1 Kubernetes in Docker (KiND)

The KiND Project uses multiple Docker containers to create a virtual Kubernetes cluster with multiple nodes. It was created by the Kubernetes authors to enable a fast and straightforward way to test and verify cluster configurations on the local machine of a developer. A KiND cluster consists of the following Docker images:

Base Image
This Image is based on Ubuntu and contains only the necessary dependencies for running nested containers, systemd and Kubernetes. Additionally it contains a custom entrypoint that allows to execute configurations before the container becomes available.

Node Image
This image is an extension of the Base Image and contains the tools required to operate and manage Kubernetes resources within a cluster. KiND aims to leverage existing tooling for Kubernetes to create a familiar environment for developers.

Each node of a cluster runs its own Docker container which is identified through a Docker object label containing the cluster name and node id. A KiND cluster consists of at least one control plane and one or many worker nodes. The control plane handles incoming network traffic, storage mounts on the host machine and additional initial configurations of the cluster. A worker node is equivalent to a compute node in a regular Kubernetes cluster [BBH19; TKA21].

[1] https://www.centos.org

Listing 5.1 KiND Cluster Configuration File

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: ecm
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 30043
        hostPort: 9043
      - containerPort: 30044
        hostPort: 9044
      - containerPort: 30080
        hostPort: 9080
      - containerPort: 30081
        hostPort: 9081
      - containerPort: 30443
        hostPort: 9443
      - containerPort: 30444
        hostPort: 9444
  - role: worker
  - role: worker
```

5.1.2 Configuration of the Used Cluster

To allow the external world to communicate with the applications inside a KiND cluster it is necessary to include a set *extraPortMappings* in the cluster configuration file shown in Listing 5.1. Since Kubernetes only allows external ports to be in between 30000 and 32767 as well as the fact that KiND runs in Docker a workaround was needed to expose the expected ports on the host machine. The workaround consists of *nodePort* components inside the Kubernetes cluster that expose the ports of the application inside a pod which are then connected to the *hostPort* of the *controlPlane* container on the host server. The applied workaround of the port mappings is shown in Figure 5.1.

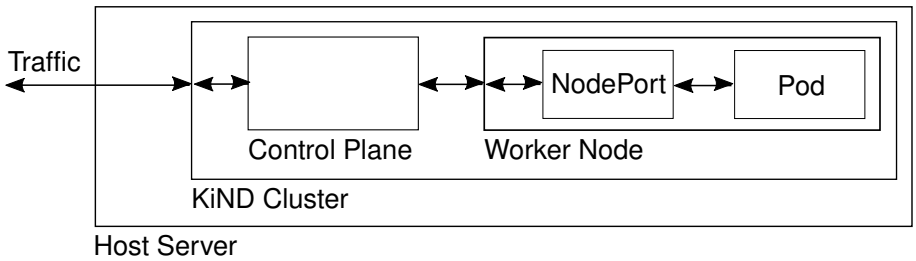


Figure 5.1: Workaround to Enable Incoming Traffic to the KiND Cluster

5.2 Kubernetes Components

To successfully run the aspired ECM system topology presented in Chapter 4 within a Kubernetes cluster the following components need to be utilized [BBH19; IH19; Pou20b; TKA20]:

Namespaces
To enable the maintenance of multiple virtual clusters on a single physical cluster the concept of *Namepaces* was introduced. It is especially helpful for managing the distribution of computing resource in large environments with multiple teams and projects. Thereby they allow the implementation of access control and resource quotas. Resource names must be unique within a *Namespace* but not across different *Namespaces*. Every cluster generates its own DNS space which is called *cluster.local* and by placing an *Service*-Object in a *Namespace* the resulting fully qualified domain name (FQDN) will be `<object-name>.<namespace>.svc.cluster.local`. A *Pod* inside the same *Namespace* can connect to the *Service* via its `<object-name>` but a *Pod* from a different *Namespace* needs the FQDN to establish a connection.

Pod
In Kubernetes a *Pod* is the smallest unit that can be deployed and is a runtime isolation for a set of containers. The grouped containers are always deployed as a collective on the same host machine and the *Scheduling Layer* strives to find a placement in the cluster which satisfies all constraints imposed by the incorporated containers. This pooling allows a fast exchange of information between each other by leveraging the file system, networking or inter process communication of the host. Additionally every *Pod* has its own IP-address and thus a port range which is shared among all enclosed containers. Therefore it should be payed attention to port allocation at configuration time to prevent port conflicts. *Pods* are designed to contain only a single instance of an application hence the *Pod* should be replicated for scaling. Another key concept of Kubernetes is the the ephemerality of *Pods* in their life cycle. That means that whenever a *Pod* encounters an error the *Pod* is not diagnosed and repaired but rather terminated and a new *Pod* with identical properties is started. The same happens when a computation node fails: the *Pods* on the node in question are not saved and moved to an available node but forgotten and a new set of *Pods* is created. Even in the case of lightweight configuration changes the modification will not be passed to a running container instead the active *Pod* will be terminated and a new one containing the changes will be spun up. To know if a container inside a *Pod* is working properly Kubernetes runs a set of diagnostic functions on a container:

ExecAction: Performs a specified command inside the container and succeeds if the command terminates without an error.

TCPSocketAction: Tries to open a TCP socket on a specified port of the *Pods* IP-address and succeeds if a socket is opened.

HTTPGetAction: Sends a HTTP `GET` request to a specified port and route on the *Pods* IP-address and succeeds if the status code of the response is between `200` and `400`.

Kubernetes provides the previously introduced functions to monitor the following conditions a *Pod* could encounter during its life cycle:

Startup: Monitors if the application inside a container successfully launched during the start up phase of the *Pod*. The following conditions are only evaluated if the Startup was successful. It is especially helpful for large and complex applications that need a certain time span to become available for service.

Readiness: Evaluates whether the container is able to respond to requests. If the probe fails Kubernetes prevents incoming traffic to be passed to the *Pod* by removing its IP-address from the *Endpoint* objects of all *Services* that reference the *Pod*.

Liveness: Continuously checks whether a container is in a functional state to process incoming requests. This probe should be used if an application has a strong dependency on a external back-end service to prevent information loss.

If the *Startup* and *Liveness* probes fail Kubernetes autonomously terminates the *Pod* and launches a new one. The termination happens gracefully since *Pods* are distributed processes running distributed within a cluster. Abruptly killing processes without a cleanup phase can leave corrupted data and open database connections behind. To prevent those undesirable residuals, clean up commands can be passed through a specified *postStop* hook. It is executed after the termination signal reached the *Pod* and before the `TERM` signal is passed to the process with the id 1 inside each container.

ReplicaSets
A *ReplicaSet* takes care of maintaining a specified number of identical active *Pods* at any given point in time. It contains a selector that identifies which *Pods* should be monitored, a count of how many *Pods* should be running simultaneously and a template which describes the properties newly generated *Pods* should comply with. It is considered a best practice to only interact directly with *ReplicaSets* when dealing with complex deployment scenarios. In regular circumstances it should be sufficient to utilize *Deployments* to manage the deployed *Pods*.

Deployments
The update and start of applications inside a cluster in a declarative way is performed through *Deployment* configuration files. They serve as a blueprint of the service an application should provide. A *Deployment* contains the desired state of an application in regard to the number of pods, which image should be used for the containers inside the *Pods*, the assignment of network ports and information about how to perform rolling updates. To accomplish a successful rolling update by deploying a new version of the image of an application the updated *Deployments* file is resubmitted to the Kubernetes API server. It detects a new desired state in the cluster and creates a new *ReplicaSet* for the *Pods* containing the updated image. So now the cluster contains a new and an old *ReplicaSet* and each time a new *Pod* with the updated image is launched a running *Pod* containing the deprecated image is terminated. This procedure allows a smooth update process with minimized downtime. After performing a rolling update by default the previous 10 outdated *ReplicaSets* containing their whole configuration are retained in the cluster without managing active *Pods*. Hence a rollback is essentially a rolling update the other way around to a desired previous version of the application. Additionally the prior described *Startup-*, *Readiness* and *Liveness*-Probes need to be specified within the *Deployments* files. Figure 5.2 illustrates that *Deployments* are controlling *ReplicaSets*, and *ReplicaSets* are managing *Pods*.

PersistentVolumes
Pods in a cluster are intended to be stateless but sometimes external storage needs to be mapped onto the cluster to provide data and storage capacity for the running applications. This component was introduced to abstract the usage of storage from its provision since managing storage differs from managing computing resources. *PersistentVolumes* differ in life cycle from *Pods* and contain the information to use NFS or storage options provided by cloud vendors. They come in two different forms:

Static: A fixed set of *PersistentVolumes* is manually created at configuration time and can be consumed by applications within the cluster.

Dynamic: Kubernetes autonomously allocates *PeristentVolumes* at run time based on a *StorageClass* defined during configuration time. This happens if and only if a matching *PersistentVolumeClaim* exists. By defining the *StorageClass* as an empty string in a *PersistentVolumeClaim* the dynamic provision is disabled.

Persistent Volume Claim
The *Persistent Volume Claim* authorizes the applications inside a *Pod* to access the specified *PersistentVolume*. The configuration file contains information about the needed storage capacity, access mode, *StorageClass* etc. It gets bound to a *PeristentVolume* by Kubernetes on a one-on-one basis as soon as the specified conditions are satisfied. Otherwise the claim will remain unbound until new resources are added to the cluster or already utilized are freed. Depending on the storage provider a *PersistentVolumeClaim* is able to consume storage with different access modes:

ReadWriteOnce: One node can exclusively write on this volume.

ReadOnlyMany: Many nodes can read jointly from this volume.

ReadWriteMany: Many nodes can read from as well as write to this volume.

Network File Storage supports all previously discussed access modes.

Storage Classes
This component allows to specify classes of provided storage which may differ in the enforced policies by the storage service administration like quality-of-service, backup etc.

Service
Because of the dynamic nature of scaling and locating *Pods* throughout a Kubernetes cluster it is not feasible to use the IP-address of a *Pod* to communicate with the applications it contains. *Services* were introduced to provide a stable and reliable networking interface for a set of ephemeral *Pods*. It exposes a defined host name and ports that are immutable at run time. *Pods* and *Services* have a one-to-one correspondence and are mapped through the *app* selector label. By omitting the selector label the corresponding *Service* won't create an *Endpoint* object. This is useful to incorporate applications that operate on an external host like database applications.

Nodeport
In Kubernetes the *Pods* within the cluster are not reachable from the outside. This component is a special type of *Service* which allows to reach the internal IP-space of the cluster from external applications. Kubernetes either autonomously assigns an access port or applies a specified port in the range of 30000 and 32767. *Nodeports* are not able to manage incoming

traffic like a *Load Balancer* component and should therefore not be used in production systems. Since this thesis aims to investigate whether ECM systems can be managed by Kubernetes at all the aspect of load balancing is out of scope but is considered in the companion master thesis [Hag21].

Endpoints
Endpoints are dynamic objects that contain all the available *Pods* and their IP-addresses and ports. Before directing traffic to *Pods* the corresponding *Service* queries the *Endpoints* object to get the addresses of currently available *Pods* and then chooses one to direct the request to.

Figure 5.2 illustrates the previously presented components and its interactions inside a cluster to maintain its specified desired state.

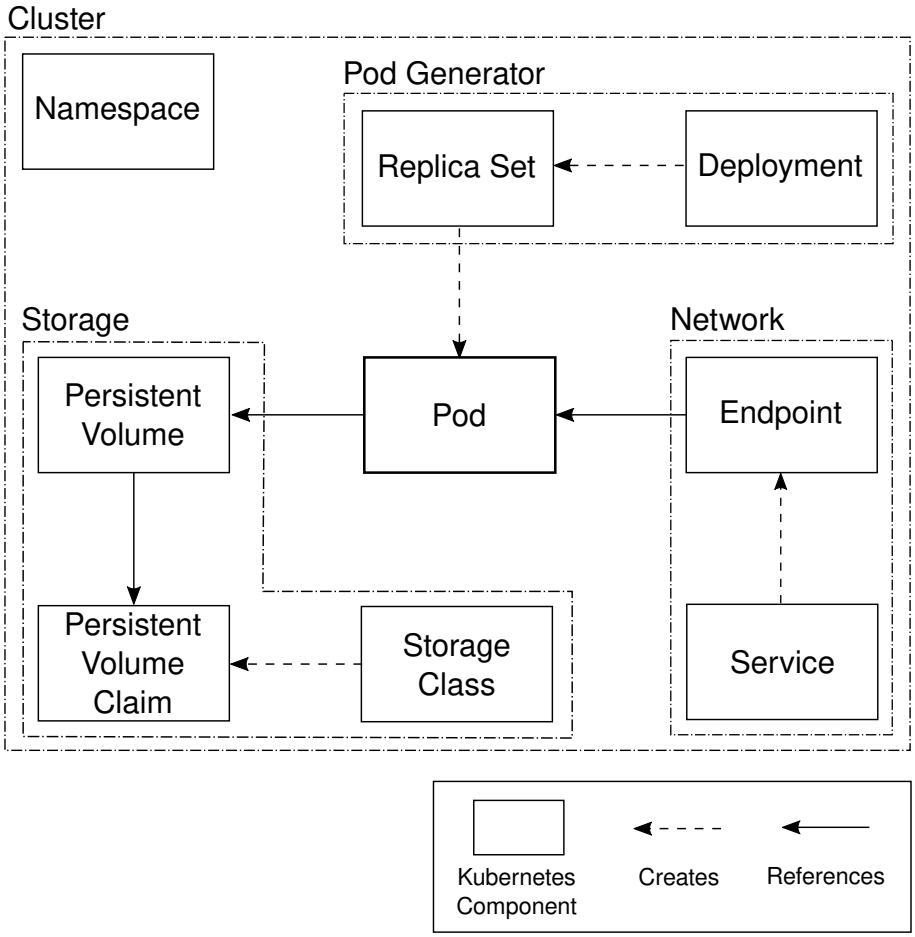


Figure 5.2: The Interactions of the Components Inside a Kubernetes Cluster [TKA20]

5.3 Initially Aspired System Topology

The following Figure 5.3 shows the previously presented components and its role to implement a scalable Enterprise Content Management system based on the concept introduced in Chapter 4.

The overall System is split in two parts the Kubernetes cluster and a NFS server which holds all the data needed to operate the applications inside the cluster. The vertical dashed lines indicate the separation of the individual ECM applications. Each application utilizes a *PersistentVolume* with a corresponding *PersistentVolumeClaim* to either load needed application data or read from or write to database files. The *Resource Manager* application requires two JDBC connections to the *Object Catalog* and *Data Catalog* as well as an external connection for administration. The *Web Client* portal application only requires an internal connection to the *Object Catalog*, an internal connection to the *Resource Manager* and an external to provide user access. External connections are implemented through *Nodeports* to enable user interaction.

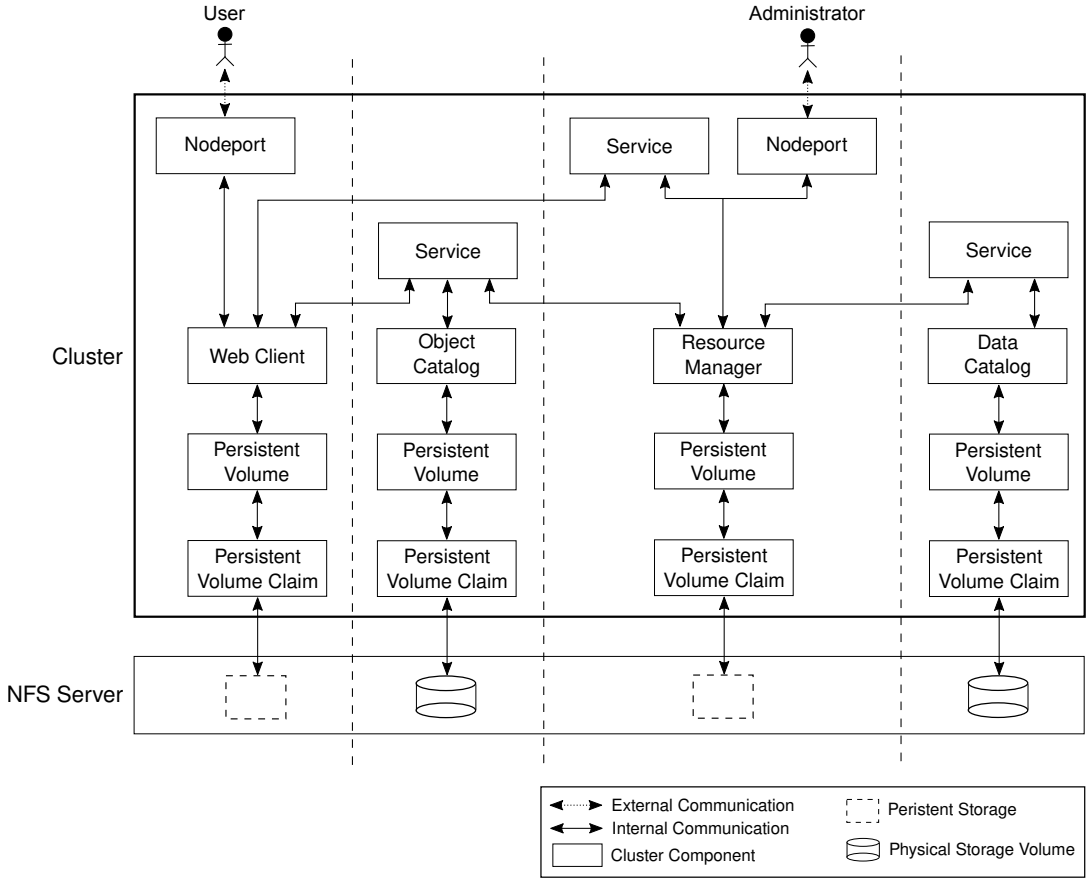


Figure 5.3: Initial Topology of the ECM System Inside a Kubernetes Cluster

5.4 Refactored ECM System Topology

The initially aspired system topology described in Section 5.3 turned out to be brittle when put under load. During functional tests it was discovered that the topology design required a refactoring as the stateful database service was very unstable and unreliable. It could sometimes handle manual queries as well as file uploads and sometimes the databases would crash for no obvious reason. Following investigations showed that running DB2 instances on NFS storage was the source of the occurring issues.

To enhance the prototype stability and therefore availability we decided to remove the stateful database services from the cluster and operate it as *Docker* containers on a different host server. This decision resulted in removing the *Deployment*, *PersistentVolume* and *PersistentVolumeClaim* components and adding a *Endpoints* component for both the *Object Catalog* and the *Data Catalog*. Besides that a new Docker image was created which contains all essential files of the *Resource Manager* and *Web Client* applications such that external file mounts are avoided. Hence the *Persistent Volume* and *Persistent Volume Claim* component were removed from the topology. The new image incorporates both applications and the startup of the separate systems is enforced through various configurations when starting the corresponding container. The following Figure 5.4 illustrates the refactored topology to prevent the previously discussed challenges of the initial implementation.

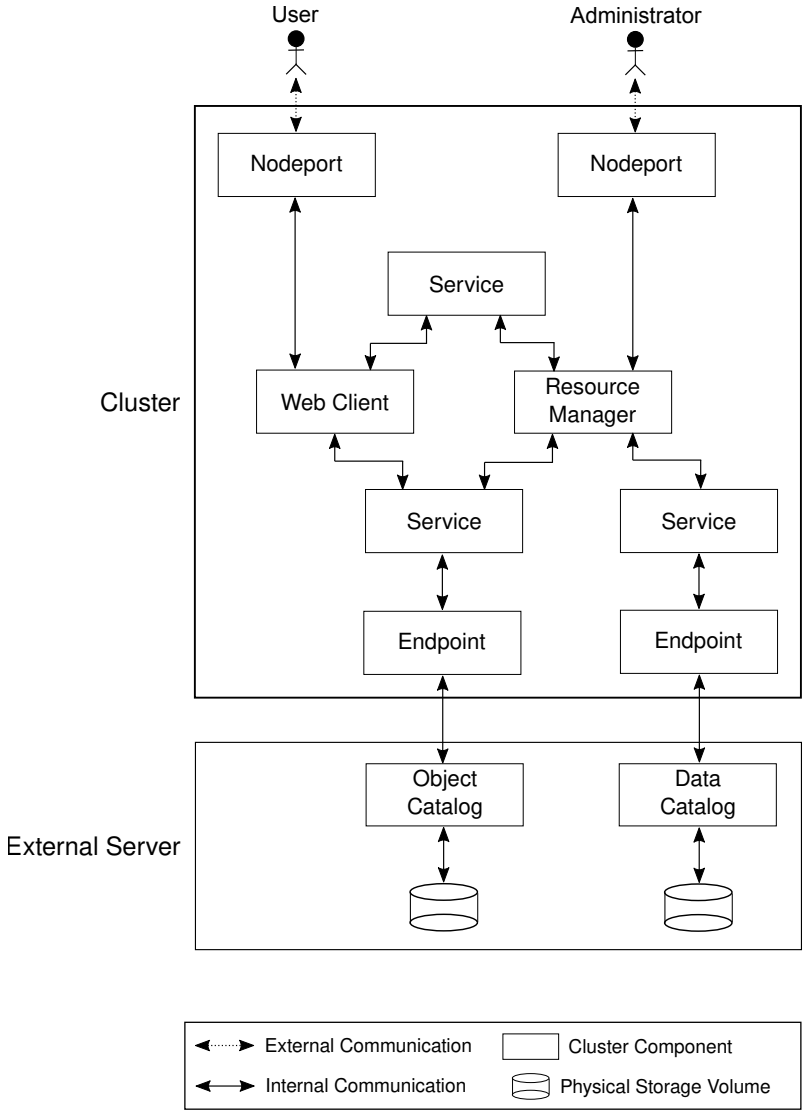


Figure 5.4: Improved Topology of the ECM System Inside a Kubernetes Cluster

5.5 Implementation Details of the Kubernetes Components

The following section describes the configuration of the components of a Enterprise Content Management system within a Kubernetes cluster. It describes the implementation by means of source code with a detailed explanation of the design decisions made.

5.5.1 Data Catalog and Object Catalog

Because the *Data Catalog* and *Object Catalog* are relocated to an external server outside the cluster they require means to communicate with the applications left inside the cluster. To achieve this a *Service* without the app selector and a manually configured *Endpoint* object is necessary. Since the configuration files of both components have only minor disparities only the *Data Catalog* configuration is displayed.

Service
The *Service* configuration file is described in Listing 5.2. The specified *port* in the *spec* section has to correspond to the port exposed within the cluster whereas the *targetPort* needs to correspond with the port configured in the *Endpoint* object. The *Service* configuration of the *Object Catalog* component is identical except for the *port* and *targetPort* number which is 50001.

Listing 5.2 Data Catalog Database *Service* Configuration File

```
kind: Service
apiVersion: v1
metadata:
  name: icm86-ls
  namespace: ecm
spec:
  ports:
    - port: 50000
      targetPort: 50000
```

Endpoint
The *Endpoint* configuration file is described in Listing 5.3. The described *port* corresponds directly to the exposed port of the external data source which is reachable through the specified IP-address. The configuration file of the *Object Catalog Endpoint* is identical except for the *port* number which is 50001

5.5.2 Resource Manager Application and Content Navigator

Since the configuration files for the components of the *Resource Manager* and *Web Client* applications are almost identical just the *Resource Manager* is considered in the following section. As displayed in Figure 5.4 both applications depend of the following Kubernetes components:

Listing 5.3 Data Catalog Database *Endpoint* Configuration File

```
kind: Endpoints
apiVersion: v1
metadata:
  name: icm86-ls
  namespace: ecm
subsets:
  - addresses:
      - ip: 192.168.221.148
    ports:
      - port: 50000
```

Nodeport

The *Nodeport* configuration file is described in Listing 5.4. Because we want Kubernetes to take as much work from our hands as possible the *Nodeport* configurations of components inside the cluster all contain the app selector in the *spec* field. Therefore we don't need to worry about *Endpoints* objects. The defined *nodePorts* inside the *ports* specification exposes the corresponding *ports* and *targetPorts* of the application inside the *Pods*. Since our cluster resides in an emulated environment using Docker containers the exposed ports need to be looped through the Docker network layer to be reachable from outside the *Control Plane* container. Figure 5.1 illustrates the applied port mappings.

Listing 5.4 Resource Manager Application *Nodeport* Configuration File

```
kind: Service
apiVersion: v1
metadata:
  name: icm86-rmapp-nodeport
  namespace: ecm
spec:
  type: NodePort
  selector:
    app: icm86-rmapp
  ports:
    - name: icm86-rmapp-websphere-admin-port
      port: 9043
      targetPort: 9043
      nodePort: 30044
      protocol: TCP
    - name: icm86-rmapp-insecure-application-port
      port: 9080
      targetPort: 9080
      nodePort: 30080
      protocol: TCP
    - name: icm86-rmapp-secure-application-port
      port: 9443
      targetPort: 9443
      nodePort: 30443
      protocol: TCP
```

Service
The *Service* configuration contains only minor differences compared to the *Nodeport* configuration described in Listing 5.4. It does not contain a `spec.type` attribute nor a `spec.ports.nodePort` attribute in each port definition. This *Service* handles the communication between *Web Client* and *Resource Manager*.

Deployment
The *Deployment* configuration file is described in Listing 5.5. It contains all information needed to operate the applications within a cluster. For the Minimum Viable Prototype implemented during this thesis the number of `spec.replicas` was set to 1. The entry `spec.selector.matchLabels.app` sets a label to describe with which *Pods* the *Deployment* is associated. The following section `spec.template` contains all information for the *ReplicaSet* to generate new *Pods* if necessary. `spec.template.metadata` defines the label inside the cluster which is used by *Services* to direct traffic to the deployed *Pods* The next section defined in `spec.template.spec.containers` specifies the structure of the launched *Containers* inside the *Pod* and details concerned with its life cycle. Following the generation of the *Container* based on the chosen image and the exposed ports the specified command is executed. The `spec.template.spec.containers.command` runs the *entrypoint* script inside the root directory of the image which is described in Listing 5.6. Simultaneously when starting the `command` the `startupProbe` is launched. It continuously evaluates whether a TCP connection can be established with port 9443. The configuration of this probe allows a 15 minute time frame for the application to start before terminating the *Pod*. The port 9443 was selected because the main application can be reached that way so as soon as a connection would succeed the *Ressource Manager Application* could be used. The large time frame is chosen due to long start up phase of the *WebSphere Application Server*. Due to design decisions of the Docker image the *Container* has to be kept running. Therefore Kubernetes is not able to know whether the application inside the *container* was still in a healthy state. As a countermeasure the `livenessProbe` and `readinessProbe` are utilized based on the same assumptions regarding the port as the `startupProbe`. The first reports to Kubernetes after 3 failed connection attempts that the *Pod* has to be restarted. The second probe instructs the *Service* to stop directing traffic to the *Pod* after 1 failed connection attempt. After a *Pod* is flagged as failed or the *Pod* gets terminated due to scaling requirements `spec.template.spec.containers.lifecycle.preStop` is executed. This command allows a graceful shutdown of the *Websphere Application Server* inside the *Pod* within the default time frame of 30 seconds to prevent eventual resource leaks.

Entrypoint Script
To successfully start the Resource Manager and the Content Navigator applications the following *entrypoint.sh* script is run during setup time. The developed script is displayed in Listing 5.6. At first it is enforced through `pipefail` that the whole script should exit with a non-zero status code if any command or pipe exits with a non-zero status code. Subsequently the start script for the *WebSphere Application Server* is called. After a successful start the script maps the logs of the application to *STDOUT* using the `tail` command. The final step serves two purposes, at first it allows to inspect the logs from the outside of the *Pod* by using `kubectl logs`. Secondly it prevents the container from exiting and thus terminating the whole *Pod*. This procedure is necessary because the initially generated Docker image was operated in *interactive* mode. This

means that the *Docker Demon* keeps the *STDIN* open even if the container is running in *detached* mode. Since Kubernetes *Deployments* do not support an *interactive* mode this workaround had to be implemented.

5.6 Source Code

The source code and image developed during this master thesis can be obtained from an internal repository resp. registry of the Institute of Parallel and Distributed Systems at the University Stuttgart. It includes the configuration files for the Kubernetes cluster as well as scripts to install all required dependencies, setup the environment and start all cluster components except the required *Docker* image. The following steps work only within the environment of the university.

To install all dependencies and load the Docker images from the internal registry simply run `./runSetupJobs.sh` in the root directory of the repository. It is important to mention that the required image to start the *Pods* inside the *KiND* cluster needs to be loaded in the Docker instance of the host beforehand. After all dependencies are available the `./start.sh` script can be executed which goes through following steps:

1. Delete any old *KiND* cluster with the name *kind-ecm*
2. Create a new cluster based on the configuration file displayed in Listing 5.1
3. Set the *kubectl* context to the newly generated cluster
4. Load the required Docker images into the cluster
5. Instruct `kubectl` to create all required components

Since loading all Docker images into *KiND* can take a long time the additional script `repopulate.sh` was created. It allows to delete all components inside the cluster and recreate them based on new configuration files without deleting the cluster and reloading all Docker images.

Listing 5.5 Resource Manager Application *Deployment* Configuration File

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: icm86-rmapp
  namespace: ecm
spec:
  replicas: 1
  selector:
    matchLabels:
      app: icm86-rmapp
  template:
    metadata:
      labels:
        app: icm86-rmapp
    spec:
      containers:
      - name: icm86-rmapp
        image: ecmdocker.novalocal:5043/ipvs-as/icm86/cm8-rmapp:v1.2.5
        ports:
        - containerPort: 80
        - containerPort: 9043
        - containerPort: 9080
        - containerPort: 9443
        command:
        - "/root/entrypoint.sh"
        startupProbe:
          failureThreshold: 30
          periodSeconds: 20
          timeoutSeconds: 10
          tcpSocket:
            port: 9443
        livenessProbe:
          periodSeconds: 5
          timeoutSeconds: 2
          successThreshold: 1
          failureThreshold: 3
          tcpSocket:
            port: 9443
        readinessProbe:
          periodSeconds: 5
          timeoutSeconds: 2
          successThreshold: 1
          failureThreshold: 1
          tcpSocket:
            port: 9443
        lifecycle:
          preStop:
            exec:
              command:
              - "/bin/bash -c"
              - "/opt/IBM/WebSphere/AppServer/profiles/icm86AppProfile/bin/stopServer.sh
                server1 -profileName icm86AppProfile -username wasadmin -password passw0rd"
```

Listing 5.6 Resource Manager Application *Deployment entrypoint.sh* script

```
#!/usr/bin/env bash
set -Eeuo pipefail

echo -e "\n start the ICN-WAS server in the foreground "
/root/icmStartWas.sh

echo -e "ICN-WAS instance ready to go; tailing the sysout log file ..."
/usr/bin/tail -n 100 -F /opt/IBM/WebSphere/AppServer/profiles/icm86AppProfile/logs/server1/rm/
icmrm/icmrm.logfile
```

6 Conclusion and Outlook

This thesis demonstrates that the approach by Shao [Sha20] can be operated properly on a *Docker* based container environment but it was not possible to apply a one to one porting into a Kubernetes cluster. The cluster needs to be further enhanced to handle stateful database services in an automated manner by leveraging Kubernetes technology.

This work evaluates two approaches of operating stateful applications inside Kubernetes. Based on this investigation the first concept is proposed and implemented in the form of a prototype. While conducting reliability tests it turned out that the developed solutions of running the *Data Catalog* and *Object Catalog* of the ECM system could not guarantee a stable and reliable operation inside the cluster. The objective of managing stateful database applications inside a Kubernetes cluster poses as a serious challenge and requires a much more complex deployment design. This is mainly because Kubernetes was designed to autonomously manage stateless workloads. Therefore the stateful components were removed and managed on an external infrastructure. In the end the effort was split in two phases: primarily focus on implementing the stateless components and secondarily further improve the initial approach to handle stateful components. Given the time constraints for this thesis and the complexity of the task the stateful databases services are left on the external docker environment. Additionally the stateless applications which stayed inside the cluster required the creation of a new Docker image which eliminated the need of external data sources mounted into the *Pods*.

Future work might improve the developed prototype through investigating the areas of load balancing, dynamic scaling and security in Kubernetes clusters.

Bibliography

[Aa10] M. Armbrust, et al. “A View of Cloud Computing”. In: *Communications of the ACM* 53.4 (Apr. 2010), pp. 50–58. DOI: 10.1145/1721654.1721672 (cit. on p. 14).

[ABG+15] S. Adkins, J. Belamaric, V. Giersch, D. Makogon, J. E. Robinson. *OpenStack Cloud Application Development*. Wrox, 2015. ISBN: 1119194318 (cit. on p. 18).

[ACSA21] The Apache CloudStack Authors. *The Official Apache CloudStack Documentation*. 2021. URL: http://docs.cloudstack.apache.org/en/latest/ (cit. on p. 19).

[AIIM13] AIIM. *What is Enterprise Content Management (ECM)?* 2013. URL: https://www.aiim.org/Resources/Glossary/Enterprise-Content-Management (cit. on p. 13).

[BBH19] B. Burns, J. Beda, K. Hightower. *Kubernetes: Up and Running - Dive into the Future of Infrastructure*. O’Reilly Media, Inc., 2019. ISBN: 1492046531 (cit. on pp. 31, 33).

[BSD20] The FreeBSD Team. *The Official FreeBSD Documentation*. 2020. URL: https://docs.freebsd.org/en/books/handbook/jails/ (cit. on p. 21).

[EKT21] M. Elder, J. Kitchener, B. Topol. *Hybrid Cloud Apps with OpenShift and Kubernetes: Delivering Highly Available Applications and Services*. O’Reilly Media, Inc., 2021. ISBN: 9781492083818 (cit. on p. 23).

[GHH+12] K. R. Grahlmann, R. W. Helms, C. Hilhorst, S. Brinkkemper, S. Van Amerongen. “Reviewing enterprise content management: A functional framework”. In: *European Journal of Information Systems* 21.3 (2012), pp. 268–286 (cit. on p. 13).

[Hag21] P. Hagemann. “Evaluating Dynamic Load Balancing of ECM Workload Pattern Wmployed in Cloud Environments Managed by a Kubernetes/Docker Eco-System”. MA thesis. Germany: University of Stuttgart, 2021 (cit. on p. 36).

[HKZ+11] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, I. Stoica. “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center”. In: *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*. Boston, MA: USENIX Association, Mar. 2011. URL: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center (cit. on p. 22).

[IH19] B. Ibryam, R. Huss. *Kubernetes Patterns: Reusable Elements for Designing Cloud Native Applications*. O’Reilly Media, Inc., 2019. ISBN: 1492050288 (cit. on pp. 27, 33).

[JBB+19] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli. “Container Orchestration Engines: A Thorough Functional and Performance Comparison”. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053 (cit. on pp. 17, 18, 22).

[Kha17] A. Khan. “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933 (cit. on p. 17).

[Kum17] S. Kumaran. *Practical LXC and LXD Linux Containers for Virtualization and Orchestration*. Apress, 2017. ISBN: 9781484230244 (cit. on pp. 15, 16, 21).

[LXC20] The LXC Team. *The Official LXC Documentation*. 2020. URL: https://linuxcontainers.org/lxc/introduction/ (cit. on p. 21).

[MG11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. 2011. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (cit. on p. 14).

[MS15] J. P. Mullerikkal, Y. Sastri. “A comparative study of OpenStack and CloudStack”. In: *2015 Fifth International Conference on Advances in Computing and Communications (ICACC)*. IEEE. 2015, pp. 81–84 (cit. on p. 19).

[New19] S. Newman. *Monolith to Microservices - Evolutionary Patterns to Transform Your Monolith*. O’Reilly Media, Inc., 2019. ISBN: 1492047848 (cit. on p. 17).

[Por16] M. Portnoy. *Virtualization Essentials*. Sybex, 2016. ISBN: 1119267722 (cit. on p. 15).

[Pou20a] N. Poulton. *Docker Deep Dive - Zero to Docker in a Single Book*. Independently Published, 2020. ISBN: 1521822808 (cit. on pp. 15, 20, 22).

[Pou20b] N. Poulton. *The Kubernetes Book*. Independently Published, 2020. ISBN: 1521823634 (cit. on pp. 21, 33).

[RB14] T. Rosado, J. Bernardino. “An overview of openstack architecture”. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. 2014, pp. 366–367 (cit. on p. 18).

[RDG20] P. Reznik, J. Dobson, M. Gienow. *Cloud Native Transformation - Practical Patterns for Innovation*. O’Reilly Media, Inc., 2020. ISBN: 9781492048909 (cit. on p. 17).

[Sha20] G. Shao. *About the Design Changes Required for Enabling ECM Systems to Exploit Cloud Technology*. 2020. URL: https://elib.uni-stuttgart.de/bitstream/11682/11274/1/GangShaoMasterArbeit.pdf (cit. on pp. 25–27, 31, 45).

[TKA20] The Kubernetes Authors. *The Official Kubernetes Documentation*. 2020. URL: https://kubernetes.io/docs/concepts/ (cit. on pp. 21, 33, 36).

[TKA21] The Kubernetes Authors. *KiND - Kubernetes in Docker*. 2021. URL: https://kind.sigs.k8s.io (cit. on p. 31).

[TONA21] The OpenNebula Authors. *The Official OpenNebula Documentation*. 2021. URL: https://opennebula.io/docs (cit. on p. 19).

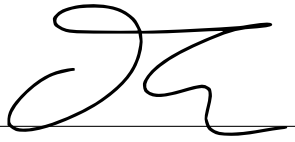
[TPT19] The Podman Team. *The Official Podman Documentation*. 2019. URL: https://docs.podman.io/en/latest/ (cit. on p. 20).

[VGM+16] A. Vogel, D. Griebler, C. A. Maron, C. Schepke, L. G. Fernandes. “Private IaaS clouds: a comparative analysis of OpenNebula, CloudStack and OpenStack”. In: *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*. IEEE. 2016, pp. 672–679 (cit. on p. 19).

All links were last followed on August 30, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Reutlingen, 01.09.2021 

place, date, signature