HSB
Hochschule Bremen
City University of Applied Sciences

ALFRED-WEGENER-INSTITUT
HELMHOLTZ-ZENTRUM FÜR POLAR- UND MEERESFORSCHUNG

Hochschule Bremen

Zentrum für Informatik und Medientechnologien
Fakultät 4 – Elektrotechnik und Informatik

Analyse der Sicherheit und der automatisierten Bereitstellung eines *On-Premises*-Clusters auf der Grundlage der Container-basierten Virtualisierung: *Kubernetes* im Wissenschaftsbetrieb

Masterarbeit im Studiengang Informatik (KSS) in Kooperation mit dem Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar- und Meeresforschung

Autor:	Fabian Mangels <fabian@mangels.it> Matrikel-Nr.: 5074215
Ort / Datum der Abgabe:	Bremen, den 15.09.2020
Erstgutachter:	Prof. Dr. Lars Braubach
Zweitgutachter:	Dipl.-Inf. (FH) Jörg Matthes
Betrieblicher Betreuer:	Angelo Steinbach

Inhaltsverzeichnis

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während der Anfertigung der Masterarbeit unterstützt und motiviert haben. Auch in den Zeiten der *Corona*-Pandemie hat dies hervorragend funktioniert.

Danken möchte ich zuallererst meinem Gutachter der *Hochschule Bremen*, Herrn Prof. Dr. Lars Braubach, welcher meine Abschlussarbeit und somit auch mich betreut hat. Durch das zweimalig stattgefundene Masterseminar während der Ausarbeitung in einem erweiterten Kreis von Studierenden und Lehrenden konnte ein Austausch im wissenschaftlichen Kontext erfolgen. Auch Probleme während der Ausarbeitung konnten hier – als auch persönlich – konstruktiv erörtert werden.

Danken möchte ich auch meinen Arbeitskollegen vom *Alfred-Wegener-Institut*. Namentlich sind dies mein betrieblicher Gutachter, Herr Dipl.-Inf. (FH) Jörg Matthes, und mein betrieblicher Betreuer, Herr Angelo Steinbach. Es wurde mir ermöglicht, während der Bearbeitung der Masterarbeit als studentische Hilfskraft im Rechenzentrum des AWIs zu arbeiten und letztendlich auch das Thema und den Kontext der Ausarbeitung zu wählen. Ich konnte stets auf Mithilfe bei Problemen als auch auf langjährige Erfahrungen im Themenkomplex (Server-)Virtualisierung / Anwendungsbereitstellungen zurückgreifen. Auch die Zusammenarbeit im relativ neuen Bereich der Container-Technologien war konstruktiv und für mich bereichernd.

Ein ganz besonderer Dank gilt außerdem meinen Eltern, Anke und Bodo Mangels, die mich im Studium finanziell unterstützt haben und in der gesamten Zeit immer Vertrauen, Verständnis und Hilfe entgegen gebracht haben. Sehr dankbar bin ich ebenfalls meiner Zwillingsschwester, Sabrina Mangels, die mir bei Zweifeln oder Problemen stets beratend und geduldig zur Seite stand.

Zusammenfassung

Thema

Analyse der Sicherheit und der automatisierten Bereitstellung eines *On-Premises*-Clusters auf der Grundlage der Container-basierten Virtualisierung: *Kubernetes* im Wissenschaftsbetrieb.

Stichworte

Virtualisierung, Container, Orchestrierung, *Microservices*, Sicherheit, *On-Premise*, *DevOps*, *Cloud Native*, *VMware vSphere*, *Docker*, *Kubernetes*, *Rancher*, *Harbor*, *GitLab*, *vRealize Suite*

Kurzzusammenfassung

Diese Ausarbeitung beschäftigt sich mit dem vielschichtigen Themenkomplex der Container-Technologien sowie der Bereitstellung von *On-Premises*-Clustern und den dort platzierten Software-Anwendungen. Das Ziel besteht darin eine skalierbare Container-Infrastruktur im Rechenzentrum des AWIs und innerhalb des organisationsübergreifenden HIFIS-Projektes zu entwerfen. Dabei sind von besonderer Bedeutung die sichere und möglichst automatisierbare Gestaltung aller notwendigen Interaktionsplattformen (*Image Registry*, *Cluster Management*, *Self-Service Portal*, *CI / CD Pipelines*). In der Domäne der Container-Technologien spielen der *Microservice*-Architekturstil, die Container-Virtualisierung sowie -Orchestrierung eine entscheidende Rolle, um einer konsistenten Bereitstellungsplattform für *Cloud Native*-Anwendungen und der vermehrten *DevOps*-Prozesse in der Software-Entwicklung gerecht zu werden. Konkret realisiert werden diese aufgezeigten Aspekte durch den Einsatz von *Kubernetes* und *Docker* in einem Cluster-Verbund. Für die Analyse der Informationssicherheit dieser Container-Infrastruktur wurden Maßnahmen des BSI aus dem IT-Grundschutz (Baustein: *SYS.1.6: Container*) und das abstrakte 4-Schichtenmodell „*The 4C's of Cloud Native Security*“ herangezogen. In diesem Zusammenhang wurde auch die Automatisierung und die Entwicklung der Rechenzentren hin zu SDDCs in Folge der stetig anhaltenden digitalen Transformation thematisiert. Letztendlich erfolgte die Realisierung der Infrastruktur vollständig in der zugrunde liegenden *VMware vSphere*-Umgebung mit diesen gewählten Interaktionsplattformen: *Harbor* als *Image Registry*, *Rancher* mit RKE als *Cluster Management*, die *VMware vRealize Suite* (*vRA*, *vRO*) als *Self-Service Portal* und *GitLab* als *CI / CD Pipeline*. Für die anschließende Sicherheitsbewertung der umgesetzten Maßnahmen in den bereitgestellten *K8s*-Clustern, wurde die anerkannte CIS *Kubernetes Benchmark* herangezogen und zufriedenstellend bestanden. Zudem wurden qualitative Aussagen der Nutzerschaft bzgl. der Verwendbarkeit durch eine interaktive Umfrage ermittelt. Obwohl sich im Zuge der Evaluation noch weitere Maßnahmen im vielfältigen und komplexen Bereich der Container-Technologien angekündigt haben, konnte ein relativ stabiles *Kubernetes*-Umfeld mit den benannten Infrastruktur-Komponenten für den Wissenschaftsbetrieb am AWI und den organisationsübergreifenden Austausch im HIFIS-Projekt erfolgreich realisiert werden.

Abstract

Topic

Analysis of the security and automated provisioning of an on-premises cluster on the basis of container-based virtualization: *Kubernetes* in the scientific community.

Keywords

Virtualization, Container, Orchestration, Microservices, Security, On-Premise, *DevOps*, *Cloud Native*, *VMware vSphere*, *Docker*, *Kubernetes*, *Rancher*, *Harbor*, *GitLab*, *vRealize Suite*

Short summary

This paper deals with the complex subject of container technologies and the provision of on-premises clusters and the software applications placed there. The goal is to design a scalable container infrastructure in the AWI data center and within the cross-organizational HIFIS project. Of particular importance is the secure and, if possible, automatable design of all necessary interaction platforms (*Image Registry*, *Cluster Management*, *Self-Service Portal*, *CI / CD Pipelines*). In the domain of container technologies, the microservice architectural style, container virtualization and orchestration play a decisive role in order to meet the requirements of a consistent provisioning platform for *Cloud Native* applications and the increased *DevOps* processes in software development. These aspects will be realized through the use of *Kubernetes* and *Docker* in a cluster network. For the analysis of the information security of this container infrastructure, measures of the BSI from the IT-Grundschutz (module: *SYS.1.6: Container*) and the abstract 4-layer model "*The 4C's of Cloud Native Security*" were used. In this context, the automation and the development of data centers towards SDDCs as a result of the ongoing digital transformation was also addressed. Ultimately, the infrastructure was implemented entirely in the underlying *VMware vSphere* environment with these chosen interaction platforms: *Harbor* as *Image Registry*, *Rancher* with RKE as *Cluster Management*, the *VMware vRealize Suite* (*vRA*, *vRO*) as *Self-Service Portal* and *GitLab* as *CI / CD Pipeline*. For the subsequent security evaluation of the implemented measures in the provided *K8s* clusters, the recognized CIS *Kubernetes Benchmark* was used and passed satisfactorily. In addition, qualitative statements of the users regarding the usability were determined by an interactive survey. Although further measures in the diverse and complex field of container technologies were announced in the course of the evaluation, a relatively stable *Kubernetes* environment with the named infrastructure components for the scientific community at the AWI and the cross-organizational exchange in the HIFIS project could be successfully realized.

Abkürzungsverzeichnis

API *Application Programming Interface*, dt. Programmierschnittstelle

AWI *Alfred-Wegener-Institut*

Bin *Binary Executable*

BSI Bundesamt für Sicherheit in der Informationstechnik

CaaS *Container as a Service*

CD *Continuous Delivery / Continuous Deployment*

CI *Continuous Integration*

CIS *Center for Internet Security*

CLI *Command-Line Interface*, dt. Kommandozeile

CNCF *Cloud Native Computing Foundation*

CNI *Container Network Interface*

CPU *Central Processing Unit*

CVE *Common Vulnerabilities and Exposures*

DevOps *Development and IT Operations*

DNS *Domain Name System*

DoS *Denial-of-Service*

ECP *Enterprise Container Platform*

FT *Fault Tolerance*

HA *High Availability*

HIFIS *Helmholtz Infrastructure for Federated ICT Services*

HTTP *Hypertext Transfer Protocol*

HTTPS *Hypertext Transfer Protocol Secure*

IaaS *Infrastructure as a Service*

IaC *Infrastructure as Code*

ICT *Information and Communications Technology*

IKT Informations- und Kommunikationstechnologie

IP *Internet Protocol*

IPC *Interprocess Communication*

IT Informationstechnologie

JS *JavaScript*

K8s *Kubernetes*

LB *Load Balancer*

NSX *Network and Security Virtualization*

OS *Operating System*

OVF *Open Virtualization Format*

PaaS *Platform as a Service*

PKI *Public Key Infrastructure*

PSP *Pod Security Policy*

RAM *Random-Access Memory*

RBAC *Role-Based Access Control*

REST *Representational State Transfer*

RKE *Rancher Kubernetes Engine*

SaaS *Software as a Service*

SDDC *Software-Defined Datacenter*

SSH *Secure Shell*

TLS *Transport Layer Security*

URL *Uniform Resource Locator*

VM *Virtual Machine*

vRA *(VMware) vRealize Automation*

vRO *(VMware) vRealize Orchestrator*

XaaS *Everything as a Service*

YAML *YAML Ain´t Markup Language*

Abbildungsverzeichnis

Tabellenverzeichnis

Auflistungsverzeichnis

1 Einleitung

Das im Jahr 1980 gegründete *Alfred-Wegener-Institut* (AWI) ist eine bedeutende Forschungseinrichtung in der Polar- und Meeresforschung, welches seinen Hauptstandort in Bremerhaven hat. Helgoland, Oldenburg, Potsdam und Sylt zählen zu weiteren Außenstellen. In den Anfängen beschäftigte die Stiftung öffentlichen Rechts nur wenige Mitarbeiter, mittlerweile sind es mehr als 1000 Beschäftigte in den verschiedensten Disziplinen. An den Standorten wird vorwiegend in den Fachbereichen Geo-, Bio- und Klimawissenschaften geforscht. Unterstützt werden die ForscherInnen dabei durch eine leistungsfähige Infrastruktur, die auch die weit entfernten Stationen in der Arktis und Antarktis zugänglich macht sowie Schiffe und Flugzeuge aufweist. Das AWI ist ein international anerkanntes Kompetenzzentrum, welches die deutsche Polarforschung koordiniert, aber auch die Nordsee und ihre deutschen Küstenregionen erforscht. [AWI19]

In der bremischen Seestadt befindet sich auch das Hauptrechenzentrum, in dem der praktische Anteil dieser Masterarbeit durchgeführt wurde.

Die Arbeits- und Forschungsprozesse in einer Wissenschaftseinrichtung basieren immer stärker auf Informations- und Kommunikationstechnologien (IKT). In Folge der Digitalisierung werden bereits heute die Methoden und das Vorgehen der WissenschaftlerInnen beeinflusst. Um den Anforderungen der Zukunft gewachsen zu sein, müssen die zur Verfügung gestellten Forschungsinfrastrukturen weiterentwickelt bzw. neugestaltet werden. Als Mitglied der *Helmholtz-Gemeinschaft*, welche ein Zusammenschluss von mehreren deutschen Forschungszentren ist, arbeitet das AWI zusammen mit weiteren Partnern an einer zukunftsorientierten Infrastruktur der Informationstechnologie (IT). Konkret wird dieses föderierte Vorhaben im Projekt *Helmholtz Infrastructure for Federated ICT Services* (HIFIS) behandelt und daraufhin entsprechende Maßnahmen erarbeitet. [Hel18, vgl. S. 9f.]

Zusammengefasst sind die in HIFIS vorherrschenden Themen:

- „eine nahtlose, Zentren-übergreifende IT-Infrastruktur mit integrierten ICT-Dienstleistungen auf der Basis schneller Netze und einheitlichem Nutzerzugang“ [Hel18, S. 9]
- „einen in die Zusammenarbeits- und Forschungsprozesse integrierten sicheren, effizienten und weltweit verfügbaren Daten- und Anwendungszugriff auf der Basis von Cloud Diensten“ [Hel18, S. 9]
- „Ausbildung und Unterstützung, um qualitativ hochwertige und nachhaltige Software zu entwickeln und zu veröffentlichen“ [Hel18, S. 9]

Das Ziel dieser Masterarbeit ist es durch Mitarbeit in dem benannten HIFIS-Projekt, *On-Premises*-Cluster auf der Grundlage der Container-basierten Virtualisierung aufzubauen. Wie der Begriff *On-Premises* schon aussagt, sollen der oder die Cluster vor Ort im eigenen Rechenzentrum des AWIs betrieben werden. Sie sollen aber dennoch den *Cloud Native*-Ansatz verfolgen, der die Bereitstellung von Software-Anwendungen in den verschiedensten *Cloud Computing*-Infrastrukturen – v. a. durch containerisierte *Microservices* – ermöglicht. Einen vereinfachten Überblick des zu realisierenden Szenarios ist der Abbildung 1.1 zu entnehmen. Bei der Erstellung eines solchen Clusters soll der Fokus auf die zugrunde liegende Sicherheit und die automatisierte Bereitstellung über mehrere involvierte Interaktionsplattformen (*Interaction Platforms*) gesetzt werden.

Grundlegend sollen es die einzelnen Plattformen im Zusammenspiel ermöglichen, dass Dienste und Ressourcen des Rechenzentrums (*Corporate Datacenter*) in Form von Clustern angefordert bzw. „gebucht“ werden können. Des Weiteren soll dem / der NutzerIn (*DevOps*) der Zugriff auf eine zentrale Management-Schnittstelle möglich sein, um einen zuvor angeforderten Cluster grundlegend zu administrieren und zu überwachen, sowie Container-Anwendungen hierüber auszurollen. Ein direkter Zugriff auf die angeforderten Cluster-Ressourcen soll auch bestehen.

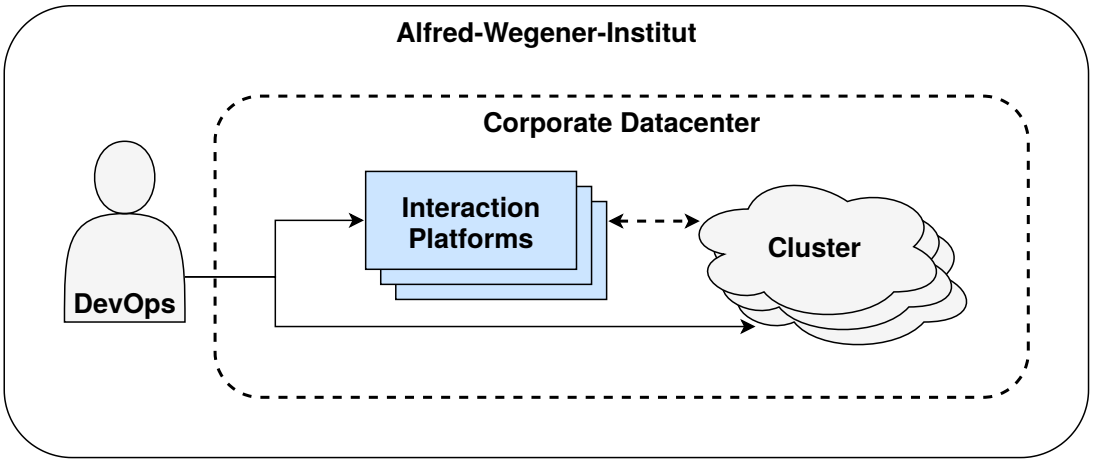


Abbildung 1.1: Vereinfachter Überblick des Szenarios

Die Umsetzung des angedachten Szenarios verfolgt dabei grundlegend zwei zentrale Aspekte: Die Schaffung einer konsistenten Bereitstellungsplattform für Container-Anwendungen – eine einheitliche Basis erleichtert zudem die organisationsübergreifende Zusammenarbeit – und der Übergang zu *Continuous Delivery* (CD). Letzteres bedeutet eine Veränderung in der Anwendungsentwicklung, denn es wird nicht mehr zwischen Entwicklungs- (*Dev*) und Betriebsteams (*Ops*) unterschieden. Beide Bereiche gehen ineinander über und arbeiten beginnend mit der Entwicklung, über die Bereitstellung, bis hin zur Pflege im gesamten Lebenszyklus einer Anwendung zusammen. Diese Arbeitsweise wird auch als *DevOps* bezeichnet [Wil15].

Für die Bereitstellung von Software-Anwendungen wird immer häufiger der *Microservice*-Architekturstil angewendet, der auch in dieser Ausarbeitung eine wichtige Rolle einnimmt und im Verbund mit der Container-Virtualisierung den aktuellen Trend nach der CNCF (*Cloud Native Computing Foundation*) in der Software-Entwicklung widerspiegelt [CNC20b]. *Microservices* lassen sich hiernach durch Container effizient und dynamisch abbilden, dennoch darf dabei der Sicherheitsgedanke nicht vernachlässigt werden. Überdies wird für die Bereitstellung von komplexen Container-Anwendungen eine automatisierte Orchestrierung notwendig. Es wird nämlich schwierig Container in einem System manuell zu verwalten, wenn die Anzahl dieser stetig zunimmt. Hierfür soll letztendlich *Kubernetes* zum Einsatz kommen, das einen Cluster bestehend aus mehreren Hosts als eine Bereitstellungsplattform für Container zusammenfasst.

Als Einstieg in diese Ausarbeitung werden im Kapitel 2 grundlegende Begriffe des Themenkomplexes Software-Architektur (Monolithen, *Microservices*) und die damit einhergehende Software-Bereitstellung mit Hilfe von VMs und Containern vermittelt. Außerdem wird auf die Dienstkomposition und dort im Speziellen auf die Orchestrierung von Container-Anwendungen eingegangen. Anschließend werden die zum Einsatz kommenden Komponenten der Infrastruktur im Kapitel 3 betrachtet und deren Verwendung im dargelegten Szenario aufgezeigt. Danach folgt im Kapitel 4 eine Auflistung der Anforderungen an die zu realisierenden Anwendungsfälle. Verwandte Arbeiten werden im darauffolgenden Kapitel 5 betrachtet, die eventuell Rückschlüsse auf die eigene Ausarbeitung ziehen lassen. Die nächsten beiden Kapitel stellen den Hauptteil der Ausarbeitung dar.

Zunächst wird im Kapitel 6 eine Analyse der möglichen Ansatzpunkte bezogen auf die Sicherheit und die automatisierte Bereitstellung eines *Kubernetes*-Clusters durchgeführt. Dies geschieht u. a. unter der Zuhilfenahme des vom Bundesamt für Sicherheit in der Informationstechnik (BSI) herausgegebenen IT-Grundschutz-Kompendiums und eines 4-Schichtenmodells. Das Kapitel 7 stellt den zweiten Hauptteil mit einem praktischen Anteil bereit. Konkret wird hier die Realisierung der Infrastruktur beschrieben, in der auch die Ergebnisse aus den vorherigen Kapiteln mit einfließen. In der im Kapitel 8 anschließenden Evaluation wird eine Cluster-Sicherheitsüberprüfung basierend auf der *Kubernetes Benchmark* des CIS (*Center for Internet Security*) vollzogen. Zusätzlich wird die umgesetzte Lösung bezogen auf die vorher definierten Anwendungsfälle und deren Verwendbarkeit kritisch bewertet und praktisch mit einigen NutzerInnen durchgeführt. Im letzten Kapitel 9 wird sich dem Fazit und Ausblick gewidmet. Die gesamte Ausarbeitung wird noch einmal kritisch begutachtet und abschließend zusammengefasst. Weitere Maßnahmen, die projektübergreifend und zukünftig Anwendung finden können, werden hier auch thematisiert.

2 Grundlagen

An dieser Stelle werden Grundlagen vermittelt, die für die nachfolgenden Kapitel zum Verständnis benötigt werden. Im Abschnitt 2.1 wird durch eine Gegenüberstellung von Monolithen und *Microservices* der Themenkomplex Software-Architektur beleuchtet. Danach wird im Abschnitt 2.2 auf die damit einhergehende Software-Bereitstellung mit Hilfe von verschiedenen Technologien fortgefahren. Abschließend folgt der Abschnitt 2.3, in dem auf die Dienstkomposition und dort speziell auf die Orchestrierung von Container-Anwendungen eingegangen wird. Zusätzliche Begriffe wie *DevOps* und *Cloud Native* werden in den restlichen Abschnitten 2.4 und 2.5 betrachtet.

2.1 Software-Architektur

Die traditionelle Software-Entwicklung und -Bereitstellung hat sich in den letzten Jahren verändert [CNC20b]. Dies hat Einfluss auf die Gestaltung vorhandener und zukünftiger Software-Architekturen. Eine Konsequenz ist die Aufteilung großer Anwendungen (Monolithen) in kleinere Bestandteile (*Microservices*) [Deh18]. Zumeist geht hiermit auch eine Änderung der Infrastruktur und Komposition der Software einher; dies wird in den Abschnitten 2.2 und 2.3 erläutert.

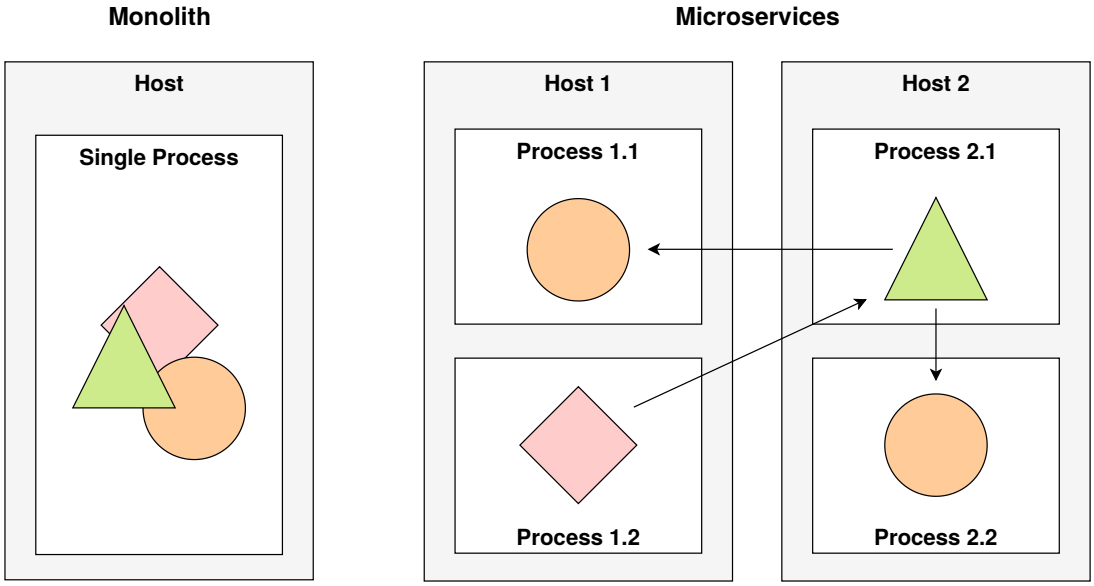


Abbildung 2.1: Gegenüberstellung eines Monolithen und dessen Aufspaltung in *Microservices* [Luk18, vgl. S. 5]

In der Abbildung 2.1 ist schematisch die Gegenüberstellung eines Monolithen und dessen Aufschlüsselung in mehrere *Microservices* dargestellt. Ein Monolith ist zumeist als ein einziger Prozess auf einem Host oder einer geringen Menge an Hosts realisiert. Aktualisierungen werden an den monolithischen Anwendungen tendenziell selten durchgeführt, dementsprechend sind die *Release*-Zyklen auch langsam. Einzelne Software-Komponenten – farbige Formen in den weißen

Vierecken – lassen sich allerdings separat betrachten. Hierdurch wird es möglich das homogene System in heterogene Strukturen zu überführen. Die so entstehenden *Microservices* separieren die jeweiligen Software-Komponenten in einzelne Prozesse und können daher schnell auf mehreren Hosts und sooft wie nötig in einem IKT-Netzwerk platziert werden. Letztendlich ermöglicht diese serviceorientierte Architektur im Kontrast zu Monolithen die individuelle Entwicklung, Bereitstellung, Aktualisierung und Skalierung einzelner Bestandteile.

2.1.1 Monolithen

Wie der Abbildung 2.1 entnommen werden kann, bestehen Monolithen aus sehr eng gekoppelten Komponenten. Da deren Bestandteile überwiegend in einem einzigen Betriebssystemprozess oder einer sehr geringen Anzahl an Prozessen laufen, werden monolithische Anwendungen auch als Ganzes entwickelt, bereitgestellt und verwaltet. Dies hat zur Konsequenz, dass bei Änderungen auch an nur einem Teil der Applikation, eine erneute Bereitstellung der gesamten Anwendung erforderlich ist. Aufgrund der nicht fest vorhandenen Separierung der Komponenten kann es mit der Zeit zu einer erhöhten Komplexität kommen. Werden zudem Abhängigkeiten zwischen den einzelnen Bestandteilen nicht eingeschränkt bzw. vermieden, kann dies zusammen zu einer sinkenden Qualität der Gesamtanwendung führen. [Luk18, vgl. S. 4f.]

Gewöhnlich setzt der Betrieb einer monolithischen Anwendung leistungsstarke Hardware voraus, um das Gesamtsystem mit ausreichenden Ressourcen zu versorgen. Nehmen die Anforderungen für die Bereitstellung eines Monolithen sukzessiv zu, gibt es zwei Arten für die Skalierung des zugrunde liegenden Hosts bzw. Servers. Entweder wird der Host vertikal skaliert, indem die zur Verfügung stehenden Ressourcen (CPU, RAM etc.) aufgestockt werden (*Scale-up*), oder die Gesamtanwendung wird durch mehrere Server, die jeweils eine eigene Instanz ausführen, horizontal skaliert (*Scale-out*). Die vertikale Skalierung erfordert keine Änderung an der eigentlichen Anwendung, dennoch wird diese Vorgehensweise relativ schnell teuer und funktioniert auch nur bis zu einer gewissen Obergrenze. Konträr dazu können bei der horizontalen Skalierung erhebliche Anpassungen am Quellcode notwendig sein. Manche Komponenten lassen sich nur schwierig oder auch gar nicht horizontal skalieren, wie z. B. relationale Datenbanken. Ein Monolith ist nicht skalierbar, wenn seine einzelnen Bestandteile nicht skaliert werden können, außer, genau diese Teile lassen sich auf irgendeine Weise aufteilen. [Luk18, vgl. S. 5]

Trotz der durchaus negativen aufgezeigten Aspekte gibt es auch Architektur-Entwicklungen, die eine *Monolith-First*-Strategie befürworten, bei der neue Anwendungen zunächst als Monolith aufgebaut werden, auch wenn davon auszugehen ist, dass später eine *Microservice*-Architektur sinnvoller erscheint. Für den Aufbau von einfachen Anwendungen sei ebenso ein Monolith begünstigt einzusetzen. Ein direkter Weg zu *Microservices* kann laut Fowler riskant sein, wenn die Komplexität der zu erschaffenden Anwendung und die Grenzen der Komponenten noch nicht bekannt sind. Nimmt die Komplexität und das erlangte Wissen über das Vorhaben zu, können einzelne Bestandteile herausgelöst werden und schlussendlich als *Microservices* agieren. [Fow15b]

2.1.2 *Microservices*

Im Unterabschnitt 2.1.1 wurde bereits erwähnt, dass monolithische Software-Architekturen für Unternehmen mit zunehmender Laufzeit oftmals zu groß und unübersichtlich werden, um damit weiterzuarbeiten. Sie sind angehalten, ihren Architekturstil für aktuelle als auch zukünftige Software-Anwendungen zu überdenken. Ein Erfolg versprechender Ansatz stellt *Microservices* in den Vordergrund. Vorhandene homogene Software-Systeme können damit in kleine unabhängige

Komponenten zerlegt werden, die wiederum zu einer Verbesserung der Gesamtarchitektur beitragen [Deh18]. Andererseits können neue Software-Projekte schon in den Anfängen mit Hilfe von *Microservices* strukturiert werden.

Nach Fowler ist der *Microservice*-Architekturstil „ein Ansatz zur Entwicklung einer einzelnen Anwendung als eine Folge kleiner Dienste, die jeweils in einem eigenen Prozess laufen und mit leichtgewichtigen Mechanismen [...] kommunizieren. Diese Dienste [...] sind durch vollautomatische Bereitstellungsmechanismen unabhängig voneinander einsetzbar. Es gibt nur ein Minimum an zentralisierter Verwaltung dieser Dienste, die in verschiedenen Programmiersprachen geschrieben sein und unterschiedliche Datenspeichertechnologien verwenden können“ [FL19, Übers.].

Die Abbildung 2.1 zeigt, dass *Microservices* als unabhängige Prozesse ausgeführt werden, die durch einfache und wohldefinierte APIs (*Application Programming Interface*, dt. Programmierschnittstelle) untereinander kommunizieren. Der Austausch erfolgt entweder über synchrone Protokolle wie HTTP (*Hypertext Transfer Protocol*) und der Bereitstellung einer REST-fähigen (*Representational State Transfer*) API, oder über asynchrone Protokolle wie AMQP (*Advanced Message Queuing Protocol*) und der Interaktion mit einem *Message Broker*. Diese Protokolle sind an keine Programmiersprachen gebunden, daher kann ein entsprechender *Microservice* jeweils in der Sprache geschrieben werden, die dem Entwickler am geeignetsten erscheint. Aufgrund der Souveränität eines Dienstes und der relativ statischen externen API, lassen sich die individuellen Komponenten getrennt voneinander entwickeln und bereitstellen. Bezogen auf die Änderung einer Komponente sind Anpassungen oder Neubereitstellungen anderer *Microservices* grundsätzlich nicht notwendig, vorausgesetzt, dass sich die API für einen externen Zugriff nicht oder nur in einer abwärtskompatiblen Weise verändert. [Luk18, vgl. S. 5f.]

Microservices lassen sich im Gegensatz zu monolithischen Anwendungen einzeln skalieren – es ist also keine Skalierung des Gesamtsystems erforderlich. Dies ermöglicht nur die Komponenten zu vergrößern oder zu verkleinern, die auch wirklich eine Anpassung der Ressourcen benötigen. Alle anderen Bestandteile einer *Microservice*-Anwendung können dementsprechend unangetastet bleiben. [Luk18, vgl. S. 6]

Die Verwaltung von bereitzustellenden *Microservices* bleibt bei einer geringen Anzahl noch überschaubar, denn die Möglichkeiten, an welchem Ort die Dienste platziert werden können, sind begrenzt und größtenteils trivial. Mit steigender Anzahl an Komponenten und einer größeren Bereitstellungsplattform werden diese Entscheidungen hingegen immer schwieriger, da die Kombinationen und Abhängigkeiten zwischen den Komponenten exponentiell zunehmen. Einzelne Dienste arbeiten im Verbund, weshalb sie sich finden und miteinander kommunizieren müssen. Damit sie als ein System zusammenarbeiten können, wird eine ordnungsgemäße Konfiguration benötigt, aus der hervorgeht, auf welche Art und Weise die Dienste miteinander in Verbindung stehen (s. Abschnitt 2.3). Erfolgt dies bei zunehmender Anzahl an *Microservices* nicht automatisiert, wird die Orchestrierung aufwendig und fehleranfällig. Die Entwicklung von eigenverantwortlichen *Microservices* wird gewöhnlich von unterschiedlichen Teams durchgeführt. Diese sind souverän in der Entscheidung, beliebige Bibliotheken einzusetzen und nach freiem Ermessen zu wechseln. Folglich führt das zu unterschiedlichen Abhängigkeiten der Komponenten innerhalb einer Gesamtanwendung. Die Notwendigkeit von verschiedenen Versionen derselben Bibliothek in einer Bereitstellungsumgebung ist daher unvermeidbar. Je mehr Komponenten auf einem Host bereitgestellt werden müssen, desto schwieriger wird die Verwaltung der Abhängigkeiten an Bibliotheken und Laufzeitumgebungen, um die individuellen Anforderungen zu realisieren. Hier ist eine flexible und isolierte Bereitstellungsumgebung unabdingbar, wie sie im Unterabschnitt 2.2.3 mit der Container-Bereitstellung behandelt wird. [Luk18, vgl. S. 6ff]

2.2 Software-Bereitstellung

Besonders wichtig für die Software-Bereitstellung ist das Vorhandensein einer konsistenten Umgebung, in der Anwendungen – unabhängig von der Anzahl der bereitzustellenden Komponenten – ausgeführt werden. Unterschiede können dabei sowohl zwischen Entwicklungs- und Produktionsumgebungen als auch unter den einzelnen Hosts auftreten. Die Hosts können sich aufgrund von verschiedener Hardware, Betriebssystemen oder Bibliotheken, die in diversen Versionen zur Verfügung stehen können, unterscheiden. Gewöhnlich werden Produktionsumgebungen von Betriebsteams (*Ops*) und Entwicklungssysteme von den Entwicklern (*Dev*) selbst verwaltet und eingerichtet. Dies führt zu relativ großen Unterschieden in beiden Umgebungen, da die Personengruppen verschiedene Gewichtungen vornehmen. Ideal wäre der Zustand, in dem Anwendungen in der Entwicklung und Produktion in exakt der gleichen Umgebung ausgeführt werden, um die Anzahl auftretender Probleme im Anwendungskontext zu vermindern. Dies schließt identische Betriebssysteme, Bibliotheken, Systemkonfigurationen, Netzwerkumgebungen etc. mit ein. Überdies wäre es von Vorteil, wenn die Umgebungen sich mit der Zeit kaum verändern würden – sprich konsistent bleiben. Es sollte die Möglichkeit bestehen, einem Server zusätzliche Software-Komponenten hinzuzufügen, ohne dabei bereits vorhandene Bereitstellungen zu beeinträchtigen. [Luk18, vgl. S. 8]

Grundsätzlich kann die Bereitstellung von Software durch zwei in den Ansätzen konkurrierende Konzepte realisiert werden. Auf der einen Seite ist dies das klassische Betreiben eines eigenen Rechenzentrums, in dem Server-Ressourcen durch eigenes Personal und zur Verfügung stehendes *Know-how* bereitgestellt werden. Da sämtliche Leistungen vor Ort und unter eigener Regie verantwortet werden, wird auch in diesem Kontext von *On-Premise* gesprochen (s. Tabelle 2.1). Auf der anderen Seite steht der Begriff *Cloud Computing*, durch den die Verantwortlichkeiten einzelner Leistungen zumeist an einen Anbieter (*Provider*) abgetreten werden. *Cloud Computing* „ist ein Konzept der Informationstechnik (IT), das IT-Ressourcen virtualisiert [...] und als skalierbare, abstrahierte Infrastrukturen, Plattformen und Anwendungen on-demand bei nutzungsabhängiger Abrechnung zur Verfügung stellt“ [SBBK15, S. 459]. Wesentlich werden drei verschiedene Organisationsformen von *Cloud*-Systemen unterschieden [SBBK15, vgl. S. 460f.]:

- *Public Cloud* –
  Bei einer öffentlichen *Cloud* wird ein Zugang zu abstrahierten IT-Infrastrukturen für die breite Öffentlichkeit über das Internet gewährt. Die Dienstanbieter und -nutzer gehören dabei unterschiedlichen Organisationen an. Der Anbieter verfolgt kommerzielle Ziele und den Kunden werden nur aktuell genutzte Ressourcen in Rechnung gestellt, ohne Kosten für die Anschaffung, den Betrieb und die Wartung eigener Hardware zu verlangen.

- *Private Cloud* –
  Bei einer *Private Cloud* hingegen stammen die Dienstanbieter und -nutzer meistens aus der gleichen Organisation. Dieses *Cloud*-System wird also ausschließlich für eine Organisation betrieben. Nachteilig sind hier die anfallenden Kosten für die zu stellende Hardware – ähnlich wie bei nicht-*Cloud*-basierten Architekturen. Umgekehrt sind hier die Bedenken hinsichtlich der Datensicherheit und des Datenschutzes durch die eigene Kontrolle sorgenfreier zu akzeptieren.

- *Hybrid Cloud* –
  Die Organisationsform einer *Hybrid Cloud* verwendet sowohl Dienste einer öffentlichen *Cloud* als auch die Vorzüge privater Umgebungen ganz nach den Bedürfnissen des Kunden.

Welche Bereitstellungsmethode – *Cloud Computing* oder *On-Premise* – ausgewählt wird, hängt von den jeweiligen Anforderungen ab; eine Kombination beider Ansätze in Form einer *Private* oder *Hybrid Cloud* in einem *On-Premise*-Umfeld ist jedoch auch möglich. Relevant sind hierfür u. a.

Aspekte wie finanzielle Vorteile, Agilität von Leistungen (Dienstleistungsumfang), Zuverlässigkeit und Skalierbarkeit. Zusätzlich sollten immer die Kriterien Datensicherheit, Datenschutz und die Abhängigkeit zu einem *Cloud*-Anbieter (*Lock-in*) hinterfragt werden.

Tabelle 2.1: Übersicht der *XaaS*-Modelle des *Cloud Computings* [Lie19, vgl. S. 56]

	On Premise	***Infrastructure*** (*as a Service*)	***Containers*** (*as a Service*)	***Platform*** (*as a Service*)	***Software*** (*as a Service*)
Applications	✓	✓	✓	✓	✗
Data	✓	✓	✓	✓	✗
Runtime	✓	✓	✓	✗	✗
Middleware	✓	✓	✓	✗	✗
OS	✓	✓	✗	✗	✗
Virtualization	✓	✗	✗	✗	✗
Servers	✓	✗	✗	✗	✗
Storage	✓	✗	✗	✗	✗
Networking	✓	✗	✗	✗	✗
✓ = *Self-Managed*, ✗ = *Provider-Supplied*					

In der Tabelle 2.1 sind verschiedene Service-Modelle des *Cloud Computings* in Bezug zum *On-Premise*-Ansatz gegenübergestellt. Die Kategorisierung erfolgt anhand der Funktionalität der Dienste, was auch als *Everything as a Service* (*XaaS*) bezeichnet wird. Alle Arten von Ressourcen werden als Dienst bzw. Service zur Verfügung gestellt und von Kunden letztendlich konsumiert. Die dabei wichtigsten Kategorien sind [SBBK15, vgl. S. 461ff]:

- *Infrastructure as a Service* (*IaaS*) –
  Die Infrastruktur-Dienste bieten die Möglichkeit an virtuelle Server-Instanzen basierend auf beliebigen Betriebssystemen mit eigenen Anwendungen auf den Hosts des Anbieters zu betreiben. Sogar komplette Rechenzentren lassen sich hierüber virtuell realisieren.

- *Platform as a Service* (*PaaS*) –
  Die Plattform-Dienste bieten für einen Kunden hingegen skalierbare Programmierungs- und Laufzeitumgebungen für die Entwicklung bzw. Bereitstellung eigener Software an, allerdings ohne den entsprechenden Administrationsaufwand für virtuelle Server-Instanzen wie es beim *IaaS*-Modell der Fall ist.

- *Software as a Service* (*SaaS*) –
  Noch einschränkender sind nur noch die Software-Dienste, die dem Kunden ausschließlich Anwendungen eines Dienstleisters über das Internet zur Verfügung stellen. Für die Verwendung der Anwendungen wird lediglich ein Webbrowser benötigt.

SaaS ist bezogen auf die Entscheidungsfreiheit eines Kunden mit Abstand das einschränkendste *XaaS*-Modell. Im Kontrast dazu bietet der *On-Premise*-Ansatz die größte Entscheidungsfreiheit für den Kunden an, dennoch geht hiermit auch die größte Verantwortlichkeit für die Bereitstellungsumgebung einher. Im Zuge des *Serverless Computings* haben sich noch weitere Service-Modelle im *Cloud Computing* entwickelt. Zu nennen sind hier *FaaS* (*Function as a Service*) und *BaaS* (*Backend as a Service*), welche in der Tabelle 2.1 zwischen *PaaS* und *SaaS* anzusiedeln sind [Lie19, vgl. S. 991f.]. Unter *Serverless Computing* wird nicht die Abwesenheit von Servern verstanden, sondern ein Konzept, welches wieder spezielle Dienstleistungen in der *Cloud* anbietet. Im Hintergrund sind natürlich Virtualisierungsumgebungen mit Servern beteiligt, von denen der Kunde jedoch nichts mitbekommt. Durch die wachsende Verbreitung

von Container- und Orchestrierungslösungen (s. Unterabschnitt 2.2.3, Unterabschnitt 2.3.1) hat sich der klassische Einsatzschwerpunkt von *Cloud*-Umgebungen verlagert. Die Bereitstellung von virtuellen Maschinen (VMs) weicht zunehmend der Umsetzung durch Container [CNC20b]. *IaaS*-Landschaften fungieren dementsprechend als Plattformen für CaaS (Container as a Service). [Lie19, vgl. S. 55f.]

Nachfolgend werden die drei wesentlichen Technologien physische Server (s. Unterabschnitt 2.2.1), VMs (s. Unterabschnitt 2.2.2) und Container (s. Unterabschnitt 2.2.3) hinter der Software-Bereitstellung thematisiert.

2.2.1 Traditionelle Bereitstellung

Ursprünglich wurde die Software-Bereitstellung in Unternehmen nur auf rein physischen bzw. *Bare Metal*-Servern (Hosts) durchgeführt. Die Abbildung 2.2 kann für diese einfache Bereitstellungsform zur Versinnbildlichung dienen. Ein traditioneller Host weist in diesem Kontext nur Hardware-Bestandteile, ein Betriebssystem (*Operating System*) und die bereitgestellten Applikationen (Farbige Formen) auf. Dieses Vorgehen verursacht allerdings Probleme bei der Ressourcenzuweisung, denn es gibt ohne Weiteres keine Möglichkeit, Ressourcengrenzen auf den physischen Servern für die Anwendungen festzulegen. Bspw. kann es in einem Szenario mit mehreren Applikationen auf dem selben Host dazu führen, dass eine Applikation zu viele Ressourcen in Anspruch nimmt und infolgedessen alle anderen Software-Bereitstellungen nicht mehr die volle Leistung erbringen können. Um das Konfliktpotential zu vermeiden bzw. zu separieren, wäre es möglich jede Anwendung auf einem anderen physischen Server laufen zu lassen. Die Ressourcen auf den Hosts wären allerdings nicht ausgelastet, was zu einer ineffizienten Skalierung führt. Außerdem ist der Betrieb und die Wartung vieler physischer Server für Unternehmen teuer und aufwendig. [Kub20m]

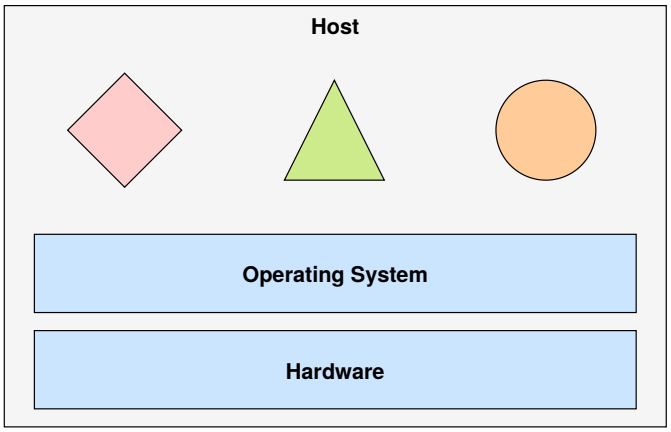


Abbildung 2.2: Traditionelle Bereitstellung auf einem einzelnen Host [Kub20m]

2.2.2 Virtualisierte Bereitstellung

Um die Probleme der traditionellen Bereitstellung aufzuheben, wurde die Server-Virtualisierung eingeführt [WAG+18, vgl. S. 37ff], [Kub20m]. Mit ihr ist es möglich mehrere virtuelle Maschinen (VMs) auf der Hardware eines einzigen physischen Servers auszuführen, denn die Ebene des Betriebssystems wird mit dieser Technologie von der Hardware-Ebene abstrahiert. Jede VM ist dabei wie ein „vollständiger“ physischer Server, auf dem alle Komponenten wie Bibliotheken und

Programme (*Bin / Library*), einschließlich des eigenen Gastbetriebssystems (*Guest OS*), auf der virtualisierten Hardware ausgeführt werden.

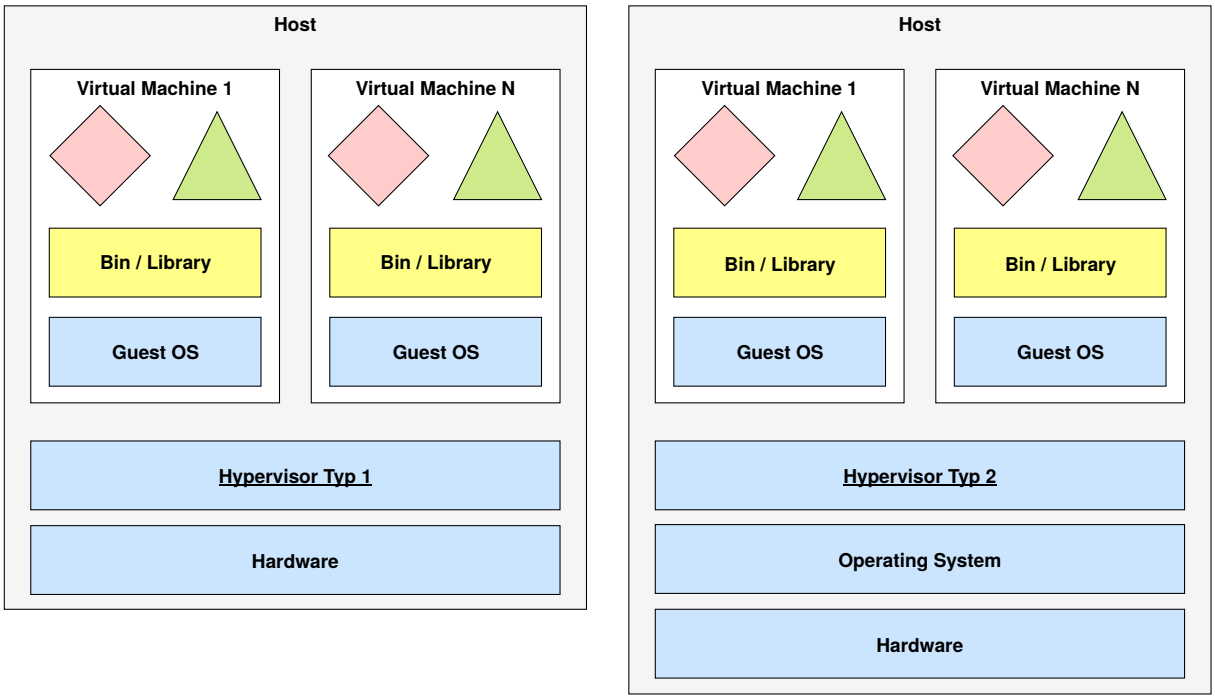


Abbildung 2.3: Virtualisierte Bereitstellung auf einem einzelnen Host durch verschiedene *Hypervisor*-Typen [Kub20m], [WAG+18, vgl. S. 40]

Die zugrunde liegende Architektur eines solchen Wirts (Host) mit den dazugehörigen Gastsystemen (VMs) ist in der Abbildung 2.3 ersichtlich. Wie schon bei der traditionellen Bereitstellung gibt es auch hier die Hardware als Basis des Hosts und nur im Fall des *Hypervisor Typ 2* ein darüber liegendes Betriebssystem (*Operating System*). Als Baustein neu hinzugekommen ist der *Hypervisor*, der letztendlich die Verwaltung der VMs verantwortet. Er wird auch als *Virtual Machine Manager* (VMM) bezeichnet und kommt in zwei Varianten vor [WAG+18, vgl. S. 40]:

- *Hypervisor Typ 1*, *Bare Metal* –
  Hypervisor diesen Typs werden unmittelbar als Betriebssystem auf der Hardware installiert. Sie sind für die Server- oder auch die Desktop-Virtualisierung prädestiniert. Durch den Wegfall des eigentlichen Betriebssystems, ist diese Technologie sehr schlank und minimiert den zwischen den Schichten anfallenden *Overhead*.

- *Hypervisor Typ 2*, *Hosted* –
  Diese *Hypervisor* werden in ein bestehendes Betriebssystem installiert und setzen daher nicht direkt auf die Hardware des Hosts auf. Diese Art der Virtualisierung wird häufig für Testsysteme genutzt, die innerhalb eines bereits bestehenden Computers verwendet werden.

Die virtualisierte Bereitstellung bewirkt, dass Anwendungen zwischen virtuellen Maschinen isoliert werden können. Dies bietet ein gewisses Maß an Sicherheit, da auf den Ausführungskontext einer Applikation nicht frei von einer anderen zugegriffen werden kann. Darüber hinaus ermöglicht die Virtualisierung eine flexible Verteilung der Ressourcen auf einem physischen Server und reduziert daraufhin die anfallenden Hardware-Kosten im Gegensatz zur traditionellen Bereitstellung. Durch diese Konsolidierung ist somit auch eine bessere Skalierbarkeit von Anwendungen möglich, da im Bedarfsfall Ressourcen optimal verteilt werden. Ein weiterer Vorteil ist die schnellere Konfiguration und Bereitstellung einer VM gegenüber dem Aufbau eines physischen Servers. Es

wird einiges an Aufwand gespart und die Ressourcenzuteilung kann noch im Nachgang – teilweise sogar zur Laufzeit – „elastisch“ angepasst werden. Die Flexibilität beschränkt sich nicht nur auf die virtuellen Maschinen, sondern schließt auch die Wirte bzw. Hosts mit ein. Bei Ausfall eines Hosts können z. B. betroffene VMs im laufenden Betrieb auf einem anderen physischen Server verschoben werden, ohne dass das Gastbetriebssystem Auswirkungen wahrnimmt. [WAG+18, vgl. S. 38f.]

Neben der Virtualisierung mit einem *Hypervisor* gibt es noch die Hardware- und die Betriebssystem-Virtualisierung. Wie der Name schon vorgibt, wird bei der Hardware-Virtualisierung die Verwaltung von Gastsystemen durch die Hardware selbst übernommen. Die Betriebssystem-Virtualisierung hingegen hebt sich von den bereits thematisierten Varianten der Virtualisierung stark ab. Sie partitioniert ein einzelnes Betriebssystem und lässt die entstehenden Instanzen isoliert voneinander laufen. Die Instanzen – auch als Container bezeichnet – sind zwingend von demselben Betriebssystem abhängig, da die Basis dieselbe ist. Eine ausführlichere Betrachtung dieser auch als Container-Virtualisierung genannten Software-Bereitstellung wird im Unterabschnitt 2.2.3 vorgenommen. [WAG+18, vgl. S. 40f.]

2.2.3 Container-Bereitstellung

Die Virtualisierung mit Hilfe von Containern erlaubt es mehrere Dienste bzw. *Microservices* auf demselben Host zu platzieren. Jeder Anwendung kann hierüber eine eigene isolierte Umgebung bereitgestellt werden. Nach Mouat sind Container „eine Kapselung einer Anwendung mit ihren Abhängigkeiten“ [Mou16, Übers. S. 3]. In Bezug auf andere Virtualisierungstechnologien, wie die der *Virtual Machines* (VM, s. Unterabschnitt 2.2.2), können Ressourcen direkt mit dem Host geteilt und die Hardware-Kosten somit noch einmal gesenkt werden; Container verwenden nämlich denselben Kernel wie der Host. Obwohl die Ausführung der Prozesse, welche in den Containern laufen, innerhalb des Host-Betriebssystems stattfindet, bleiben die Prozesse voneinander getrennt. Für einen Container-Prozess sieht es so aus, als ob dieser isoliert auf einem eigenen Host mit Betriebssystem läuft. [Luk18, vgl. S. 10f.], [Mou16, vgl. S. 3ff]

Die Abbildung 2.4 verdeutlicht, auf welche Schichten die Container-Virtualisierung aufbaut, damit die Bereitstellung von Containern auf einem Host als sogenanntes *Single-Node* Container-System funktioniert [Lie19, vgl. S. 77ff]. Grundsätzlich sind dies die drei Bestandteile *Hardware*, *Operating System* und *Container Runtime*. Ein isolierter Container-Prozess – farbige Formen in den weißen Vierecken – kann sofort gestartet werden und beansprucht nur die Ressourcen, die die enthaltende Anwendung auch wirklich benötigt. Container können daher im Gegensatz zu VMs als „viel schlanker“ [Luk18, S. 11] angesehen werden. Es entsteht kein Mehraufwand für zusätzliche Prozesse, wie es bei einem VM-Gastbetriebssystem der Fall wäre. Im Vergleich können somit noch mehr Software-Komponenten auf demselben Host bereitgestellt werden. [Luk18, vgl. S. 11ff], [Mou16, vgl. S. 3ff]

Eine vollständige Isolation der Prozesse, wie es bei VMs der Fall ist, wird in Containern nicht geboten. Während VMs nur die Hardware gemeinsam nutzen, richten Container auf einem Host Aufrufe an denselben *Linux*-Kernel; genau dies stellt ein Sicherheitsrisiko dar. Bei eingeschränkten Hardware-Ressourcen sind virtuelle Maschinen nur dann sinnvoll, wenn eine geringe Anzahl von Prozessen isoliert werden soll. Um eine größere Menge an isolierten Prozessen auf einem Host zu realisieren, sind Container aufgrund des geringeren *Overheads* die bessere Entscheidung. Jede VM führt schließlich ihre eigenen Systemdienste aus, was bei Containern durch die gemeinsame Verwendung eines Betriebssystems nicht der Fall ist. Anders als bei VMs, muss für die Ausführung von Containern auch nichts hochgefahren werden. Ein Prozess in einem Container startet auf Anhieb. Die Container-Isolierung beruht rudimentär auf Namensräume (*Linux-Namespaces*) und

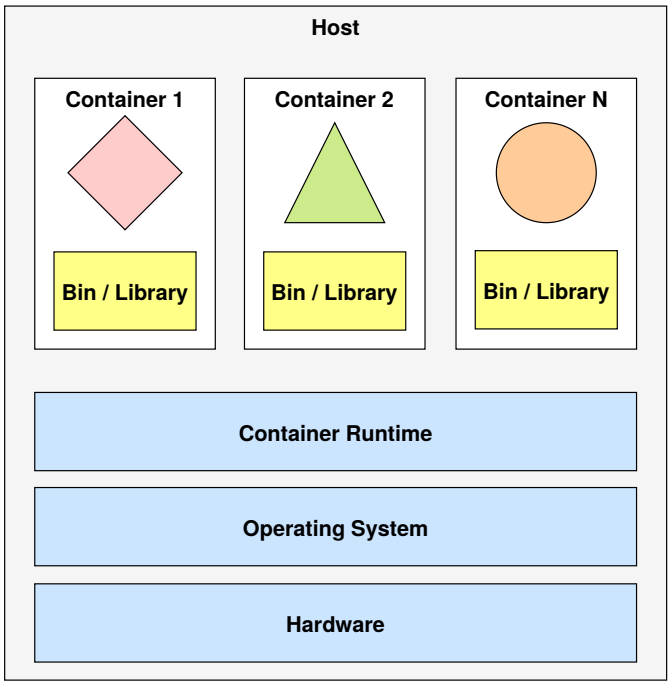


Abbildung 2.4: Container-Bereitstellung auf einem einzelnen Host [Kub20m], [Lie19, vgl. S. 114]

den Kontrollgruppen (*Linux-Cgroups*) des *Linux*-Kernels. Während Namensräume die Sicht des einzelnen Prozesses auf das System einschränken, verhindern die Kontrollgruppen die Menge an Ressourcen, die ein Prozess anfordern kann. Beide Mechanismen machen es erst möglich die ressourceneffiziente Container-Virtualisierung zu betreiben. Die Sicherheitsproblematik der nicht vollständigen Isolation der Container-Prozesse wird im Kapitel 6 noch einmal aufgegriffen. [Luk18, vgl. S. 13f.], [Lie19, vgl. S. 80ff]

2.3 Dienstkomposition

Hauptsächlich wird der Begriff der Dienstkomposition im Bereich der SOAs (*Service-Oriented Architecture*) angewendet [SBBK15, vgl. S. 118]. Dieses Konzept kann auch in den *Microservice*-Architekturstil grundlegend überführt werden. In diesem Kontext sind demzufolge die Container (*Microservices*) die elementaren Einheiten. Erfolgt zusätzlich die Bereitstellung von Komponenten nicht mehr nur auf einem Host, sondern auf einer Reihe von Hosts (Cluster), dann wird eine möglichst gleichmäßige Verteilung von *Microservices* erreicht. Die Dienstkomposition beschreibt letztendlich die Eigenschaft, wie diese Komponenten miteinander in Verbindung stehen. Unterschieden werden kann in die zwei Arten Orchestrierung (s. Unterabschnitt 2.3.1) und Choreographie (s. Unterabschnitt 2.3.2). Eine Dienstkomposition kann dabei aus einer oder beiden Arten bestehen. [SBBK15, vgl. S. 122ff]

Wie schon im Unterabschnitt 2.2.3 aufgezeigt wurde, können durch die Container-Virtualisierung im Vergleich zu VMs mehr Software-Komponenten auf demselben Host bereitgestellt werden. Allerdings wird es mit steigender Anzahl an verteilten Komponenten bzw. Containern und unter der Verwendung von Clustern in Rechenzentren immer schwieriger, diese Vielzahl an Instanzen zu administrieren und lauffähig zu halten. Container ressourcenschonend in *Microservice*-Architekturen bereitzustellen, ist manuell in einem größeren Umfang nicht praktikabel. Eine Automatisierung zur Bereitstellung und Verwaltung von Software-Komponenten und der

dazugehörigen Infrastruktur wird notwendig. Werkzeuge der Dienstkomposition im besonderen die Container-Orchestrierung können hier als Lösung agieren. [Luk18, vgl. S. 3f.]

2.3.1 Orchestrierung

Unter der Orchestrierung wird die flexible Schaffung einer neuen Gesamtanwendung verstanden, die aus vorhandenen Komponenten durch einen zentralen Koordinator (*Orchestrator*) gesteuert wird [SBBK15, vgl. S. 122]. Der *Orchestrator* nimmt dabei Anweisungen von außen entgegen und verteilt diese an die einzelnen Komponenten. Der Bedarf solcher Hilfsmittel zur Orchestrierung ergibt sich aus dem Trend, von monolithischen Strukturen zu einzelnen *Microservices* umzusteigen (s. Abschnitt 2.1). Projekte werden zunehmend im Fokus des Architekturstils geführt, der die Aufteilung von Software-Komponenten in *Microservice*-Containern vorgibt [CNC20b]. Diese Segmentierung ermöglicht dabei eine Skalierbarkeit der heterogenen Bestandteile im Gegensatz zu komplexen Monolithen. Außerdem können unterschiedliche Anforderungen der unabhängigen Komponenten bezogen auf die Bibliotheken unter Zuhilfenahme der Container-Bereitstellung gekapselt werden (s. Unterabschnitt 2.2.3). [Luk18, vgl. S. 4ff]

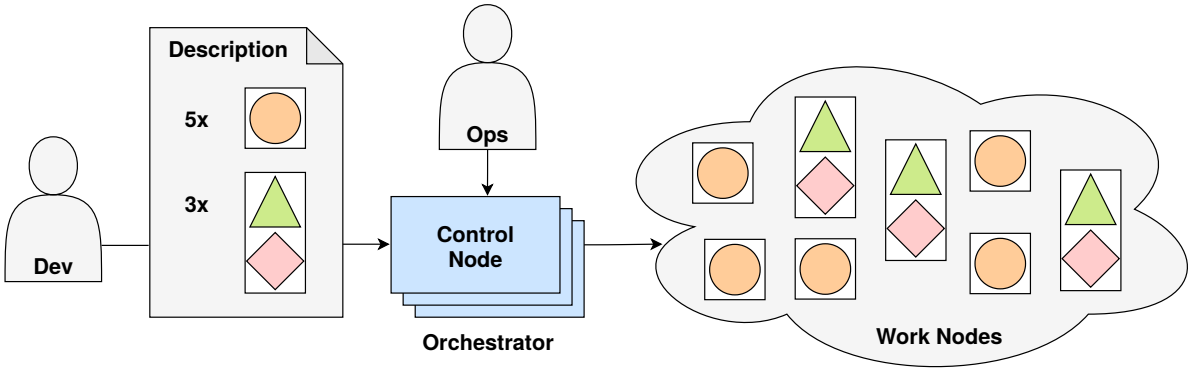


Abbildung 2.5: Container-Bereitstellung durch Orchestrierung als Dienstkomposition [Luk18, vgl. S. 20]

Die Abbildung 2.5 stellt vereinfacht die Kernaufgabe der Container-Orchestrierung dar. Ein Cluster bestehend aus mehreren Hosts wird als eine Bereitstellungsplattform zusammengefasst. In ihr wird konkret zwischen einigen *Control Nodes* und vielen *Work Nodes* als Host-Ausprägungen unterschieden. Entwickler können Beschreibungen ihrer Anwendungen (*Description*) bei einem *Control Node* einreichen. Daraufhin werden die spezifizierten Komponenten bzw. Container – farbige Formen in den weißen Vierecken – auf den Knoten bereitgestellt, die als *Work Nodes* klassifiziert sind. Welcher Knoten für eine einzelne Komponente verantwortlich ist, spielt dabei sowohl für den Entwickler (*Dev*) als auch Administrator (*Ops*) keine entscheidende Rolle. Wichtig ist nur, dass die bereitzustellenden Container ausgerollt werden und im Cluster-Netzwerk ggf. miteinander kommunizieren können. [Luk18, vgl. S. 20]

Durch die Verfügbarkeit der Container-Orchestrierung geht die Entwicklung vom einzelnen *Single-Node* Container-System (s. Abbildung 2.4) hin zu skalierbaren Container-Clustern. Für diese Verwirklichung sind verschiedene abstrahierte Bestandteile involviert, die der Abbildung 2.6 zu entnehmen sind. Das Diagramm ist von unten nach oben zu lesen und stellt vereinfacht die aufeinander aufbauenden Schichten der Container-Welt dar. Zu den Kernkomponenten zählen die Schichten eins bis vier: Die (virtuelle) Infrastruktur (*Infrastructure*), das Betriebssystem (*Operating System*) und die *Container Runtime*. Diese Basis ist für die Bereitstellung von Containern auf einem einzelnen Host notwendig. Werden mehrere Hosts zu einem Cluster-Verbund

zusammengeschaltet, dann kommen die Schichten fünf (*Scheduling*) und sechs (*Orchestration*) noch dazu. Das *Scheduling* (*Dispatcher*) kümmert sich als untergeordneter Teil der Orchestrierung um das Service- und Ressourcen-Management in einem Cluster. Bspw. sorgt ein *Dispatcher* dafür, dass eine ausgerollte Ressource (Container) am richtigen physikalischen bzw. virtuellen Platz im Knoten-Verbund positioniert wird. Für eine erfolgreiche Container-Orchestrierungslösung bedarf es des Zusammenwirkens all dieser Schichten. [Lie19, vgl. S. 79f., 507ff]

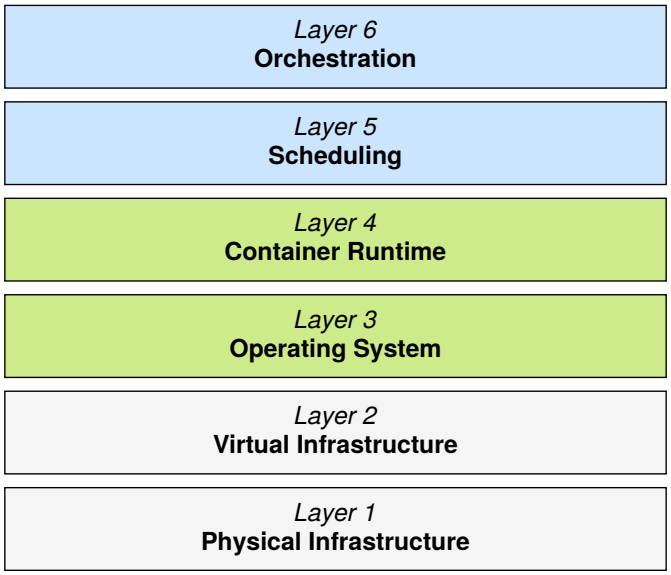


Abbildung 2.6: (Vereinfachte) Schichten der Container-Welt [Lie19, vgl. S. 79, 507]

Wesentliche Begrifflichkeiten sind noch einmal nachstehend definiert, die von den Werkzeugen der Container-Orchestrierung vereint bzw. synonym verwendet werden [Mou16, vgl. S. 251]:

- *Clustering* –
  „Gruppieren von 'Hosts' – entweder VMs oder Bare Metal – und deren Vernetzung. Ein Cluster sollte sich wie eine einzige Ressource anfühlen und nicht wie eine Gruppe ungleicher Maschinen“ [Mou16, Übers. S. 251].

- *Orchestrierung* –
  „Damit alle Teile zusammen funktionieren. Container auf geeigneten Hosts starten und miteinander verbinden. Ein Orchestrierungssystem kann auch Unterstützung für Skalierung, automatische Ausfallsicherung und Knoten-Neuausrichtung beinhalten“ [Mou16, Übers. S. 251].

- *Management* –
  „Bereitstellung einer Einsicht in das System und Unterstützung bei verschiedenen Verwaltungsaufgaben“ [Mou16, Übers. S. 251].

Letztendlich wird durch den Einsatz der Container-Orchestrierung Entwicklern die Möglichkeit gegeben, sich auf ihre Kernkompetenz, nämlich der Programmierung von Anwendungen, zu konzentrieren. Die Integration mit der Infrastruktur bleibt dabei vereinfacht, da auf eine konsistente Container-Umgebung aufgebaut werden kann. Des Weiteren erzielt die automatisierte Orchestrierung eine bessere Ressourcennutzung. Einzelne Container von Anwendungen werden irgendwo im Cluster auf Knoten bereitgestellt und können zu jeder Zeit verlegt werden. Dies wäre mit manuell geplanten Anwendungen nicht effizient zu realisieren. [Luk18, vgl. S. 20f.]

2.3.2 Choreographie

Im Gegensatz zur Orchestrierung besitzt die Choreographie keinen zentralen Akteur, welcher die Korrektheit sicherstellt und die Aufgabenerfüllung jeder einzelnen Komponente steuert. Es wird lediglich die Kommunikation zwischen den Bestandteilen bestimmt und beschrieben. Ein durch diese Kompositionsart erschaffener Gesamtservice ergibt sich aus einer Vielzahl von *End-to-End*-Verbindungen (*E2E*) unter den separaten Komponenten. Der Fokus dieser Dienstkomposition liegt deshalb auf dem Nachrichtenaustausch der einzelnen Bestandteile. [SBBK15, vgl. S. 122]

Im Endeffekt kann die Choreographie in *Microservice*-Architekturen verwendet werden, um eine Gesamtanwendung aus einzelnen Bestandteilen zu erschaffen. Jeder *Microservice* muss lediglich seine eigene Aufgabe in der Komposition wissen und über zuvor definierte Nachrichten kommunizieren können.

2.4 *DevOps*

Das Vorgehen in der Anwendungsentwicklung und die Bereitstellung von Software in der Produktion haben sich im Laufe der Zeit verändert [Luk18, vgl. S. 8f.], [Wil15]. Früher erstellten Entwicklungsteams (*Dev*) Anwendungen und übergaben das fertige Software-Produkt dem Betriebsteam (*Ops*), die wiederum die Anwendung in der Produktivumgebung bereitstellten und am Laufen hielten. Die Organisationen stellten allerdings fest, dass die fortlaufende Einbindung von Entwicklern, die am Entstehungsprozess des jeweiligen Produktes beteiligt waren, in diesem Zusammenhang Vorteile bot. Dieses Vorgehen wird auch als *DevOps* bezeichnet und beschreibt im Wesentlichen die Zusammenarbeit von Entwicklungs- und Betriebsteams im gesamten Lebenszyklus einer Software-Anwendung.

Vorteile für den Entwickler durch den stärkeren Bezug zur Produktivumgebung seiner Anwendung sind u. a. ein besseres Verständnis von Problemen der Benutzer sowie Schwierigkeiten der Systemadministratoren bei Wartungsarbeiten. Durch diese Wahrnehmung sind Entwickler auch eher geneigt, ihre Anwendungen häufiger bereitzustellen und mit Hilfe der kontinuierlichen Rückmeldungen der Anwender den Entwicklungsprozess zu steuern. Grundvoraussetzung dieses Vorgehens ist die Überarbeitung des klassischen Bereitstellungsprozesses, um des Öfteren neue Versionen von Anwendungen veröffentlichen zu können. Ideal wäre das Szenario, indem Entwickler ohne die aktive Hilfe von Systemadministratoren ihre Software selbst bereitstellen können. Gewöhnlich werden jedoch Kenntnisse über die zugrunde liegende Infrastruktur benötigt.

Obwohl Entwicklungsteams und Betriebsteams im Grunde genommen auf dasselbe Ziel – nämlich der Bereitstellung einer Anwendung für den Kunden – hinarbeiten, verfolgen sie dabei individuelle Interessen basierend auf einer unterschiedlichen Motivation. Ein Entwickler möchte vor allem neue Funktionen erstellen und das Benutzererlebnis verbessern, ohne sich dabei um Themen wie bspw. der Betriebssystemsicherheit kümmern zu müssen. Betriebsteams hingegen sind für die Produktionsbereitstellung und die elementare Hardware-Infrastruktur verantwortlich. Sie kümmern sich neben anderen Aspekten vor allem um die Sicherheit und Nutzung dieser Systeme. Dabei entscheidend ist, dass sich ein Systemadministrator nicht unbedingt mit den Abhängigkeiten zwischen den Softwarekomponenten beschäftigen möchte, es aber im Zweifelsfall tun muss. Idealerweise sollten Entwickler (*Dev*) daher in der Lage sein, selbstverantwortlich Anwendungen in den dafür vorgesehenen Umgebungen bereitzustellen, ohne auf die Hilfe des Betriebsteams (*Ops*) angewiesen zu sein. Diese konkrete Vorgehensweise wird auch als *NoOps* bezeichnet. Selbstverständlich werden weiterhin Administratoren für die Infrastruktur benötigt, allerdings sollten diese sich nicht mehr mit den darauf laufenden Anwendungen beschäftigen müssen.

Das *DevOps*-Prinzip wird u. a. durch nachstehende Verfahren unterstützt bzw. „gelebt“ [Lie19, vgl. S. 62f.], [AD19, vgl. S. 3ff]:

- *Continuous Integration* (CI) –
  Unter CI wird ein (semi-)automatisiertes Verfahren verstanden, welches sich um vollautomatisierte Tests einer Software während und nach ihrem *Build*-Prozess kümmert. Dies ermöglicht eine schnelle und teil-automatisierte Kontrolle (*Feedback*) bei Code-Änderungen. Das Verfahren besteht dabei aus einzelnen Schritten, die aufeinander aufbauen; es wird daher auch von einer Pipeline(-Verarbeitung) gesprochen. Beispiel: Ein Entwickler modifiziert Code im lokalen *Git-Repository* und lädt diesen hoch. Daraufhin findet eine automatisierte Code-Überprüfung auf Korrektheit statt. Falls ein Fehler auftritt, wird der Entwickler benachrichtigt. Er kann anschließend eine neue Version hochladen, die wieder getestet wird. Verlaufen nun alle Tests positiv, wird z. B. ein *Build*-Prozess angestoßen. Wird dieser ebenfalls positiv durchgeführt, dann werden die erschaffenen Artefakte für weitere Überprüfungen und der abschließenden Bereitstellung der CD-Pipeline übergeben.

- *Continuous Delivery* (CD) –
  CD erweitert im Wesentlichen das CI-Verfahren im Bereich der *Commit*- und *Build*-Prozesse. Es fügt der Pipeline Post-*Deployment*-Tests hinzu und realisiert zusätzlich die Bereitstellung des validierten Software-Codes in einer Produktivumgebung. Die Verarbeitung wird dabei wie schon beim CI möglichst automatisiert durchgeführt und in mehrere kleine Schritte unterteilt.

- *Infrastructure as Code* (*IaC*) –
  „Infrastruktur als Code ist der Ansatz, Rechen- und Netzwerkinfrastruktur durch Quellcode zu definieren, der dann wie jedes Softwaresystem behandelt werden kann“ [Fow16, Übers.]. Diese Idee unterstützt durch eine einfachere Zusammenarbeit zwischen den Entwicklern und dem Betriebsteam den *DevOps*-Gedanken, indem statt physische Hardware miteinander zu verkabeln, virtuelle Hardware als Software-Komponenten provisioniert wird. Bezogen auf die angesprochene Pipeline-Verarbeitung beim CI / CD ermöglicht dies eine noch bessere Überprüfbarkeit und Reproduzierbarkeit von *Builds*. Vor allem durch *Cloud Computing*-Plattformen hat sich dieses Vorgehen etabliert. [Fow16]

2.5 *Cloud Native*

Im Kontext moderner Anwendungen und Services, die auf *Cloud Computing*-Infrastrukturen basieren, die Container-Virtualisierung und -Orchestrierung verwenden sowie häufig auf *Open Source*-Software aufbauen, fällt oft der Begriff „*Cloud Native*“. Dieser Ausdruck hat unmittelbar etwas mit der im Jahr 2015 gegründeten *Cloud Native Computing Foundation* (CNCF) zu tun. Die Stiftung ist Teil der *Linux Foundation*, „baut nachhaltige Ökosysteme auf und fördert Gemeinschaften, um das Wachstum und die Erhaltung von Cloud Native Open Source-Software zu unterstützen“ [CNC20a, Übers.]. Die CNCF bringt Entwickler und Endanwender mit Herstellern zusammen, u. a. auch mit den großen öffentlichen *Cloud*-Anbietern. Eines der bekanntesten Projekte ist *Kubernetes*, welches zu den zentralen Komponenten des *Cloud Native*-Ökosystems zählt. [AD19, vgl. S. 16f.]

Von *Cloud Native*-Anwendungen, die hauptsächlich in *Cloud*-Umgebungen bereitgestellt werden, werden grundsätzlich folgende Eigenschaften erwartet [AD19, vgl. S. 16ff]:

- Automatisierbar –
  Anwendungen müssen verbreitete Standards, Formate und Schnittstellen erfüllen, damit

sie von Maschinen automatisiert und nicht mehr durch einen Menschen bereitgestellt und verwaltet werden.

- Überall einsetzbar und flexibel –
  Container-Anwendungen sind von realen Ressourcen wie bspw. Festplatten und spezifischen Rechnerumgebungen entkoppelt. Dadurch lassen sich die Container leicht zwischen Host-Knoten oder gesamten Clustern flexible verschieben.

- Widerstandsfähig und skalierbar –
  Cloud Native-Anwendungen können durch Möglichkeiten der Redundanz und der „*Graceful Degradation*“ – eingeschränkte Funktionalität aufgrund einer Beeinträchtigung – hochverfügbar gemacht werden, da sie im Grundansatz schon verteilt vorliegen. Klassische Anwendungen hingegen neigen eher zu einzelnen Schwachstellen (*Single Points of Failure*): Wenn der Hauptprozess abstürzt, ein Hardwaredefekt vorliegt oder eine Netzwerkkomponente überlastet ist, wird die Anwendung nicht mehr ausgeführt.

- Dynamisch –
  Orchestrator können Container so einsetzen, dass die zur Verfügung stehenden Ressourcen optimal ausgenutzt werden. Um zusätzlich eine Hochverfügbarkeit zu gewähren, können mehrere Instanzen erstellt und in Clustern platziert werden. Des Weiteren können Aktualisierungen von Services nahezu unbemerkt durch sogenannte „*Rolling Updates*“ durchgeführt werden. Bei ausreichender Redundanz / Dynamik wird der Netzwerkverkehr dabei nicht beeinflusst.

- *Observability* –
  Das *Debugging* bzw. Untersuchen von *Cloud Native*-Anwendungen ist aufgrund der Architektur oftmals schwerfällig zu realisieren. Die Beobachtbarkeit (*Observability*) ist daher eine zentrale Anforderung von verteilten Systemen. Um einzuschätzen, warum sich die bereitgestellten Systeme falsch verhalten, werden Verfahren wie das *Monitoring*, *Logging*, *Tracing* und erstellte Metriken herangezogen.

- Verteilt –
  Mit Hilfe von *Cloud Native* werden Anwendungen gebaut und ausgeführt, die die Vorzüge der verteilten und dezentralen *Cloud*-Infrastruktur verwenden. Grundlegend stellt sich nicht die Frage, wo die Anwendung läuft, sondern wie der bereitzustellende Service funktioniert. *Cloud Native*-Anwendungen bestehen vorwiegend aus vielen miteinander vernetzten und verteilten *Microservices*. Seltener wird der Software-Code als einzelne Entität in Form von Monolithen bereitgestellt. Ein *Microservice* ist im Allgemeinen ein abgeschlossener Dienst, der eine Aufgabe erledigt. Wenn demzufolge mehrere *Microservices* zusammengebracht werden, entsteht eine (*Cloud Native*-)Anwendung.

Schlussendlich basieren gut entworfene Anwendungen, die dem *Cloud Native*-Paradigma folgen, aus *Microservices* und verwenden *Cloud Computing*-Mechanismen. Die *Microservices* werden dabei durch die Container-Virtualisierung und -Orchestrierung realisiert. Herausfordernd sind jedoch die Architekturentscheidungen, was die einzelnen Komponenten sein sollen, wo sich die Grenzen befinden und wie untereinander kommuniziert wird. Die Gesamtsysteme liegen verteilt vor, was zu einer enthaltenen Komplexität führen kann, da sich die *Microservices* im Verbund schlecht beobachten und inspizieren lassen sowie unvorhergesehene Fehler auftreten können. [AD19, vgl. S. 16ff]

2.6 Zusammenfassung

Wie in diesem Kapitel ersichtlich wurde, bieten *Microservices* in der Software-Architektur durch ihre Entkopplung einige Vorteile gegenüber Monolithen. Sie können u. a. einzeln entwickelt, bereitgestellt, aktualisiert und skaliert werden. Die einzelnen Bestandteile können somit schnell und sooft wie notwendig angepasst werden, um den dynamischen Geschäftsprozessen entgegenzuwirken. Allerdings kann der *Microservice*-Architekturstil, wie auch jeder andere Ansatz, Nachteile mit sich bringen [Fow15a]. Zu nennen sind hier die Schwierigkeiten verteilter Systeme, mögliche Inkonsistenzen und die zunehmende Betriebskomplexität aufgrund der vielen separaten Bestandteile. Dennoch überwiegen die Vorteile und der Trend von Monolithen zu *Microservices* bleibt bestehen. In der Software-Bereitstellung wird dieser Architekturstil mit Hilfe der Container-Virtualisierung weitergeführt [CNC20b]. Container stellen isolierte Umgebungen auf einem Host für einen *Microservice* mit unterschiedlichen Anforderungen bereit. Die darunterliegende Plattform – *On-Premise*- oder *Cloud*-Umgebungen – spielt für die Realisierung keine allzu entscheidende Rolle, solange hier der Funktionsumfang des *CaaS* gegeben ist. Vielmehr sind hier die betriebswirtschaftlichen und datenschutzrechtlichen Aspekte relevant. Nimmt die Anzahl der bereitzustellenden *Microservice*-Container in einem großen Umfang zu und werden mehrere Hosts eingesetzt, muss das manuelle Verwalten zwingend durch eine Automatisierung abgelöst werden. Die Container-Orchestrierung als Dienstkomposition kann in skalierbaren Container-Clustern hierfür Abhilfe schaffen. Die beiden Konzepte *DevOps* und *Cloud Native* unterstützen dabei die aufgezeigte Entwicklung in der IT-Landschaft. Die maximale Automation bspw. durch CI / CD, das Zusammenwachsen von Entwicklungs- und Betriebsteams und die Möglichkeit der horizontalen Skalierbarkeit auf Anforderung sind nur einige Aspekte, die auch zukünftig in diesem Umfeld immer wichtiger werden.

3 Komponenten der Infrastruktur

In diesem Kapitel werden die wichtigsten Komponenten im Kontext des angedachten Szenarios dieser Ausarbeitung vorgestellt. Zu ihnen zählen im Speziellen die Infrastruktur-Bestandteile wie die im Abschnitt 3.1 thematisierte und vorhandene *VMware*-Virtualisierungsumgebung, die angestrebte Container-Orchestrierungslösung *Kubernetes* (*K8s*) im Abschnitt 3.5 sowie die im Abschnitt 3.6 darauf aufbauenden *Rancher Labs*-Management-Tools. Zuvor werden noch im Abschnitt 3.2 *Ubuntu* als Betriebssystem, die *Docker*-Container-Virtualisierung im Abschnitt 3.3 und die noch fehlende Container-*Image Registry Harbor* im Abschnitt 3.4 behandelt. Abschließend wird im Abschnitt 3.7 die bereits zur Verfügung stehende *DevOps*-Plattform *GitLab* betrachtet.

3.1 *VMware*

Das 1998 gegründete US-amerikanische Unternehmen *VMware* ist eines der führenden Software-Anbieter im Bereich der (Server-)Virtualisierung und des *Cloud Computings*. In den Anfängen bestand das Unternehmensziel, physische Computer mit Hilfe einer noch nicht vorhandenen Technologie zu abstrahieren und somit die Charakteristiken eines Computersystems virtuell abbilden zu können. Die so entstandenen virtuellen Maschinen (VMs) – wie im Unterabschnitt 2.2.2 bereits betrachtet – ermöglichen die Konsolidierung von vielen VMs auf einem physischen Host, welches eine bessere Ausnutzung der zugrunde liegenden Hardware gewährt. Im Betrieb eines Rechenzentrums ist vorrangig die Server-Virtualisierung mit dem effizienteren *Hypervisor Typ 1* verbreitet. Im Verlauf der Zeit hat *VMware* als börsennotiertes Unternehmen sein Produktportfolio erweitert und bietet verstärkt im Bereich der *Cloud*-Infrastrukturen und -Management-Plattformen Lösungskonzepte an. Nach den Service-Modellen *IaaS*, *PaaS* und *SaaS* des *Cloud Computings* sollen den Nutzern daraufhin Anwendungsdienste angeboten werden. Neben der kommerziellen Software ist *VMware* auch in der *Open Source*-Gemeinschaft aktiv und unterstützt diverse Projekte (s. *Harbor* im Abschnitt 3.4), die insbesondere im *Cloud Native*-Umfeld einzuordnen sind. [VMw20d], [WAG+18, vgl. S. 41ff]

Nachstehend werden im Unterabschnitt 3.1.1 die *VMware*-Virtualisierungsumgebung *vSphere* mit einigen zugehörigen Produkten / Funktionen und im Unterabschnitt 3.1.2 noch einmal im Besonderen einige Komponenten der *vRealize*-Produktlinie zur Automatisierung sowie als einheitliche *Cloud*-Managementlösung für Multi-*Cloud*-Umgebungen behandelt.

3.1.1 *vSphere*

VMwares Produktlinie für die Rechenzentrums- und Server-Virtualisierung nennt sich *vSphere*. In ihr sind bzw. können eine Vielzahl von weiteren *VMware*-Produkten verankert sein. Darüber hinaus kann *vSphere* durch Produkte von Drittanbietern erweitert werden. Wie in der Abbildung 3.1 zu sehen ist, kommen als Basis mehrere Instanzen des *Hypervisor Typ 1 ESXi* zum Einsatz, der die Server- und Desktop-Virtualisierung ermöglicht. *VMware ESXi* wird als Betriebssystem auf einem physischen Server (*vSphere*-Host) installiert und kann sowohl physische als auch virtuelle Ressourcen in der Virtualisierungsumgebung verwalten. Eine weitere zentrale Komponente ist

der *vCenter Server*, der als Hauptanlaufstelle in der *vSphere*-Umgebung agiert. Hierüber erfolgt die Verwaltung und Konfiguration der gesamten Ressourcen (*Datacenters*, Cluster, Netzwerke, *Datastores*, VMs, CPUs etc.) und vieler Zusatzprodukte. Für die grafische Administration gibt es den hierauf aufbauenden *vSphere Client*. Neben den vielen lizenzpflichtigen *vSphere*-Varianten, stellt *VMware* auch den kostenfreien *ESXi-Hypervisor* mit dem Produktnamen *vSphere Hypervisor* zur Verfügung, der allerdings etliche Einschränkungen durch Hardware-Limitationen und keiner Möglichkeit eines zentralen Managements mit Cluster-Betrieb aufweist. [WAG+18, vgl. S. 42ff]

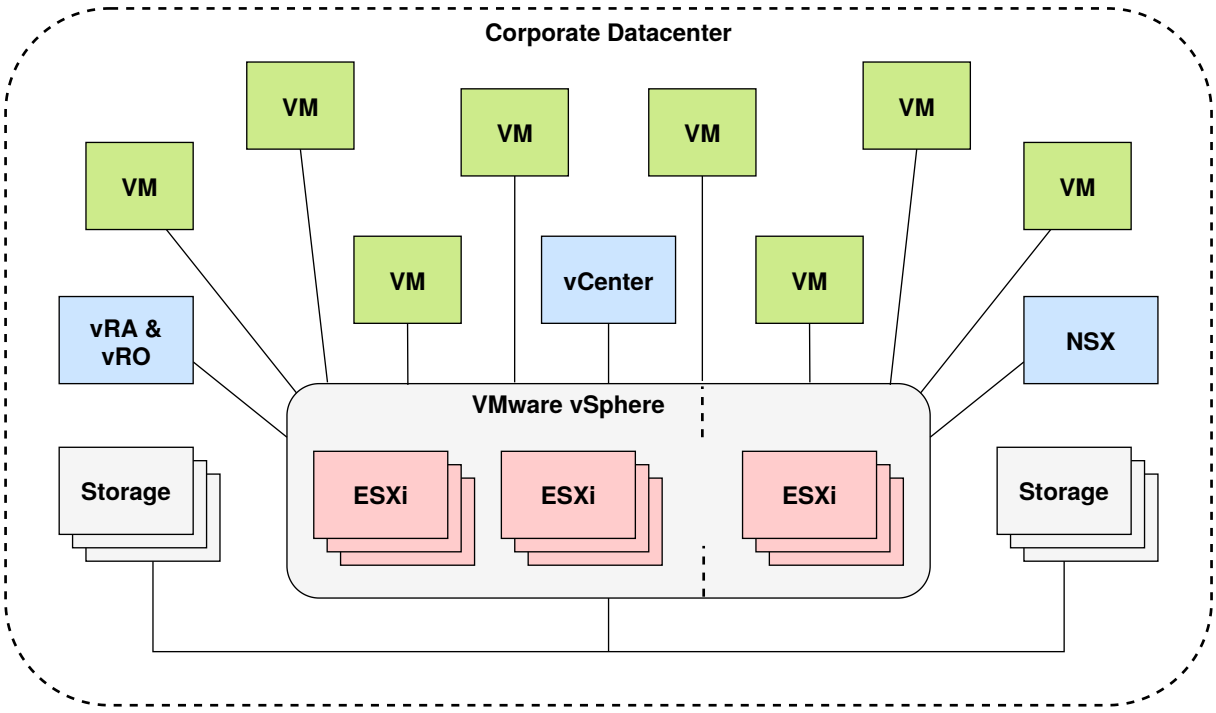


Abbildung 3.1: Aufbau einer *VMware vSphere*-Virtualisierungsumgebung [WAG+18, vgl. S. 53ff, 167ff]

Langfristig verfolgt *VMware* mit *vSphere* das Ziel ein sogenanntes *Software-Defined Datacenter* (SDDC) aufzubauen und zu etablieren. Hiermit ist die nahezu vollständige Ressourcen-Virtualisierung und Automatisierung einer (*On-Premise-*)Infrastruktur ähnlich wie beim *Cloud Computing* mit der *XaaS*-Philosophie angedacht. Die wichtigsten architektonischen Kernkomponenten, aus denen ein SDDC besteht, sind die folgenden [WAG+18, vgl. S. 53ff, 1235]:

- *Compute Virtualization* –
  Hierunter sind die *vSphere*-Komponenten für die (Server-)Virtualisierung von Betriebssystemen zu verstehen. Zumeist werden mehrere physische *vSphere*-Hosts, die ihre Ressourcen (CPU, RAM, Netzwerkkarte, Festplattenspeicher etc.) über die Virtualisierungsschicht des *ESXi-Hypervisors* den VMs bereitstellen, in einem Cluster-Verbund zusammengeschaltet. [WAG+18, vgl. S. 43f., 54ff]

- *Software-Defined Networking* (SDN), *Network Virtualization* –
  SDN umfasst Komponenten für die Software-basierte Virtualisierung eines zugrunde liegenden Netzwerks. Das von *VMware* bereitgestellte Produkt NSX (*Network and Security Virtualization*) erweitert *vSphere* hierfür um die wesentlichen Funktionen wie logisches *Switching*, *Routing* und der Netzwerksicherheit auf den OSI-Schichten (*Open Systems Interconnection*) zwei bis sieben. Wie schon unter der bereits definierten Abkürzung *IaC*,

wird auch hiermit die Trennung und Abstraktion von Netzwerkfunktionalitäten und ihrer Bereitstellung in Software verstanden. [WAG+18, vgl. S. 477ff]

- *Software-Defined Storage* (SDS), *Storage Virtualization* –
  Unter SDS sind Komponenten für die Speichervirtualisierung zu verstehen. Auch hier wird wieder eine Abstraktion der (Speicher-)Ressourcen von der basierenden Hardware-Technologie, u. a. durch das Zusammenfassen von verbauten Festplatten (HDDs / *Hard Disk Drives*, SSDs / *Solid-State Drives*) zu *Shared Storage Volumes*, realisiert. Produktlösungen im Bereich *Storage* bieten sowohl *VMware* mit *vSAN* oder auch Dritthersteller wie bspw. *NetApp* [Net20] an. [WAG+18, vgl. S. 666ff, 691ff]

- *Management* –
  Von essentieller Bedeutung sind Komponenten für das Management der virtuellen Infrastruktur, damit es einem Administrator möglich ist, alle Software-definierten Ressourcen bereitzustellen, zu kontrollieren und letztendlich zu verwalten. Wichtig in diesem Umfeld sind u. a. der *vCenter Server*, der *Platform Services Controller* (PSC) und der *Update Manager*. Als grafisches Management-Tool wird der *VMware vSphere Client* eingesetzt, welcher bald nur noch auf HTML5 (*Hypertext Markup Language*) basieren wird. [WAG+18, vgl. S. 44f., 49, 72ff, 368ff, 377ff, 827ff]

- *Automation* –
  Zur erweiterten Betrachtung eines SDDCs gehören zudem Komponenten für die Automatisierung. Hier sind vor allem die *vRealize*-Produktlinie mit *vRealize Automation* (*vRA*) und dem *vRealize Orchestrator* (*vRO*) zu nennen, die es ermöglichen, Ressourcen eines Rechenzentrums weitestgehend automatisiert bereitzustellen. Im Speziellen wird hierzu noch einmal im Unterabschnitt 3.1.2 eingegangen.

Ein paar entscheidende Detailfunktionen der *vSphere*-Umgebung sind noch nicht angesprochen worden. Wie in der Abbildung 3.1 dargestellt wird, sind solche Virtualisierungsumgebungen in einem Cluster-Betrieb mit mehreren *vSphere*-Hosts organisiert, auf denen der *ESXi-Hypervisor* installiert ist [WAG+18, vgl. S. 54ff, 167ff]. Die gestrichelten Linien zwischen den *ESXis* symbolisieren die notwendige örtliche Trennung einiger Hosts zur Gewährleistung der Hochverfügbarkeit. Auch die *Storage*-Anbindung erfolgt dieser Anforderung in einem sogar eigens dafür vorgesehenen Netzwerk. An dieser Stelle werden nun einige der wichtigsten Eigenschaften eines solchen Clusters erläutert: Durch *Fault Tolerance* (FT) kann eine laufende VM in Echtzeit auf einen anderen Host gespiegelt werden [WAG+18, vgl. S. 46, 1049ff]. Falls der Host mit der aktiven Instanz der virtuellen Maschine ausfallen sollte, übernimmt dann die gespiegelte Instanz unterbrechungsfrei den Betrieb. *High Availability* (HA) hingegen ermöglicht es, Ausfälle von ganzen Hosts und VMs zu erkennen und darauf automatisiert zu reagieren [WAG+18, vgl. S. 46, 171ff]. Beendete Instanzen können bei Ausfall auf einem anderen Host im HA-Cluster wieder gestartet werden. Damit virtuelle Maschinen während der Laufzeit ohne Unterbrechungen zwischen Hosts verschoben oder Datenspeicher (*Datastores*) gewechselt werden können, werden die *VMware*-Funktionen *vMotion* und *Storage vMotion* benötigt [WAG+18, vgl. S. 46, 97ff]. Der Anwender bekommt in der Regel nichts von diesen Maßnahmen mit und er kann die VM in der gesamten Zeit ohne Einschränkungen nutzen. Eine weitere Komponente ist der *Distributed Resource Scheduler* (DRS), der die Ressourcenauslastung der Hosts in einem Cluster überwacht und Vorschläge für eine optimale Ressourcenverteilung erstellt [WAG+18, vgl. S. 46, 206ff]. Diese Optimierungen kann der *Scheduler* währenddessen auch vollautomatisiert durchführen und ermöglicht somit eine automatische Lastverteilung. Eine letzte hier zu erwähnende Eigenschaft ist das Erstellen von *Snapshots* bezogen auf eine VM [WAG+18, vgl. S. 1178ff]. Sie halten den Zustand einer virtuellen Maschine mit allen dazugehörigen Abhängigkeiten fest und erlauben eine spätere Rückkehr in diesen vorherigen Zustand.

3.1.2 *vRealize Suite*

Mit der *vRealize Suite* bietet *VMware* mehrere Werkzeuge an, die in der Kombination ein *Cloud*-übergreifendes Management ermöglichen [VMw20b]. Ein Bestandteil davon ist *vRealize Automation* (*vRA*), was eine kommerzielle *Hybrid Cloud*-Automatisierungsplattform ist, über die IT-Dienstleistungen einheitlich und konsistent in *VMware*-Virtualisierungsumgebungen orchestriert werden können. Ein Service-Katalog ermöglicht es Endanwendern Infrastruktur- und Anwendungsressourcen über ein *DevOps*-fähiges *Self-Service Portal* anzufordern. Die Einhaltung der Unternehmensrichtlinien wird durch die Bereitstellung von *On-Demand*-Diensten über die zentrale Plattform sichergestellt. Auch eine zugrunde liegende heterogene Infrastruktur spielt keine all zu große Rolle, da ein gemeinsamer Servicekatalog die Anforderungen an alle beteiligten Ressourcen in den Geschäftsprozessen überschaubar hält. Als weiteres Werkzeug zum Automatisieren von IT-Umgebungen über sogenannte *Workflows* steht der *vRealize Orchestrator* (*vRO*) zur Verfügung [WAG+18, vgl. 49, 1086f.]. Die Orchestrierungs-*Engine* liefert bereits eine Menge automatisierbarer *Workflows* von Haus aus mit, die ansonsten manuell mit Hilfe des *vSphere Clients* vorgenommen werden müssten.

In dieser *VMware*-basierten Umgebung gibt es bei der Fehleranalyse Unterstützung von den Produkten *vRealize Operations* und *vRealize Log Insight* [WAG+18, vgl. S. 53]. Die bereitzustellenden, virtuellen Ressourcen können somit über eine integrierte Protokollierung u. a. bzgl. der Kapazitäten und Ausfälle überwacht werden. In diesem Zusammenhang ist auch das Drittanbieter-Werkzeug *Opvisor Performance Analyzer* (OPA) zu nennen [WAG+18, vgl. S. 945], welches eine detaillierte Analyse von Performance-Problemen und Engpässen in *vSphere*-Umgebungen und deren Peripherie (*NetApp*, *Cisco*, *Microsoft*, *Linux*, *Docker* etc.) ermöglicht. Der OPA optimiert letztendlich die Nutzung der Ressourcen und minimiert Leistungsprobleme durch die Bereitstellung von *Dashboards* und Standardmetriken.

3.2 *Ubuntu*

Das *Open Source*-Betriebssystem *Ubuntu* ist eine *Linux*-Distribution, welche auf *Debian* basiert. Sie wird vom Software-Hersteller *Canonical* und anderen unabhängigen Entwicklern für jeden zur freien Verwendung bereitgestellt. Im Grundansatz wird die fortlaufende Weiterentwicklung eines einfach zu installierenden und leicht zu bedienenden Betriebssystem für sämtliche Plattformen mit einer aufeinander abgestimmten Software verfolgt. Mit der ersten *Ubuntu*-Veröffentlichung im Oktober 2004 wurde ein weltweites Interesse für diese Software ausgelöst und mittlerweile gehört sie zu den verbreitetsten *Linux*-Betriebssystemen im Desktop- und Server-Umfeld. Neue Versionen erscheinen jedes halbe Jahr im April („*04er*“-Version) und im Oktober („*10er*“-Version). Alle zwei Jahre wird zudem eine Betriebssystemversion mit Langzeitunterstützung (LTS, *Long Term Support*) herausgebracht. [Ubu20a]

Von *Canonical* werden für den *Cloud Native*-Bereich die speziell angepassten *Ubuntu Cloud Images* [Ubu20b] ausgeliefert. Dies sind vorinstallierte Betriebssystemvorlagen, die für *Cloud Computing*-Infrastrukturen extra angepasst sind. Als integriertes Werkzeug steht u. a. das eigens entwickelte und frei zur Verfügung stehende *Cloud-init* bereit, mit dem plattformübergreifend *Cloud*-Instanzen initialisiert werden können. Diese Distributionsmethode wird von nahezu allen großen *Cloud*-Anbietern, privaten *Cloud*-Umgebungen und *Bare Metal*-Installationen unterstützt. Eine Cloud-Instanz wird dabei aus einem zugrunde liegenden *Disk Image* und den folgenden Instanzdaten initialisiert: *Cloud*-Metadaten, *User Data* und *Vendor Data*. Im ersten Schritt wird die Instanz bezogen auf die Metadaten der Bereitstellungsumgebung (*Cloud* etc.) entsprechend angepasst. In den danach folgenden Schritten werden alle optionalen Benutzer- oder Anbieterdaten

(*User*, *Vendor*), die an die Instanz übergeben wurden, analysiert und verarbeitet. Die individuelle Bereitstellung von *Cloud*-Instanzen wird somit durch die Automatisierung deutlich vereinfacht. [Can20]

3.3 *Docker*

Eine konkrete Ausprägung der Container-Virtualisierung (s. Unterabschnitt 2.2.3) und Erweiterung der *Linux*-Container-Technologie ist *Docker*. Technologien in diesem Bereich gibt es schon seit längerer Zeit, allerdings gewann *Docker* durch die einfache Verwendbarkeit und Verfügbarkeit einen großen Bekanntheitsgrad. Die seit 2013 [Doc20b, v0.1.0, 2013-03-23] veröffentlichte *Open Source*-Plattform macht es möglich eine Anwendung mit zugehörigen Bibliotheken und weiteren Abhängigkeiten wie die eines vollständigen Dateisystems, in einem einfachen und übertragbaren Format zu kapseln. So lassen sich Applikationen mit ihrer gesamten Umgebung auf unterschiedlichen Systemen bereitstellen und ausführen, auf denen die *Docker Engine* vorhanden ist. [Doc20d, Doc20a], [Luk18, vgl. S. 14f.], [Lie19, vgl. S. 117ff]

Grundlegend können drei *Docker*-Bestandteile unterschieden werden [Luk18, vgl. S. 15], [Doc20d]:

- *Images* –
  Eine Anwendung mit der dazugehörigen Umgebung wird in einem Behältnis, dem sogenannten (Container-)*Image*, aufbewahrt. Enthalten sind sowohl ein Dateisystem, als auch zusätzliche Metadaten des Anwendungskontextes, wie bspw. der Pfad zur ausführbaren Datei, die nach der Bereitstellung des *Images* gestartet werden soll. *Images* sind in Schichten aufgebaut (s. Abbildung 3.3), die von mehreren *Images* wiederverwendet werden können.
- *Registries* –
  In *Registries* werden *Docker Images* gespeichert und anderen Parteien für einen erleichterten Austausch zugänglich gemacht – ähnlich wie ein *Repository* für Software-Quellcode. Wird ein *Image* erstellt, kann es lokal auf demselben Host ausgeführt, oder in eine *Registry* hochgeladen werden, um die Anwendung letztendlich auf einem anderen Host herunterzuladen und auszuführen. Einige *Registries* sind öffentlich zugänglich, wie die *Docker*-eigene *Registry Docker Hub* unter der URL (*Uniform Resource Locator*) „`https://hub.docker.com`“ [Doc20c], andere sind wiederum nur bestimmten Personen vorbehalten.
- *Container* –
  Als *Docker*-Container werden ausgeführte *Docker Images* bezeichnet. Ein aktiver *Docker*-Container besitzt Prozesse, die letztendlich auf einem Host mit Hilfe von *Docker* ausgeführt werden. Die Prozesse sind dabei im Ausführungskontext isoliert und laufen Ressourcenbeschränkt, was bedeutet, dass nur auf die Ressourcen (CPU, RAM etc.) des Hosts zugegriffen werden kann, die ihm zugeteilt wurden.

In der Abbildung 3.2 wird verdeutlicht, wie das Zusammenspiel der *Docker*-Bestandteile aus der oberen Auflistung aussieht. Ein Entwickler (*Dev*) erstellt mit Hilfe der *Docker CLI* (1., *Command-Line Interface*, dt. Kommandozeile) ein *Image* auf einem Entwicklungs-Host, welcher die *Docker Engine* ausführt (2., *docker build*). Anschließend gibt er die Anweisung das *Image* in eine *Registry* abzulegen (3., *docker push*). Das *Docker Image* ist nun für alle Parteien frei zugänglich, die einen Zugang zur *Registry* besitzen. Nun können berechtigte Personen (4.) – wie auch der Entwickler – das soeben hochgeladene *Image* mit der *Docker CLI* auf beliebige Hosts, bspw. auf einem Produktions-Host, wieder herunterladen (5., *docker pull*) und ausführen. Letztendlich erstellt *Docker* auf dem Host einen isolierten Container auf der Grundlage des *Images* (6., *docker run*) und führt eine Binärdatei aus, welche Bestandteil des ursprünglichen *Images* ist. [Luk18, vgl. S. 15f.], [Doc20d]

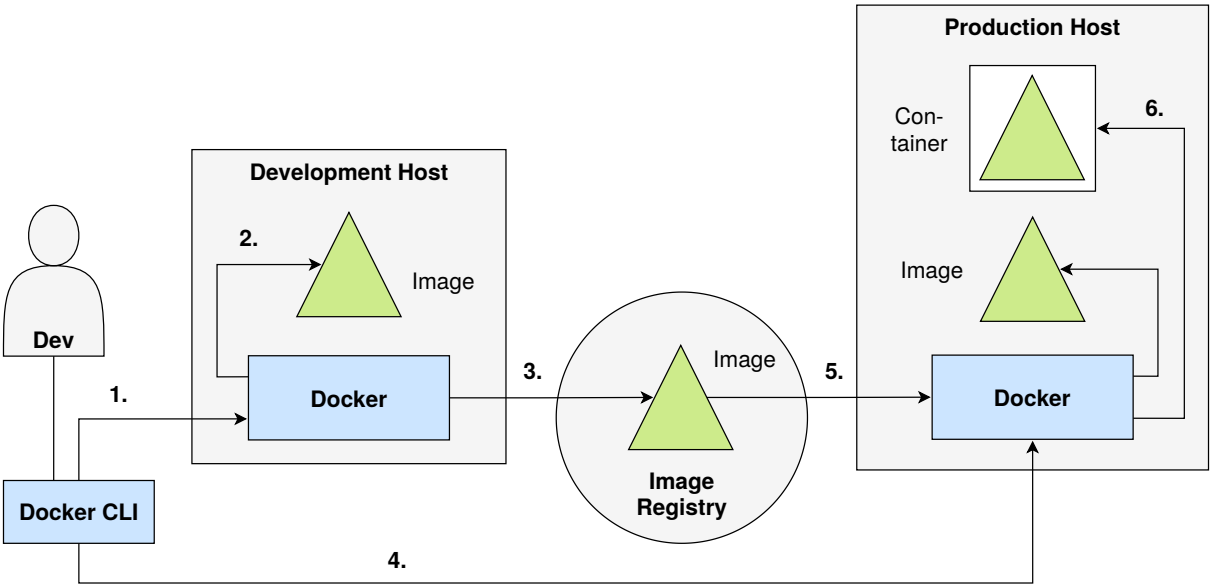


Abbildung 3.2: Bereitstellen von *Docker*-Containern [Luk18, vgl. S. 16], [Doc20d]

Container-*Images* sind, wie in der Abbildung 3.3 zu sehen ist, aus einem *Base Image* und weiteren Schichten (*Image Layers*) aufgebaut, die auch von mehreren *Images* gemeinsam genutzt und wiederverwendet werden können. Dies hat den Vorteil, dass nur einzelne Schichten über das Netzwerk heruntergeladen werden müssen, wenn die verbliebenen Schichten bereits zuvor durch die Ausführung eines anderen Containers abgerufen wurden. Neben der effizienten Verteilung von *Images*, sinkt durch die Verwendung von Schichten auch der Speicherbedarf; jede Schicht wird nur einmal gespeichert. Container mit denselben *Layern*, lesen zwar dieselben Dateien, Änderungen wirken sich allerdings durch den vorhandenen Schreibschutz (*Read-Only*) nur auf den Initiator aus. Bei der Ausführung eines Containers wird darum immer eine schreibbare Schicht (*Container Layer*) oberhalb der anderen Schichten hinzugefügt. Falls also ein Container-Prozess in eine Datei der unteren Schichten schreiben möchte, wird davon im *Container Layer* eine Kopie platziert und letztendlich die Modifikation ausgeführt. Damit die durchgeführten Änderungen auch nach dem Löschen eines Containers bestehen bleiben, müssen sie durch das Hinzufügen eines neuen *Layers* im *Image* eingecheckt werden. Neue Container-Instanzen, die von diesem *Image*-Stand starten, erhalten alle zuvor durchgeführten Anpassungen. [Luk18, vgl. S. 17f.], [Lie19, vgl. S. 114]

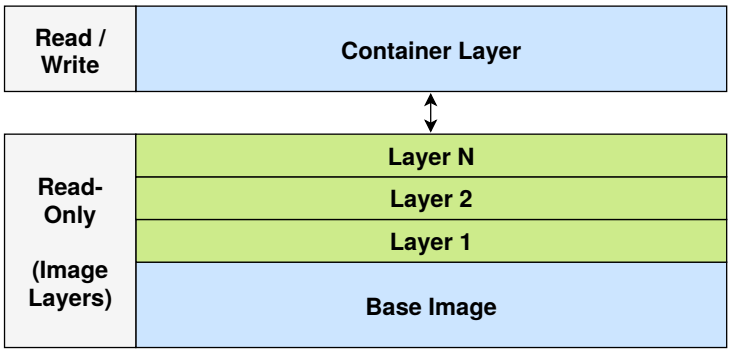


Abbildung 3.3: Schematischer Aufbau eines Container-*Images* [Lie19, vgl. S. 114]

Abschließend betrachtet ermöglicht *Docker* als konkrete Technologie der Container-Virtualisierung den *Microservice*-Architekturstil (s. Unterabschnitt 2.1.2). Die Umsetzung basiert dabei auf der Kapselung von Anwendungen mit ihren Abhängigkeiten in *Images* und zur Laufzeit in

Containern. Einschränkungen können in der Portierbarkeit von *Docker Images* bestehen, wenn eine Anwendung bestimmte Anforderungen an den Kernel oder die Hardware-Architektur stellt, die auf einem entsprechenden Host bspw. durch eine andere Version des *Linux*-Kernels oder fehlender Kernel-Module nicht erfüllt werden können [Luk18, vgl. S. 18].

3.4 *Harbor*

Harbor ist eine frei verfügbare Container-*Image Registry*, welche einen rollen-basierten Zugriff (RBAC, *Role-Based Access Control*) auf in *Repositories* organisierte OCI *Images* (*Open Container Initiative*) ermöglicht. Darüber hinaus können Schwachstellen einzelner *Image Layers* analysiert und vertrauenswürdige Abbilder signiert werden. Die *Open Source*-Anwendung bietet eine sichere und konsistente Verwaltung von *Images* für *Cloud Native*-Umgebungen wie z. B. *Kubernetes* und *Docker* an. Ursprünglich wurde *Harbor* von *VMware* entwickelt, inzwischen wird das Projekt allerdings bei der CNCF kollaborativ weitergeführt. Bezogen auf Anforderungen der Hochverfügbarkeit und des Lastausgleichs für Produktivumgebungen ist anzumerken, dass *Harbor* Konfigurationen besitzt, um die gesamte *Registry* zu replizieren. *Images* können bspw. von einer *Registry*-Instanz zu einer anderen automatisiert über Richtlinien synchronisiert werden; dafür muss es nicht unbedingt eine zweite *Harbor-Registry* sein. Auch die *Harbor*-Architektur einer Instanz lässt sich für HA auslegen. Die Authentifizierung von Benutzern und Gruppen kann entweder lokal in einer Datenbank oder über einen Verzeichnisdienst geschehen. Durch eine integrierte *Garbage Collection* können *Images* gelöscht und der Speicherplatz wieder freigegeben werden. Außerdem bietet *Harbor* ein grafisches Webportal an, welches eine übersichtliche Verwaltung von Projekten und *Repositories* ermöglicht. Über eine REST-API lassen sich die meisten Operationen in externe Systeme einbinden und über ein zu konfigurierendes Audit nachverfolgen. [Har20], [Lie19, vgl. S. 468ff]

Der Signierer *Notary* [Doc20e] kann als Werkzeug für „*Trusted Images*“ agieren, damit die *Image*-Authentizität verbessert und schließlich gewährleistet wird. Einzelne *Images* können hiermit digital signiert und in eine *Registry* hochgeladen werden. Nutzer können anschließend vor der Verwendung eines signierten *Images* die Integrität und Herkunft überprüfen. Das CLI-basierte *Notary* übernimmt die Schlüsselverwaltungs- und Signaturschnittstelle sowohl auf der Client als auch der Server-Seite. Bei Bedarf können in *Harbor* u. a. die *Open Source*-Projekte *Clair* [Qua20] oder *Trivy* [Aqu20b] als CVE-*Scanner* (*Common Vulnerabilities and Exposures*) transparent integriert werden. *Images* können hierüber regelmäßig statisch auf Schwachstellen analysiert und verantwortliche *Repository*-Inhaber daraufhin benachrichtigt werden. Die Prüfung umfasst u. a. die Erkennung von bekannter und unsicherer Software bzw. -Versionen.

3.5 *Kubernetes* (*K8s*)

Kubernetes – die Kurzbezeichnung lautet *K8s* – ist eine *Open Source*-basierte Plattform für die Container-Orchestrierung (s. Unterabschnitt 2.3.1). Initialisiert wurde das Projekt von *Google* und ist seit dem Jahr 2014 der Öffentlichkeit zugänglich; mittlerweile wird *K8s* bei der CNCF verwaltet. Die Plattform kann Abhilfe bei der Koordinierung von Containern in einem Cluster schaffen, indem sie in diesem Kontext die Bereitstellung, Verwaltung, automatische Skalierung, Hochverfügbarkeit und weitere Aufgaben übernimmt. Entwickelt wurde *Kubernetes* aufgrund der Tatsache, dass zu diesem Zeitpunkt keine in Frage kommende Lösung existierte, um containerisierte Arbeitslasten in einem großen Maßstab – wie sie u. a. bei *Google* anfallen

– zu administrieren. In *On-Premise*- und *Cloud*-Infrastrukturen ist *K8s* eine der populärsten Lösungen für die Container-Cluster-Orchestrierung. [Kub20m], [Lie19, vgl. S. 555f.]

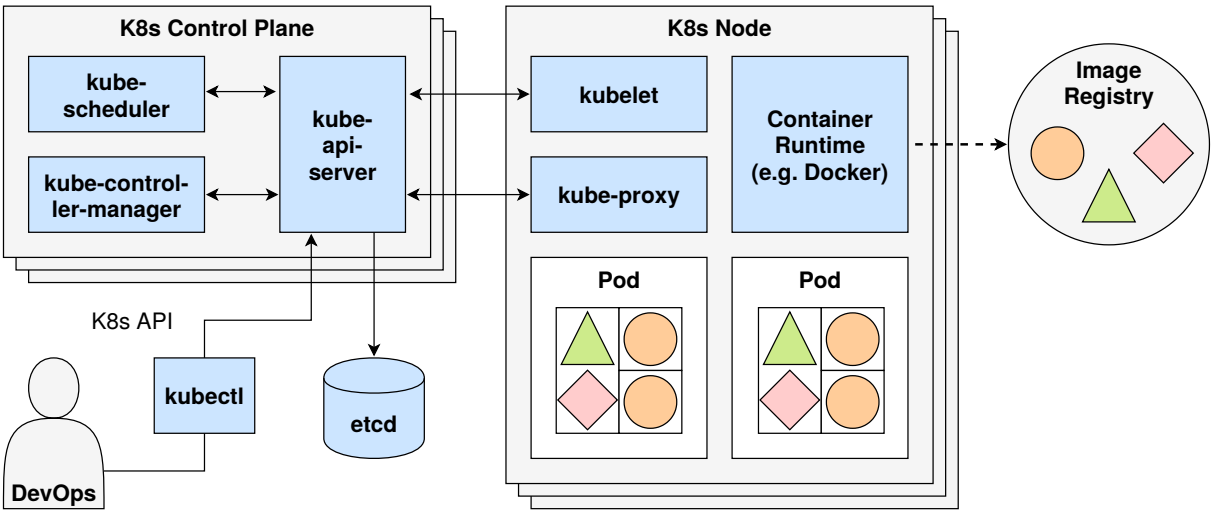


Abbildung 3.4: *Kubernetes*-Architektur mit *Control Plane* und *Nodes* [Kub20d, Kub20f]

In der Abbildung 3.4 wird ersichtlich, dass die *K8s*-Architektur in zwei verschiedene Arten von Hosts strukturiert ist. Es wird zwischen der *Control Plane* und den *Nodes* unterschieden [Kub20d]. Für die Bereitstellung von *Kubernetes* wird ein Cluster von mindestens einer *Control Plane* und einem *Node* benötigt; beide Bestandteile können für Entwicklungsumgebungen auch auf einem Host angesiedelt sein. Einfache *Nodes* stellen sogenannte *Pods* bereit, die einen einzelnen Container oder eine Gruppe von zusammengehörigen Containern einer Anwendung ausführen [Kub20h]. *Pods* bieten dabei den Vorteil mehrere Container in einem gemeinsamen Kontext zu abstrahieren, auf dem selben Knoten laufen und lokal kommunizieren zu lassen sowie ggf. gemeinsamen *Storage* zu verwenden. Ein *Pod* repräsentiert zudem die kleinste *K8s*-Bereitstellungseinheit. Eine *Control Plane* nimmt hingegen die Verwaltungsaufgabe ein und koordiniert auszuführende *Pods* an die *Nodes* im Cluster. Es können jeweils mehrere Instanzen eingesetzt werden, um die Bedingungen der Hochverfügbarkeit (HA) und Ausfallsicherheit (FT) zu erfüllen [Kub20f].

Darüber hinaus besteht eine *Control Plane* aus weiteren Bestandteilen, die zusammen die Steuerungsebene eines Clusters ergeben [Kub20d]. Sie treffen globale Entscheidungen, erkennen auftretende Ereignisse und reagieren entsprechend mit Maßnahmen. Der *kube-api-server* stellt dabei die *Kubernetes*-API zur Verfügung, auf die mit der *kubectl* CLI (s. Auflistung A.4) zugegriffen werden kann. Außerdem wird hierüber das *Frontend* der Steuerungsebene angeboten. Durch die Bereitstellung von weiteren Instanzen des API-Servers, kann dieser horizontal skaliert werden. Als Sicherungsspeicher für die Daten des Clusters wird *etcd* [Etc20] eingesetzt. *Etcd* ist ein konsistenter und hochverfügbarer Schlüsselwertspeicher für kritische Daten in einem verteilten Umfeld und sollte möglichst mit einem *Backup*-Plan versehen werden. Des Weiteren kommt ein *kube-schedular* zum Einsatz, welcher *Pods* überwacht, denen noch kein Knoten zugewiesen wurde. Diese Komponente wählt daraufhin unter Berücksichtigung einiger Faktoren für die Planungsentscheidung einen *Node* für die Ausführung aus. Die letzte Komponente der *Control Plane* ist der *kube-controller-manager*, der verschiedene *Controller* in der Steuerungsebene verwaltet. Er ist u. a. verantwortlich für das Erkennen und Reagieren bei einem Knotenausfall (*Node Controller*), für die Verwaltung der korrekten Anzahl von *Pods* (*Replication Controller*), für die Endpunktverbindung zu den *Services* und *Pods* (*Endpoint Controller*) und für die Erstellung von Standardkonten und API-Zugriffstoken für neue Namensräume (*Service Account*, *Token Controller*).

Auch *Nodes* – sprich die Arbeitsknoten oder auch als *Worker* bezeichnet – setzen sich aus mehreren Komponenten zusammen, die die *Kubernetes*-Laufzeitumgebung bereitstellen [Kub20d]. Zum einen ist dies der Agent *kubelet*, der auf jedem *Node* im Cluster ausgeführt wird und sicherstellt, dass die Container in einem *Pod* aktiv sind. Er kontrolliert die spezifischen *Pod*-Beschreibungen (*PodSpecs*) auf die fehlerfreie Umsetzung im Ausführungskontext und kommuniziert mit der *Control Plane*. Der *kube-proxy* hingegen ist ein Netzwerk-*Proxy*, welcher Netzwerkregeln verwaltet und die Kommunikation der *Pods* innerhalb und außerhalb des Clusters ermöglicht. Zudem beherbergt er auch Lastenausgleichsfunktionen für den Netzwerkdatenverkehr. Der letzte hier zu erwähnende Bestandteil ist die *Container Runtime*, die für die Ausführung von Containern verantwortlich ist. *Kubernetes* unterstützt mehrere Laufzeitumgebungen für Container. Dies sind u. a. *Docker*, *containerd*, *cri-o*, *rktlet* und jede weitere Implementierung des *Kubernetes CRI* (*Container Runtime Interface*).

Um eine Anwendung in *K8s* bereitzustellen, muss sie in Container-*Images* verpackt, diese *Images* dann in eine *Image Registry* hochgeladen und abschließend eine Beschreibung der Container-Anwendung z. B. mit Hilfe der *kubectl* über den *kube-api-server* veröffentlicht werden. Diese Bereitstellung sollte wenn möglich nicht in einem bloßen *Pod*-Kontext erfolgen [Kub20a]. Bei einem Knotenausfall würden betroffene *Pods* dann nämlich nicht wieder ausgerollt werden. Nach den *Best Practices* ist es daher sinnvoll, den Kontext eines *Deployments* zu verwenden, der dafür sorgt, dass die gewünschte Anzahl an *Pods* – auch Replikate – immer verfügbar ist und eine Strategie zum Ersetzen dieser vorsieht [Kub20b]. Es wird hiermit sozusagen die Gesamtkomposition einer Anwendung beschrieben, die letztendlich im Cluster vorherrschen soll und durch *Kubernetes* kontinuierlich kontrolliert wird. Für die interne Kommunikation unterschiedlicher *Pods* miteinander im Netzwerk und die externe Erreichbarkeit einer Anwendung, ist es zudem notwendig die *Deployments* durch einen *Service* bekanntzumachen [Kub20k]. Ein *Service* muss wiederum vor den *Pods* erstellt werden, die auf diesen Zugriff erlangen sollen. Essential wird diese *K8s*-Ressource aufgrund der Tatsache, dass Verbindungen mit den entsprechenden *Pods* / Containern möglich sein müssen, auch wenn diese im Cluster dynamisch verschoben werden. Hat der API-Server die am Anfang beschriebene Anwendung verarbeitet, dann plant der *kube-schedular* die angegebenen Container-Gruppen auf die zur Verfügung stehenden *Nodes* ein. Berücksichtigt werden dabei die freien Ressourcen der Knoten und die Anforderungen der Bereitstellung. Im jeweiligen Arbeitsknoten weist daraufhin der *kubelet* die *Container Runtime* an, die involvierten Container-*Images* herunterzuladen und die Container auszuführen. [Luk18, vgl. S. 22ff]

Der Einsatz von *Kubernetes* bringt einige Vorteile für die Ausführung von Anwendungen mit sich, da es ein komplettes Rechenzentrum als eine einzige, geschlossene Rechenressource vereinfacht darstellen kann. Sind Entwickler erst einmal mit dieser Plattform vertraut und haben die anfängliche Komplexität überwunden, dann können sie ihre Anwendungen grundsätzlich ohne die Hilfe des Betriebsteams bereitstellen. Außerdem sind durch den Einsatz der automatischen Container-Orchestrierung Anwendungen von der Infrastruktur entkoppelt. Dies bewirkt eine bessere Ausnutzung der dahinterliegenden Hardware, denn Anwendungen können frei im Cluster-Verbund platziert werden. Des Weiteren ermöglicht *K8s* durch die kontinuierlichen Cluster-Zustandsprüfungen eine Art „Selbstheilung“. Ausgefallene Knoten oder darauf befindliche Ressourcen werden automatisch behandelt und ggf. neu bereitgestellt bzw. repariert. Durch die ständige Überwachung lassen sich auch die Anzahl der laufenden Instanzen der einzelnen Komponenten fortlaufend anpassen. Diese automatische Skalierung bietet aus technischer Sicht einen großen Mehrwert. Es ist diesbezüglich, bspw. bei vorübergehenden Lastspitzen, kein manuelles Eingreifen des *DevOps*-Teams mehr notwendig. [Luk18, vgl. S. 24ff]

3.6 *Rancher Labs*

Das Unternehmen *Rancher Labs* wurde 2014 gegründet und entwickelt *Open Source*-Software weitestgehend im Bereich der Container-Technologien. Mit den selbst entwickelten Werkzeugen hat es sich *Rancher Labs* als oberste Priorität gesetzt, *Kubernetes* in beliebigen Infrastrukturen (*On-Premise*-Rechenzentrum, *Cloud*, IoT – *Internet of Things*, *Edge*) skalierbar bereitzustellen und zu verwalten. Das aufgeschlüsselte Produktportfolio kann der Abbildung 3.5 zur Übersicht entnommen werden. Das mittig angeordnete Hauptprodukt ist *Rancher*, welches eine komplette Cluster-Verwaltungsplattform für einen einfachen Einstieg in die Arbeit mit allen zertifizierten Arten von *Kubernetes*-Installationen darstellt. Es ist eine der führenden *K8s*-Verwaltungsplattformen der Branche [BD18, Lin20]. Zu den *Rancher*-Schlüsselfunktionen zählen u. a. konsistente Cluster-Operationen und ein Infrastruktur-Management, die Verwaltung von (Sicherheits-)Richtlinien und authentifizierten Benutzern über ein RBAC, sowie die Bereitstellung von Anwendungen und weiteren Hilfswerkzeugen über eine Benutzeroberfläche, mit einer CLI oder auch API. Als untere Basis für die einzelnen Cluster-Knoten kommen dabei zertifizierte *Kubernetes*-Distributionen zum Einsatz. Mit der *Rancher Kubernetes Engine* (RKE, s. Unterabschnitt 3.6.1) und *K3s* besitzt *Rancher Labs* seine eigenen Implementierungen des von der CNCF verwalteten *K8s*. Währenddessen RKE ein nahezu vollwertiges *Kubernetes* für virtualisierte und *Bare Metal*-Umgebungen anbietet, erfüllt das *K3s* als leichtgewichtige Distribution seine Aufgaben in *IoT*- und *Edge*-Infrastrukturen. Außerdem können sämtliche *Kubernetes-Cloud*-Services, wie bspw. GKE (*Google Kubernetes Engine*), EKS (*Amazon, Elastic Kubernetes Service*) und AKS (*Azure Kubernetes Service*), in einer Plattform integriert und die damit einhergehenden *Cloud Computing*-Ressourcen verwaltet werden. Mit dem noch in der Beta-Phase befindlichem Produkt *Rio*, eröffnet *Rancher Labs* die Möglichkeit einem von Rancher verwalteten Cluster eine *Application Deployment Engine* hinzuzufügen. Hiermit soll die Benutzererfahrung gesteigert werden, sodass letztendlich auf noch einfachere Weise Anwendungen in einem *Kubernetes*-Cluster (automatisiert) bereitgestellt werden können. [Ran19, Ran20f]

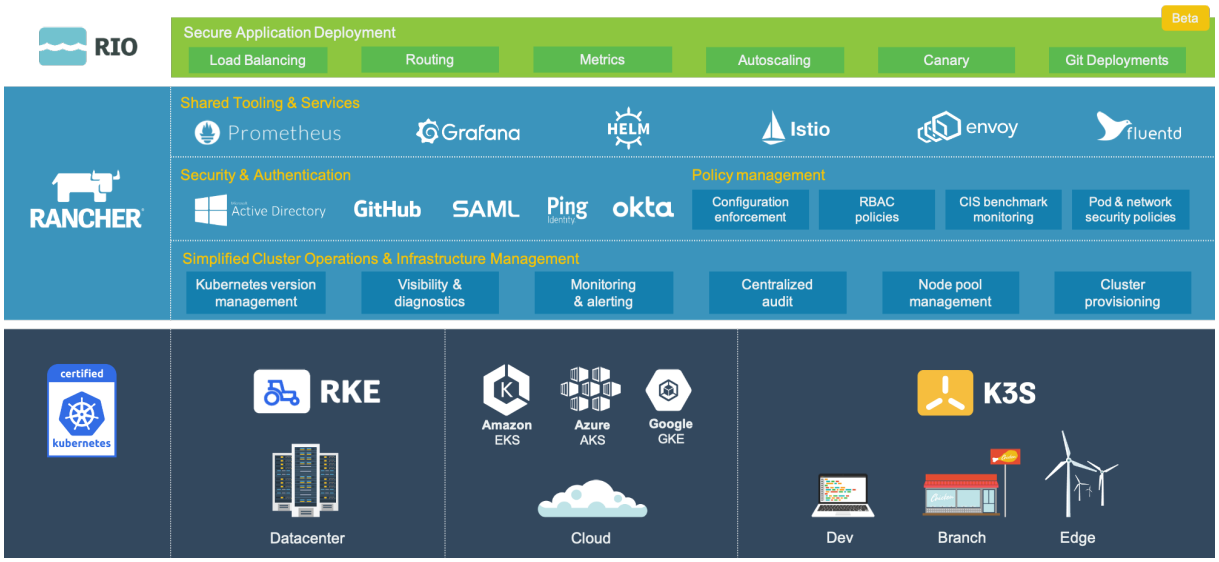


Abbildung 3.5: Bestandteile einer *Rancher*-Umgebung [Ran20f, S. 4]

Nachfolgend werden noch einmal im Unterabschnitt 3.6.1 die *Kubernetes*-Distribution RKE und im Unterabschnitt 3.6.2 die Cluster-Verwaltungsplattform *Rancher* differenziert anhand der zugrunde liegenden Architektur betrachtet.

3.6.1 *Rancher Kubernetes Engine* (RKE)

RKE ist eine von der CNCF zertifizierte *Kubernetes*-Distribution, die vollständig durch Container-Virtualisierung umgesetzt wird. Sie erleichtert die Installationskomplexität von *Kubernetes* und lässt den Betrieb eines Clusters nahezu unabhängig von der zugrunde liegenden Plattform automatisieren. Vor allem in *VMware*-Umgebungen, auf *Bare Metal*-Hosts und VM-Instanzen in *Clouds*, die noch keinen *Kubernetes*-Service anbieten, kann RKE besonders nützlich sein. Darüber hinaus kann die Distribution natürlich auch bei *Cloud*-Anbietern verwendet werden, die bereits entsprechende Services anbieten, um überall eine einheitliche bzw. konsistente *K8s*-Implementierung zu verwenden. Cluster-Bereitstellungen können sowohl auf *Linux*- als auch auf *Windows*-Systemen – hier allerdings nur auf Arbeitsknoten beschränkt – vorgenommen werden. Innerhalb von *Rancher* verwaltet RKE sozusagen den gesamten Lebenszyklus eines *Kubernetes*-Clusters beginnend bei der initialen Installation bis zum fortlaufenden und produktiven Betrieb. Im Zusammenspiel mit *Rancher* werden dem Administrator u. a. folgende Funktionalitäten geboten: Automatisierte Bereitstellung von VM-Instanzen in vielen *Cloud*-Umgebungen durch spezielle Treiber, Installation von *K8s-Control Plane*- und *etcd*-Knoten, Bereitstellen von *Workern* (Arbeitsknoten) auf *Windows*- und *Linux*-Hosts, Hinzufügen und Entfernen von *Nodes* in bestehenden Clustern, Aktualisieren von *Kubernetes*-Clustern auf eine neue Version und die Zustandsüberwachung von bereitgestellten Clustern. [Ran20f, vgl. S. 4]

3.6.2 *Rancher*

Mit Hilfe von *Rancher* können *Kubernetes*-Cluster (*On-Premises*- oder *Hybrid*-Instanzen) an einer zentralen Stelle verwaltet und erstellt werden. Das durch *Rancher Labs* bereitgestellte *Open Source*-Werkzeug erfüllt sowohl die Anforderungen von Benutzern, die Container-Anwendungen bereitstellen wollen, als auch die der Administratoren von Rechenzentren, die für die Erstellung und Instandhaltung von kritischen IT-Dienstleistungen verantwortlich sind.

Ein Blick auf die Abbildung 3.6, die die *Rancher*-Architektur mit einem oder mehreren zu verwaltenden *Downstream*-RKE-Clustern darstellt, verrät, dass die *K8s*-Verwaltungsplattform aus zwei zentralen Komponenten besteht. Dies sind auf der einen Seite der *Rancher Server*, welcher für die Verwaltung der gesamten *Rancher*-Bereitstellungen zuständig ist und auf der anderen Seite die Agenten-Komponenten, die in den *Kubernetes*-Clustern und -Knoten (*Downstream RKE Clustern*) ihre Aufgaben erfüllen.

Der *Rancher Server* hat vier zentrale Bestandteile, die nachfolgend erläutert werden. Den Hauptzugriffspunkt stellt der *Rancher API Server*, welcher einen *kube-api-server* integriert und zudem – wie schon bei *Kubernetes* selbst – den konsistenten und hochverfügbaren Sicherungsspeicher *etcd* [Etc20] für alle hierüber angelegten *Rancher*-spezifischen Ressourcen des Clusters verwendet. Dieser API-Server ist die grundlegende Schicht für alle anderen *Controller* im *Rancher Server*. Aktivitäten, die auf der Ebene des *Rancher Servers* und nicht spezifisch auf einem einzelnen *Downstream*-Cluster angesiedelt sind, werden durch einen zuständigen *Management Controller* realisiert. Über diese Komponente lassen sich diverse Vorlagen, Richtlinien, Kataloge und Benutzerberechtigungen global für die jeweiligen Cluster konfigurieren. Auch werden hierüber Cluster-Statistiken und -Ereignisse aggregiert und angezeigt. Des Weiteren ermöglicht erst der *Management Controller* die Bereitstellung von *Downstream*-Clustern mit Hilfe von speziellen *Cloud*-Treibern – u. a. auch für *VMware vSphere* – und / oder durch die Verwendung von RKE. Die Cluster Controller führen hingegen Aktivitäten aus, die eindeutig einem bestimmten *Downstream*-Cluster zuzuordnen sind. Es können hierdurch z. B. *Workloads* wie die *K8s*-Ressourcen *Pod* oder *Desployment* erzeugt und verwaltet, oder auch diverse Richtlinien, Quoten und *Secrets* auf den jeweiligen Cluster verteilt bzw. kontrolliert werden. Die Verbindung des *Cluster Controllers*

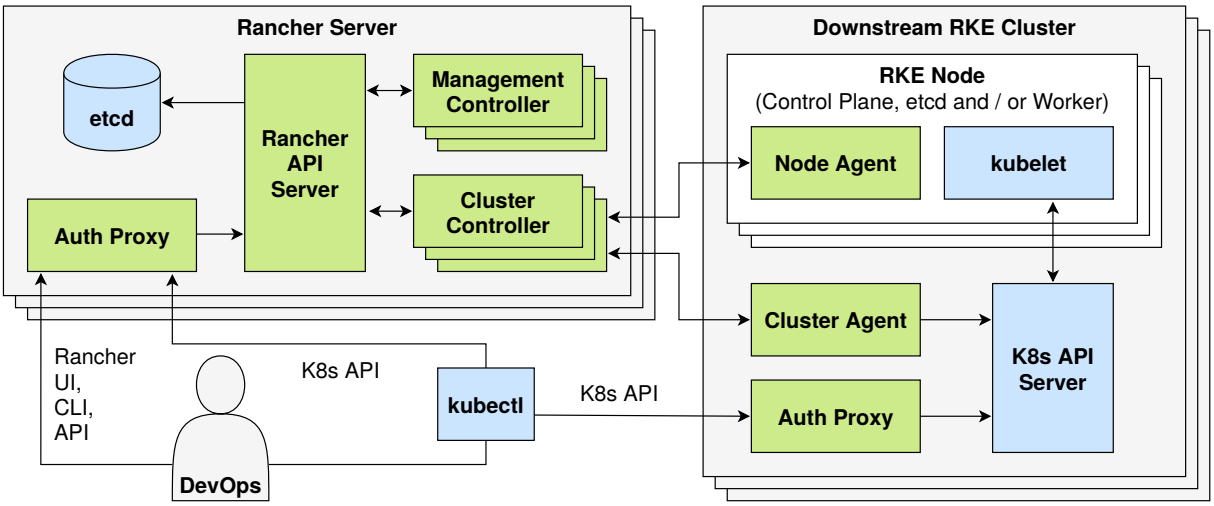


Abbildung 3.6: *Rancher*-Architektur mit *Downstream*-RKE-Cluster [Ran20f, vgl. S. 8], [Ran20c]

zu einem API-Server eines RKE-Clusters, wird letztendlich über einen *Cluster Agent* ermöglicht. Die letzte Komponente im *Rancher Server* ist der *Auth Proxy*, über den alle *Kubernetes*-API-Aufrufe gestartet werden. Er integriert zudem einen lokalen Authentifizierungsdienst oder auch globale Verzeichnisdienste zur Identitätsprüfung. Bei jeder Anfrage wird der Aufrufer (*DevOps*) im ersten Schritt authentifiziert, dann werden die richtigen *Kubernetes-Header*-Daten gesetzt und anschließend wird er an die entsprechende *Kubernetes-Control Plane* weitergeleitet. *Rancher* verwendet für die Verbindung mit einem verwalteten Cluster ein Dienstkonto. Darüber hinaus ist es möglich auch direkt mit einem *Downstream*-Cluster zu kommunizieren, ohne Komponenten des *Rancher Servers* zu verwenden. Dadurch wird schließlich die Verfügbarkeit erhöht, indem der Authentifizierungspfad abgekürzt wird. [Ran20f, vgl. S. 8f.], [Ran20c]

Die Agenten-Komponenten, die von *Rancher* in den verwalteten *Downstream*-Clustern eingesetzt werden, bestehen im Wesentlichen aus zwei verschiedenen Agenten. Für jeden *Downstream*-Cluster wird jeweils ein sogenannter *Cluster Agent* eingesetzt. Damit der zuständige *Cluster Controller* und der *Auth Proxy* mit dem *K8s API Server* kommunizieren können, öffnet der *Cluster Agent* einen *WebSocket*-Tunnel zurück zum *Rancher Server*. Außerdem agieren diese Agenten als *Proxy* für diverse weitere Dienste, die somit im Cluster über den *Cluster Agent* erreicht werden können. Während der Registrierungsphase erhalten diese Agenten vom neuen *K8s*-Cluster auch das Dienstkonto mit den Anmeldedaten, welche anschließend dem *Rancher Server* zur Verfügung gestellt werden. Der zweite Agenten-Typ sind die *Node Agents*. Sie werden von RKE hauptsächlich bei der Erstinstallation zur Bereitstellung der *Kubernetes*-Komponenten und bei späteren Aktualisierungen dieser verwendet. Darüber hinaus erfüllen die Knoten-Agenten zwei weitere Funktionen in allen Clustern. Falls der Cluster-Agent aus irgendeinem Grund ausfallen sollte, stellt der *Rancher Server* über einen *Node Agent* eine Verbindung mit dem *K8s API Server* her. Des Weiteren wird über diese Agenten die *kubectl-Shell* in der Benutzeroberfläche getunnelt. Für diesen Vorgang, der letztendlich vom *Rancher Server* ausgeht, werden höhere Berechtigungen im *Downstream*-Cluster benötigt, die nur die *Node Agents* besitzen. [Ran20f, vgl. S. 10], [Ran20c]

Die komplette *Rancher Server*-Instanz lässt sich durch einen Upgrade-Vorgang einfach aktualisieren. Die von *Rancher* ausgerollten *Downstream*-Cluster in Ausprägung der *Kubernetes*-Distribution RKE lassen sich hierüber ebenso zentral aktualisieren. Um die Hochverfügbarkeitsanforderungen eines *Rancher Servers* zu gewährleisten, kann dieser selbst in einem dedizierten

RKE-Cluster betrieben werden. Standardmäßig wird eine Bereitstellung mit insgesamt drei Knoten vom Hersteller *Rancher Labs* befürwortet, auf denen jeweils eine Instanz des API-Servers und der *etcd*-Datenbank läuft. Bezogen auf die Skalierbarkeit ist zu sagen, dass reine *K8s*-Cluster bis zu 5.000 Knoten verwalten können. *Rancher* hingegen kann jeweils *Downstream*-RKE-Cluster mit bis zu 100.000 Knoten verwalten und in Infrastrukturen bereitstellen. Für den *Rancher Server* an sich gibt es keine inhärente Begrenzung, wie viele einzelne Cluster-Instanzen dieser verwalten kann. Laut *Rancher Labs* sollte die Verwaltung von bis zu 2.000 Clustern keine Schwierigkeiten darstellen. Limitierende Faktoren sind eher die totale Anzahl von Knoten aller verwalteten Cluster, Benutzer und Gruppen sowie die Sammlung von allen anfallenden Ereignisdatensätzen. [Ran20f, vgl. S. 10f.]

3.7 *GitLab*

GitLab stellt eine komplette *DevOps*-Plattform in einer einzigen ausgelieferten Webanwendung bereit. Das *Open Source*-Projekt unterstützt die kollaborierende Software-Entwicklung durch die Versionsverwaltung des Software-Quellcodes auf der Basis von *Git* und diversen weiteren Komponenten, wie bspw. die einer *CI / CD Pipeline*. Durch *GitLab* CI / CD, ein System zur kontinuierlichen Integration und Auslieferung von Software, kann Anwendungs-Quellcode automatisierbar durch eine Pipeline-Verarbeitung gebaut, getestet und anschließend in benutzerdefinierten Umgebungen bereitgestellt werden. Um einen solchen Pipeline-Prozess durchzuführen, wird mindestens ein sogenannter *GitLab Runner* benötigt. Diese spezielle Anwendung arbeitet letztendlich über eine API wieder mit *GitLab* CI / CD zusammen und führt die einzelnen Phasen (*Stages*) einer Pipeline aus. *Git-Repositories* können sowohl mit einer *GitLab*-Installation auf eigener Hardware betrieben werden oder es wird die kostenfreie *SaaS*-Möglichkeit auf der Hersteller-Webseite unter der URL „`https://gitlab.com`“ verwendet. [Git20c, Git20a]

3.8 Zusammenfassung

In diesem Kapitel wurden Software-Produkte bzw. Infrastrukturkomponenten aufgezeigt, die die allgemeinen Ansätze des Grundlagen-Kapitels konkretisieren. Im Kontext einer *On-Premise*-Bereitstellung dient die bereits vorhandene Produktlinie *VMware vSphere* mit ihren *ESXi-Hypervisorn* als Virtualisierungsumgebung. Außerdem wird *Docker* zur Container-Virtualisierung und als *Container Runtime* auf den jeweiligen Hosts im Zuge des *Microservice*-Architekturstils benötigt. Damit *Docker Images* in einer verteilten Cluster-Umgebung verwendet werden können, bietet sich *Harbor* als Container-*Image Registry* an. In ihr können Abbilder über einen eindeutigen Namen abgelegt und zu einem anderen Zeitpunkt wieder abgerufen werden. Zusätzlich werden Werkzeuge für die digitale Signierung und die Sicherheitsüberprüfung von *Docker Images* mitgeliefert. Das *Cloud Native*-Paradigma lässt sich wiederum mit einem *Ubuntu Cloud Image* als Knoten-Betriebssystem für die Bereitstellung von virtualisierten Clustern vereinbaren. Als Plattform für die Container-Orchestrierung wird letztendlich *K8s* notwendig, um die Vorteile von großen, skalierbaren Cluster-Umgebungen auszunutzen und eine Vielzahl von Containern automatisiert zu administrieren. Die Erstellung und das Verwalten solcher *Kubernetes*-Cluster kann durch erweiterte Konfigurationswünsche und zunehmender Anzahl komplex ausfallen. Abhilfe kann hierbei das Management-Tool *Rancher* mit der *Kubernetes*-Distribution RKE schaffen, welche die Komplexität durch weitestgehend automatisierte Verfahren absenken können. Auch hohe Einarbeitungszeiten in die *K8s*-Materie durch den vielfältigen Funktionsumfang kann durch die Kombination der *Rancher Labs*-Produkte gering gehalten werden. Dies schließt natürlich auch Bereitstellungen von Anwendungen mit ein, die über den *Rancher Server* mit Hilfe unterstützter

Workflows und Kataloge getätigt werden können. Auch die Integration der *DevOps*-Plattform *GitLab* kann zu einer verbesserten und kontinuierlichen Anwendungsbereitstellung in diesem Kontext führen. Zu den Automatisierungsmechanismen bietet es sich zudem an, ein *Self-Service*-Portal durch die *VMware*-Produktreihe der *vRealize Suite* einzubinden. Im Endeffekt können dann einem Kunden über *vRA* und durch den im Hintergrund agierenden *vRO* Dienste wie *CaaS On-Premise* angeboten werden. Wie bereits in der Einleitung erwähnt, ist das angestrebte Ziel dieser Ausarbeitung, die hier erwähnten Interaktionsplattformen bzw. Komponenten miteinander zu verschaltet. Dabei dürfen die Aspekte der Automatisierung und vor allem der Sicherheit nicht aus dem Sichtfeld geraten. Weitere Begründungen, warum diese Auswahl getroffen wurde, können den nachfolgenden Kapiteln entnommen werden.

4 Anforderungen

Im Allgemeinen wird zwischen drei Arten von Anforderungen unterschieden [PR15, vgl. S. 8f.]: Funktionale Anforderungen, Qualitätsanforderungen und Randbedingungen. Funktionale Anforderungen legen dabei die Funktionalität eines geplanten Systems fest, die von diesem gefordert wird. Qualitätsanforderungen hingegen sind Bedürfnisse, die sich auf ein Qualitätsmerkmal (Performanz, Verfügbarkeit, Zuverlässigkeit, Skalierbarkeit, Portabilität etc.) beziehen und nicht von den funktionalen Anforderungen erfasst werden. Die Umsetzungsmöglichkeiten der beiden Anforderungsarten werden wiederum von Randbedingungen eingeschränkt, die nicht explizit umgesetzt werden, sondern lediglich den Rahmen vorgeben. In diesem Kapitel werden für das umzusetzende Szenario die funktionalen Anforderungen sowie die Qualitätsanforderungen und die Randbedingungen als nicht-funktionale Anforderungen eingeordnet und aufgeführt. Liebel bietet für die Planung, den Aufbau und Betrieb von Container-Clustern bereits hilfreiche Stichpunkte an [Lie19, vgl. S. 1357ff]. Darüber hinaus haben sich die nachstehenden Anforderungen aus den umzusetzenden Anwendungsfällen und den Vorgaben des AWIs ergeben.

Das zu erwartende Ergebnis dieser Ausarbeitung wird in der Abbildung 4.1 verdeutlicht. Hier werden vereinfacht die Anwendungsfälle des betrachteten Szenarios grafisch dargestellt.

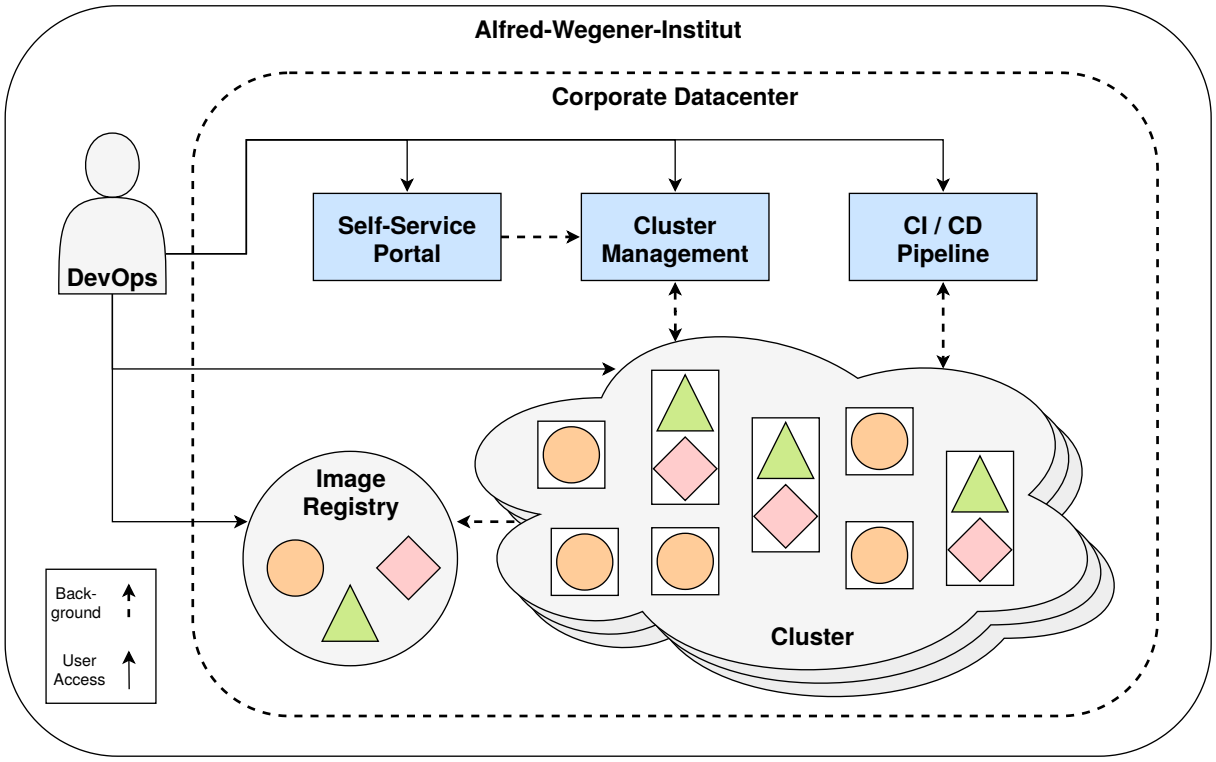


Abbildung 4.1: Kontextdiagramm des Szenarios

Im Rechenzentrum (*Corporate Datacenter*) des *Alfred-Wegener-Instituts* wird für Wissenschaftler-Innen (*DevOps*) die Möglichkeit bestehen, eigene *On-Premises*-Cluster für Container-Anwendungen über ein bereits vorhandenes webbasiertes *Self-Service Portal* zu „bestellen“ bzw. anzufordern.

Daraufhin wird über einen zu erstellenden *Workflow* die angeforderte Ressource automatisiert in der vorhandenen Virtualisierungsumgebung über einen *Orchestrator* (*Cluster Management*) bereitgestellt. Außerdem wird über diese (grafische) *Cluster Management*-Plattform die Administration und Überwachung eines zuvor ausgerollten Clusters für den Endanwender gegeben sein. Des Weiteren soll dieser direkt hierüber die Bereitstellung seiner entwickelten Container-Anwendungen vornehmen. Ggf. wird dazu ein direkter Zugriff auf die Knoten des Clusters für Fortgeschrittenere existieren, um die nativ zur Verfügung stehenden Cluster-Werkzeuge, wie bspw. eine CLI oder API, zu verwenden. Ein IP-basierter (*Internet Protocol*) Zugang auf die im Cluster befindlichen Container-Anwendungen über entsprechende Ports und Protokolle wird selbstverständlich auch möglich sein. Die Einbindung einer noch fehlenden *Image Registry*, um Container-*Images* innerhalb des Rechenzentrums zu verwalten, wird ebenfalls vorgenommen. Zusätzlich wird es über eine benutzerdefinierte *CI / CD Pipeline* ermöglicht, automatisiert Anwendungen im Cluster-Verbund bereitzustellen. Dies intensiviert wiederum den in der Ausarbeitung eingangs beschriebenen *Continuous Delivery*-Gedanken der *DevOps*-Vorgehensweise.

4.1 Funktionale Anforderungen

Folgende funktionale Anforderungen, die den einzelnen Interaktionsplattformen zugeordnet werden können, sollen durch die Ausarbeitung im Rechenzentrum des AWIs konkret umgesetzt werden:

- *Image Registry* –
  Es wird eine *Image Registry* in der Virtualisierungsumgebung installiert und eingerichtet. Dabei werden Maßnahmen für die Hochverfügbarkeit und die Replikation der *Repositories* entwickelt und ggf. umgesetzt. Des Weiteren wird die möglichst RBAC-basierte Benutzerverwaltung mit Hilfe eines internen Verzeichnisdienstes konfiguriert. Die Verwaltung von Projekten bzw. *Repositories* wird getestet und somit das Hochladen, Herunterladen und Löschen von Container-*Images* gewährleistet. Darüber hinaus werden sicherheitsrelevante Konfigurationen an dieser Interaktionsplattform durchgeführt, u. a. eine Schwachstellen-Analyse einzelner *Image Layers* mit einem CVE-*Scanner* und das Signieren von Container-*Images* (*Trusted Images*).
- *Cluster Management* –
  Es wird ein *Cluster Management* in der Virtualisierungsumgebung installiert und eingerichtet. Dabei werden Maßnahmen für die Hochverfügbarkeit und Lastverteilung sowie die ausschließliche Verwendung einer *Private Registry* entwickelt und umgesetzt. Des Weiteren wird die möglichst RBAC-basierte Benutzerverwaltung mit Hilfe eines internen Verzeichnisdienstes konfiguriert. Um die automatisierte Bereitstellung eines Clusters in der Virtualisierungsumgebung zu veranlassen, werden Vorlagen (*Templates*) für *Nodes* und Cluster erstellt. Nach der Provisionierung wird die Administration und Überwachung eines zuvor angeforderten Clusters getestet. Außerdem werden hierüber Anwendungsbereitstellungen mit Hilfe der Benutzeroberfläche, Kataloge oder auch nativ über eine CLI durchgeführt. Darüber hinaus werden sicherheitsrelevante Konfigurationen an dieser Interaktionsplattform durchgeführt, u. a. *Namespaces*, *Network Policies*, *Pod Security Policies* (PSP), *Hardened Templates* etc.
- *Self-Service Portal* –
  Im webbasierten *Self-Service Portal* wird ein Katalogeintrag für den autonomen Bestellvorgang eingerichtet. Zudem wird hier ein automatisierter *Workflow* entwickelt bzw. hinterlegt, um einen Cluster nach Anforderung über das *Cluster Management* ausrollen zu lassen. Ggf. lassen sich noch weitere Katalogeinträge vornehmen, um z. B. einem vorhandenen

Cluster *Nodes* (*Worker*) hinzuzufügen bzw. zu entfernen, oder Container-Anwendungen direkt in einem schon vorhandenen Cluster durch die Aufnahme von neuen Nutzerkonten bereitzustellen.

- *CI / CD Pipeline* –
  Die *CI / CD Pipeline* wird durch das Anlegen eines *Repositories* in der vorhandenen *DevOps*-Plattform realisiert. Des Weiteren wird eine Beispielanwendung basierend auf der Container-Virtualisierung hinzugefügt, die letztendlich in einem Cluster-Verbund ausgerollt werden soll. Wesentlich für diese Interaktionsplattform ist der Aufbau einer minimalen Pipeline und der Vorgang der kontinuierlichen Bereitstellung einer Container-Anwendung in einem zuvor angeforderten Cluster.

4.2 Nicht-funktionale Anforderungen

Die nachstehenden nicht-funktionalen Anforderungen sollen bei der Umsetzung des angedachten Szenarios berücksichtigt werden:

- Sicherheit –
  Die eingesetzten Systeme sollen „sicher“ sein, also die aktuellen Sicherheitsstandards nach dem Stand der Technik erfüllen und die *Best Practices* der Hersteller und Softwareentwickler verwirklichen.

- Verwendung vorhandener Systeme –
  Es sollen die *On-Premise*-Virtualisierungsumgebung *VMware vSphere* mit dem *Self-Service*-Portal und den Werkzeugen zur Automatisierung der *vRealize Suite* sowie die bereits eingeführte *DevOps*-Plattform *GitLab* verwendet werden.

- Plattformunabhängigkeit –
  Die Plattformübergreifende Nutzung der Interaktionsplattformen durch die zugrunde liegende heterogene Systemlandschaft aus (*Ubuntu*) *Linux*-, *Apple Mac OS X*- und *Microsoft Windows*-Clients muss gewährleistet werden.

- Wartbarkeit –
  Die Wartbarkeit (Aktualisierung, Konfiguration etc.) der angeforderten Cluster soll für das Betriebsteam überschaubar bleiben. Es können keine extra Mitarbeiter hierfür abgestellt werden. Ein zentrales Monitoring im *Cluster Management* soll zur Unterstützung dienen. Eine Wartung im laufenden Betrieb durch eine verteilte Auslegung der System-Architekturen soll gewährleistet werden.

- Anwendungsbereitstellung –
  Auch ohne spezifische Kenntnisse der zugrunde liegenden Bereitstellungsplattform soll ein einfaches Ausrollen einer Container-Anwendung in einem Cluster möglich sein. Hierfür eignen sich z. B. eine intuitive Benutzeroberfläche oder die Integration von rudimentären *CI / CD Pipelines*.

- Verfügbarkeit –
  Die neu zu installierenden Plattformen sollen von ihrer Architektur her möglichst hochverfügbar und ausfallsicher ausgelegt sein. An geeigneten Stellen sollten auch *Load Balancer* (LB) für eine Lastverteilung und einer Zurverfügungstellung von Anwendungen integriert werden.

- Lizenzkosten, *Open Source* –
  Neu anfallende Lizenzkosten sollen gering gehalten oder besser sogar vermieden werden. Hierfür bietet es sich an, möglichst *Open Source*-Software einzusetzen.
- Knoten-Betriebssystem –
  Als Betriebssysteme für die Cluster-Knoten sollen extra angepasste Container- bzw. *Cloud Native*-Betriebssysteme eingesetzt werden.
- *Container Runtime* –
  Vorwiegend soll die Container-Virtualisierung mit Hilfe von *Docker* im Cluster-Verbund realisiert werden; später sollen auch weitere *Container Runtimes* integriert werden können.
- Föderierte Cluster –
  Im Kontext von HIFIS wird die organisationsübergreifende Zusammenarbeit der *Helmholtz*-Forschungszentren ausgebaut. Zu einem späteren Zeitpunkt soll daher die Nutzung und der Zugang eines Clusters im Umfeld einer Föderation, bspw. der DFN-AAI (*Deutsches Forschungsnetz-Authentication and Authorization Infrastructure*), möglich sein.
- Einbindung von *Cloud*-Anbietern –
  Es soll die Möglichkeit der späteren Anbindung von externen *Cloud*-Anbietern, wie bspw. *Amazon Web Services* (AWS), *Microsoft Azure* oder *Google Cloud Platform* (GCP), in der *Cluster Management*-Plattform bestehen.

5 Verwandte Arbeiten

In diesem Kapitel werden einige verwandte Arbeiten sowie erwähnenswerte Ansätze in Hinblick auf die sichere und automatisierte Bereitstellung von Container-Clustern insbesondere in Ausprägung von *Kubernetes*-Umgebungen vorgestellt. Die Erkenntnisse aus diesen Ausarbeitungen sollen herausgearbeitet und die vermeintlichen Bezüge zum angedachten Szenario vermerkt werden. Es wurde eine kleine Auswahl getroffen, die nachfolgend näher betrachtet wird.

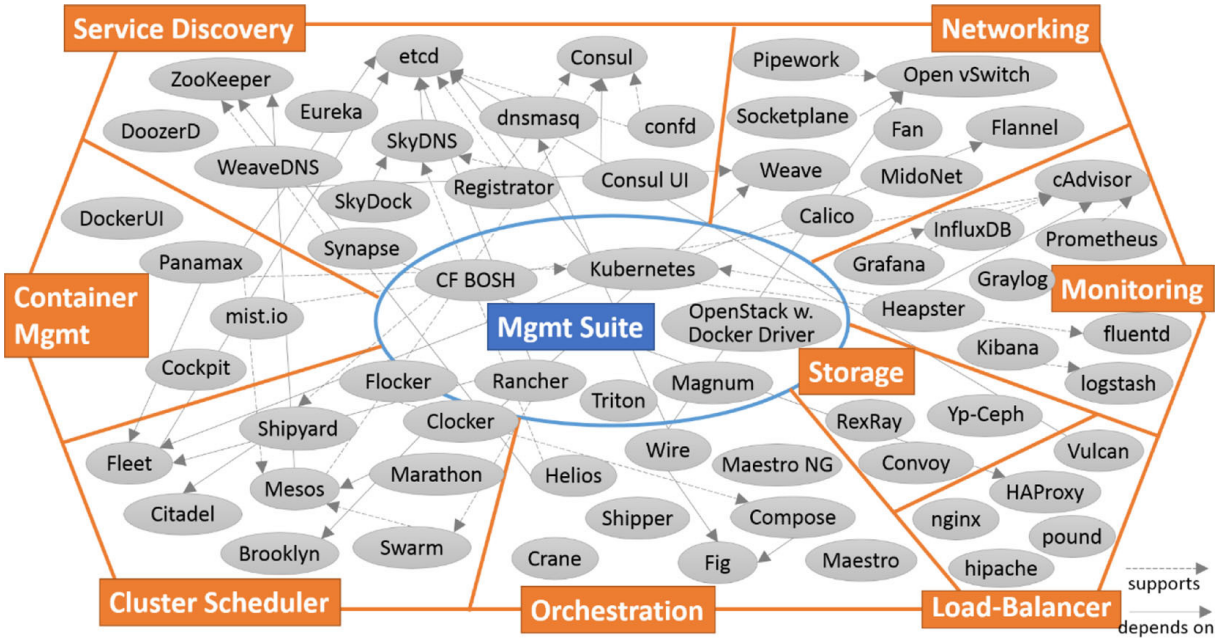


Abbildung 5.1: *Docker*-Ökosystem mit Abhängigkeiten [PHP16, S. 269]

Die Ausarbeitung in [PHP16] beschäftigt sich mit dem Aufbau einer *Open Source*-Container-Verwaltungslösung für mehrere Hosts. Als *Container Runtime* wird hierbei *Docker* verwendet. Der Bedarf einer solchen Lösung bestand, da *Docker* in der ursprünglichen Form für die Verwaltung von Containern nur auf einem Host spezialisiert ist. Anhand einer konkreten Umsetzung durch zuvor klassifizierte Anforderungen und der Gegenüberstellung von diversen Komponenten des *Docker*-Ökosystems – s. Abbildung 5.1 – wird eine gute Übersicht der zusammen agierenden Bestandteile solcher Verwaltungslösungen gegeben. Eine grobe Kategorisierung wurde dabei in *Container Mgmt*, *Cluster Scheduler*, *Orchestration*, *Service Discovery*, *Storage*, *Networking*, *Load Balancer*, *Monitoring* und letztendlich *Management Suits* vorgenommen. Anzumerken ist, dass nicht alle einzelnen Bestandteile für eine Umsetzung benötigt werden und die angebotenen Lösungen in dieser komplexen Domäne im stetigen Wandel sind. *Management Suites* wie *Kubernetes* und *Rancher* aggregieren bereits Einzelbestandteile der Gruppierungen und erleichtern damit den Aufbau von Container-Clustern. Allerdings konnten in diesem speziellen Kontext mit den vorhandenen *Suits* nicht alle Anforderungen erfüllt werden und es wurde eine eigene Lösung basierend auf *Apache Mesos* mit *Marathon* verwirklicht.

Auch die Ausführungen in [SF17] beschäftigen sich mit der Container-Orchestrierung und bestär-

ken noch einmal die Notwendigkeit einer Management-Ebene mit zusätzlichen Bestandteilen in verteilten Cluster-Umgebungen. Es wird ein abstraktes *Container Manager Pattern* vorgestellt, mit dem weitestgehend alle Anforderungen einer solchen Systemlandschaft abgedeckt werden. Durch die Umsetzung des *Patterns* können Cluster aufgrund der automatisierten Prozesse effektiver, sicherer und skalierbarer betrieben werden. Vorhandene Implementierungen, die allerdings nicht alle aufgeführten Bestandteile dieses Musters verwirklichen, werden mit *Kubernetes*, *Docker Swarm* und *Marathon* benannt. Gerechtfertigt ist der Einsatz solcher Werkzeuge nur in größeren Umgebungen, da mehr Komplexität verursacht wird und Anwender spezielle Fähigkeiten für Container-basierte Systeme benötigen. Auch sollten die Sicherheitsrisiken in diesem verteilten Aufbau vor der Einführung thematisiert werden.

In der Dissertation [Pal17] werden die unterschiedlichsten Virtualisierungstechnologien und -lösungen analysiert und bewertet. *Kubernetes* wird als funktionsreichere Lösung gegenüber anderen Container-*Orchestratorn* eingestuft. Allerdings wird auch hier die komplexe Architektur negativ angemerkt und aufgezeigt, dass *Kubernetes* nicht die beste Lösung für jeden Einsatzbereich darstellt. Palopoli sagt aus, dass eine starke Lernkurve vorausgesetzt wird, die allerdings durch den unterstützenden Einsatz von Management-Plattformen wie etwa *Rancher* oder einer noch weitreichenderen und vollständigeren Umgebung wie *OpenShift* abgeflacht werden kann.

Abdelmassih hat sich in [Abd18] mit den Aspekten der Container-Orchestrierung in sicherheitskritischen Umgebungen, wie die der schwedischen Polizeibehörde, beschäftigt. Untersucht wurde die Isolation zwischen *Docker*-Anwendungen in *Kubernetes*-Clustern bzw. wie dies am besten erreicht werden kann. Es wurden drei Architekturvorschläge ausgearbeitet, zwei beschäftigen sich dabei mit einer Trennung von Container-Anwendungen untereinander und der dritte mit einer Strategie zur Host-basierten Segmentierung von Containern. Abdelmassih kommt zum Schluss, dass eine ausreichende Isolation der Anwendungen grundsätzlich erreicht werden und der Einsatz von Container-Technologien in sicherheitsrelevanten Umgebungen stattfinden kann. Es sollten dennoch weitere Untersuchungen in diesem Themenkomplex getätigt und ausführlichere Sicherheitskonzepte erarbeitet werden; diese Arbeit könnte dafür als Grundlage dienen.

In der Masterarbeit [Lin18] wird die Konzeption und Realisierung einer *Cloud*-Plattform für die Datenverarbeitung von Werkzeugmaschinen behandelt. Link setzt dabei einen Schwerpunkt auf ein anbieterunabhängiges Multi-*Cloud*-Szenario und eine flexible Bedienbarkeit der Plattform ohne die Notwendigkeit von technischen Vorkenntnissen der Endkunden. Als Werkzeug für die Orchestrierung und für die Anwendungsbereitstellung wurde in diesem Zusammenhang *Rancher* im *Back-End* integriert.

Narjes hat sich in der Ausarbeitung [Nar18] mit der Umsetzung einer automatisierten Bereitstellung mehrerer zusammenhängender Container-Anwendungen in einer vorhandenen *Kubernetes*-Infrastruktur auseinandergesetzt. Realisiert wurde die kontinuierliche Bereitstellung der einzelnen Services über die *CI / CD Pipeline* eines *GitLab*-Servers. Narjes merkt an, dass bei der Nutzung des *Microservice*-Architekturstils keine komplette Bereitstellung der Gesamtanwendung vorgenommen werden sollte, es müssen nur die Bestandteile neu platziert werden, die auch wirklich zuvor im *Repository* angepasst wurden.

Die Untersuchungen in [VSTK19] beschäftigen sich mit der Verfügbarkeit von *Microservices* in *Kubernetes*-Clustern. Eine ständige Erreichbarkeit der ausgerollten Container-Anwendungen ist in produktiven Umgebungen von besonderer Wichtigkeit. In dieser Ausarbeitung werden verschiedene Architekturen für öffentliche als auch private *Cloud*-Umgebungen vorgestellt und die unterschiedlichen Zugriffsmöglichkeiten auf die eigentlichen Anwendungen im Cluster dargestellt. Es wurde festgestellt, dass *Kubernetes* besonders in öffentlichen *Clouds* seine Vorteile ausspielt. Falls ein Knoten ausfallen sollte, kann bei der Standardkonfiguration die vollständige Hochverfügbarkeit von Anwendungen durch die *K8s*-Selbstheilung nicht unbedingt gewährleistet werden.

Das Hinzufügen von Redundanzen gerade bei den bereitzustellenden (System-)Anwendungen kann die Ausfallzeit erheblich verringern.

Das Thema der IT-Sicherheit bei der Container-Virtualisierung wird noch einmal in [HB19] aufgegriffen. Das Bundesamt für Sicherheit in der Informationstechnik (BSI) stellt mit dem IT-Grundschutz-Kompendium [BSI20a] jährlich einen überarbeiteten Katalog zur Umsetzung von angemessenen Schutzmaßnahmen im IT-Umfeld zur Verfügung. Mit dem Baustein „*SYS 1.6: Container*“ [BSI20b] wurden dementsprechende Maßnahmen als *Community Draft*-Version veröffentlicht. Bei dieser Untersuchung wurde der IT-Grundschutz auf einem *Docker*-basierten *Web-Shop* angewendet und die vorgeschlagenen Maßnahmen des Katalogs kritisch begutachtet. Als Ergebnis stellte sich heraus, dass der Baustein „*SYS 1.6: Container*“ ein wertvolles Werkzeug für die Umsetzung eines Sicherheitskonzeptes im Kontext der Container-Technologien ist. Es wurden dennoch zwei weitere nicht aufgeführte Gefährdungen erkannt. Sie beziehen sich einmal auf einen möglichen „*Container Breakout*“ und andererseits auf die unautorisierte Änderung von Konfigurationsdateien.

Die Ausarbeitung in [Bjø19] fasst die gemachten Erfahrungen des Zentrums für Informationstechnologie der Universität Oslo mit der Einführung von Containern zusammen. Als zentrale *On-Premise*-Container-Plattform wird die *Openshift Community Distribution of Kubernetes* (OKD) eingesetzt, um zuverlässig Dienste am *Red Hat*-geprägten Campus anzubieten. Zuvor wurde eine *Docker*-Infrastruktur mit *Ansible* und *Jinja2* als Container-Management- und Orchestrierungs-Werkzeuge sowie einer *Private Registry* in Ausprägung von *Harbor* eingesetzt. Aufgrund von Verbesserungspotentialen in Bezug auf die Orchestrierung, Beobachtbarkeit, Selbstbedienung und Unterstützung für kontinuierliche und automatisierte Bereitstellungen wurde schließlich ein Wechsel zur OKD-Plattform angestrebt. Die replizierbare *Registry*-Lösung mit dem CVE-*Scanner Clair* wurde beibehalten. Zukünftig soll die Container-Umgebung der Universität Oslo noch weiter den *DevOps*-Gedanken verwirklichen.

Wie Godlove in [God19] beschreibt, verwendet *Singularity* als alternative *Container Runtime* ein einzigartiges Sicherheitsmodell und bietet eine einfache, sichere sowie funktionsreiche Container-Lösung vor allem im HPC-Umfeld (*High-Performance Computing*) an. Nicht vertrauenswürdigen Benutzern wird es hiermit ermöglicht, nicht vertrauenerweckende Container sicher auf Systemen mit mehreren Mandanten parallel zu betreiben. Als Grundlage dient das spezielle Dateiformat SIF (*Singularity Image Format*), in dem Container verpackt und verteilt werden. Laut Godlove hat sich *Singularity* zur Standard-Container-Lösung im HPC und zur bevorzugten Container-Plattform für rechenintensive Arbeitslasten überall auf der Welt entwickelt.

Der *451 Research*-Übersichtsartikel in [Lym19] befasst sich mit dem effektiven Unternehmenseinsatz von *Cloud Native*-Software wie etwa Container und *Kubernetes*. Der Bedarf solcher Technologien steigt stetig, um die anhaltenden Ziele im Bereich der digitalen Transformation zu erreichen. Die Organisationen stehen diesbezüglich aber auch vor großen Herausforderungen wie einer erhöhten Komplexität, Bedenken hinsichtlich der IT-Sicherheit, Lernbereitschaft sowie mangelnder Fähigkeiten. Dabei entscheidend kann das Finden eines Gleichgewichts zwischen einem zentralen als auch verteilten Managements dieser neuen Plattformen sein. Lyman analysiert wichtige Forschungsergebnisse zum aktuellen Stand und zu den Chancen sowie Herausforderungen bzgl. *Kubernetes* in Unternehmen. Das darauf aufbauende Papier in [Ate19] konkretisiert diese Überlegungen. Atelsek stellt das Software-Portfolio von *Rancher Labs* vor und plädiert für den Einsatz von *Rancher* mit RKE in Unternehmen bei einem überfüllten Markt an Optionen.

Die Marktuntersuchungen in [BD18] und [Lin20] geben eine Übersicht über die vorhandenen Lösungen im dynamischen Gebiet der *Enterprise Container Platforms* (ECP). In der Untersuchung, die von *Forrester* im Jahr 2018 durchgeführt wurde, sind acht verschiedene Anbieter (*Docker*, IBM, *Mesosphere*, *Pivotal*, *Platform9*, *Rancher Labs*, *Red Hat*, SUSE) vorwiegend für den

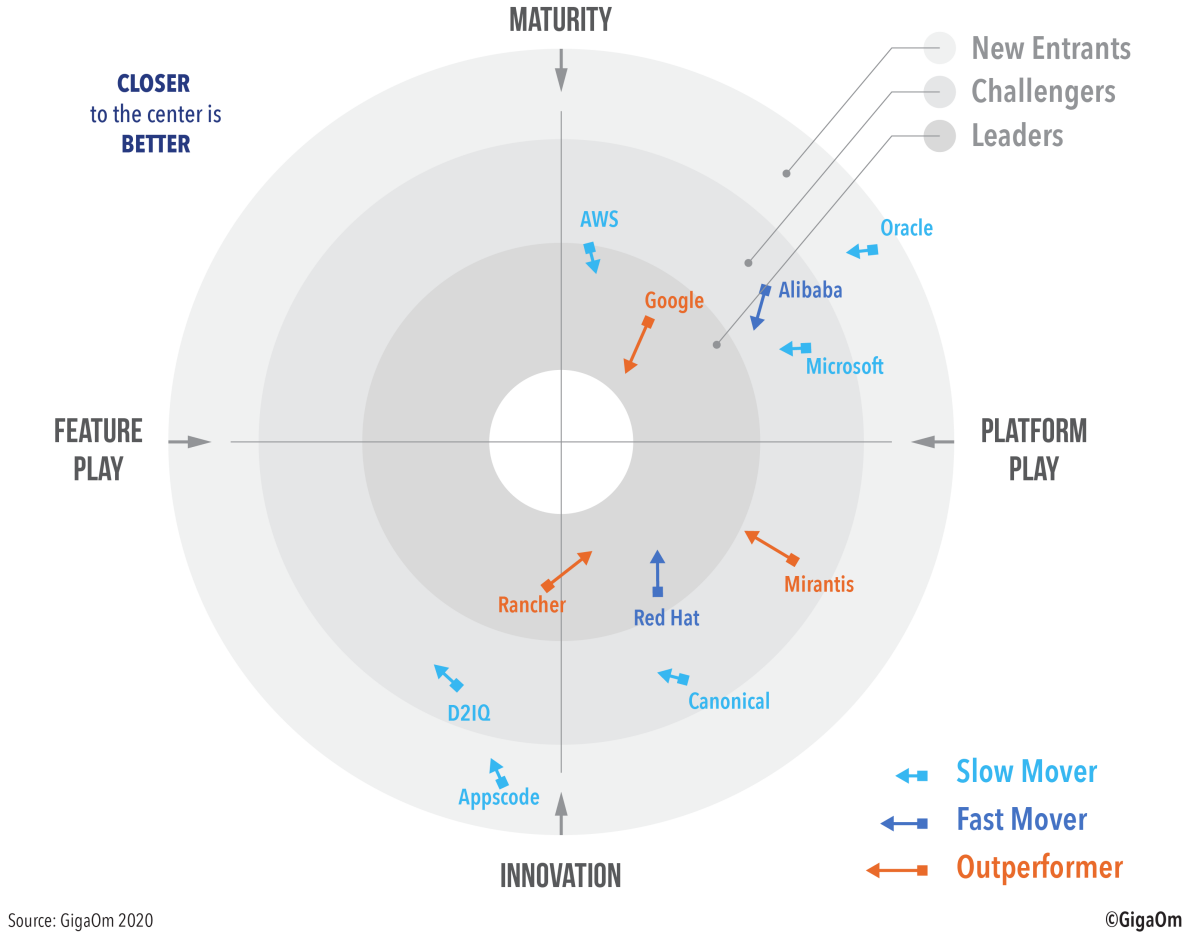


Abbildung 5.2: Markübersicht der Anbieter von *Enterprise Container Platforms* [Lin20, S. 9]

On-Premise-Bereich anhand von Kriterien miteinander verglichen worden. Vor allem die Produkte von *Docker*, *Red Hat* und *Rancher Labs* waren zu diesem Zeitpunkt durch ihren gebotenen Funktionsumfang auf dem Markt führend. Die aktuellere und von *GigaOm* durchgeführte Marktanalyse aus dem Jahr 2020 nimmt die großen *Cloud*-Anbieter (*Google*, *Amazon*, *Microsoft*) als Kandidaten mit auf und legt den Fokus verstärkt auf die mögliche Einbindung von *Kubernetes*-Clustern in Föderationen (*KubeFed*, s. Abschnitt A.8). Das Ergebnis dieser Auswertung ist der Abbildung 5.2 zu entnehmen. Für das angestrebte Szenario sind hier vorwiegend die Lösungsansätze von *Rancher Labs* und *Red Hat* interessant, welche sich wiederum zu den Marktführern im *On-Premises*- und *Cloud*-Segment zählen können, sowie laut Linthicum ein starkes Potential für die Zukunft besitzen.

Tabelle 5.1: Anforderungszuordnung der verwandten Arbeiten

Verwandte Arbeiten	**Funktionale Anforderungen**	**Nicht-funktionale Anforderungen**
[PHP16]	*Image Registry*, *Cluster Management*, *Self-Service Portal*	*Container Runtime* (*Docker*), Lizenzkosten, *Open Source*
[SF17]	*Cluster Management*	Sicherheit, Wartbarkeit, Verfügbarkeit
[Pal17]	*Cluster Management*	Wartbarkeit, Verfügbarkeit

[Abd18]	*Cluster Management*	Sicherheit, Verfügbarkeit, Knoten-Betriebssystem, *Container Runtime*
[Lin18]	*Cluster Management*	Wartbarkeit, Anwendungsbereitstellung, Einbindung von *Cloud*-Anbietern
[Nar18]	*Cluster Management*, *CI / CD Pipeline*	Verwendung vorhandener Systeme (*GitLab*), Anwendungsbereitstellung
[VSTK19]	*Cluster Management*	Wartbarkeit, Verfügbarkeit, Einbindung von *Cloud*-Anbietern
[HB19]	*Image Registry*, *Cluster Management*	Sicherheit, Knoten-Betriebssystem, *Container Runtime* (*Docker*)
[Bjø19]	*Image Registry*, *Cluster Management*, *CI / CD Pipeline*	Wartbarkeit, Anwendungsbereitstellung, Verfügbarkeit
[God19]	*Cluster Management*	Sicherheit, *Container Runtime* (*Singularity*)
[Lym19], [Ate19]	*Cluster Management*, *CI / CD Pipeline*	Wartbarkeit, Anwendungsbereitstellung, Verfügbarkeit, Lizenzkosten, *Open Source*
[BD18], [Lin20]	*Cluster Management*	Lizenzkosten, *Open Source*, Föderierte Cluster, Einbindung von *Cloud*-Anbietern

Generell ist anzumerken, dass in den verwandten Arbeiten die Vorzüge von skalierbaren Container-Infrastrukturen und die Entwicklung der Rechenzentren dort hin noch einmal ersichtlich werden; die Tabelle 5.1 liefert hierzu unterstützend eine Übersichtsaufstellung bezogen auf die zuzuordnenden Anforderungen aus dem Kapitel 4. Um der Komplexität der vielen dynamischen Bestandteile solcher Umgebungen, wie in der Abbildung 5.1 zu sehen ist, entgegen zu wirken, ist der Einsatz von Management-Plattformen notwendig. Für die Bereitstellung von *Kubernetes*-Clustern in *On-Premises*-Vorhaben eignen sich besonders die ECPs *Rancher* und *Red Hat OpenShift*, wie der Marktübersicht in der Abbildung 5.2 entnommen werden kann. In den *Cloud Computing*-Infrastrukturen nimmt hingegen *Google* eine vorherrschende Stellung ein. Mit Einzug der *Cloud Native*-Technologien darf dennoch die IT-Sicherheit nicht vernachlässigt werden. Das BSI stellt in diesem Zusammenhang das IT-Grundschutz-Kompendium zur Verfügung, welches zukünftig auch einen Baustein zu Containern enthalten wird; dieser wird im nächsten Kapitel noch einmal näher betrachtet. Interessant wäre in diesem Zusammenhang auch der Einsatz von „sichereren“ *Container Runtimes* wie etwa *Singularity* im Gegensatz zu *Docker*. Eine *Image Registry* wie bspw. *Harbor* stellt sozusagen das Fundament dar und sollte auch mit entsprechenden Maßnahmen gesichert werden. Die Automatisierung in Rechenzentren bzw. *Kubernetes*-Clustern mit Hilfe von *CI / CD Pipelines* spielt überdies eine immer wichtigere Rolle, um den *DevOps*-Gedanken in Unternehmensstrukturen voranzutreiben und zu verwirklichen.

6 Analyse

Die Container-Virtualisierung erhält in Unternehmen eine immer stärkere Gewichtung. Dies ist auch ganz konkret nach der im Jahr 2020 veröffentlichten Befragung [CNC20b] der CNCF zu schlussfolgern. Allerdings stehen Unternehmen dabei großen Herausforderungen gegenüber, die vor allem an den einhergehenden Kulturkonflikten innerhalb der Entwickler- und Betriebsteams, sowie an der Sicherheit und Komplexität der relativ neuen Container-Technologien liegen. Laut der von *451 Research* durchgeführten Analyse [Lym19, vgl. S. 4] werden dennoch 76% der befragten Unternehmen innerhalb der nächsten drei Jahre *Kubernetes* in ihrer IT-Infrastruktur aufgrund der konsistenten Funktionalitäten sowohl in eigenen Rechenzentren als auch in *Cloud*-Umgebungen standardisieren und somit einführen.

Durch den Einsatz der *K8s*-Plattform im IT-Betrieb kann eine durchgängig hohe Zuverlässigkeit der Infrastruktur geschaffen werden. Auch lässt sich hierdurch die Effizienz des *DevOps*-Gedankens durch Standards in der Automatisierung verbessern, sowie Sicherheitsrichtlinien in der Infrastruktur durchsetzen. Wird dabei auf die ursprüngliche Version von *Kubernetes* gesetzt, die in der *Community* als „*Vanilla Kubernetes*“ bezeichnet wird, können damit einige Risiken verbunden sein. Mehrere *K8s*-Cluster müssten unabhängig voneinander verwaltet werden, eine zentrale Sichtbarkeit bzw. Anlaufstelle wäre nicht vorhanden und globale Sicherheitsrichtlinien könnten nicht konsistent verankert werden. Abhilfe schafft die zentrale *Open Source*-Verwaltungsplattform *Rancher* mit RKE [Ran20f, vgl. S. 3]. Über diese *Enterprise Container Platform* (ECP) werden u. a. Schlüsselfunktionen wie konsistente Cluster-Operationen (*K8s*-Aktualisierungen, -Sicherungen und -Anwendungsbereitstellungen), konfigurierbare *Pod*-Sicherheitsrichtlinien (PSP) und eine RBAC-basierte Benutzerverwaltung angeboten. Als Alternative im *On-Premises*-Vorhaben ist an dieser Stelle *Red Hat OpenShift* zu nennen. *Rancher* lässt sich allerdings leichter und flexibler in die bestehende Infrastruktur integrieren und es Bedarf keinen intensiveren Beschäftigungsaufwand mit dem großen *Red Hat*-Ökosystem. Natürlich könnte die Erstellung und Verwaltung eines Clusters auch nur mit Hilfe der nativen *Kubernetes*-Werkzeuge *kubeadm* und *kubectl* durchgeführt werden [Kub20l]. Dies schafft allerdings in den betrachteten Anwendungsfällen keinen Mehrwert und führt nur noch mehr Komplexität ein. Des Weiteren werden mit den *Ubuntu Cloud Images* optimierte *Cloud Native*-Betriebssysteme im Szenario eingesetzt, die vor allem auch mit *Rancher* harmonieren. Als frei verfügbare *Image Registry* und somit „sichere“ Basis in den verteilten Cluster-Strukturen wird *Harbor* verwendet. Sie unterstützt von vornherein Replikationsansätze, kann *Trusted Images* verwalten und integriert benutzerdefinierte CVE-*Scanner*.

Wie bereits erwähnt, zählen die Sicherheit und Komplexität zu den größten Schwierigkeiten der Container-Virtualisierung im Organisationskontext. Daher werden in diesem Kapitel mögliche Ansatzpunkte bezogen auf die Sicherheit und Automatisierung einer *K8s*-Cluster-Systemlandschaft analysiert. Im Abschnitt 6.1 steht die Informationssicherheit im Vordergrund. Für den Aufbau von sicheren IT-Systemen stellt das BSI ein IT-Grundschutz-Kompendium zur Verfügung, mit dem die zentralen Schutzziele der Informationssicherheit durch definierte Anforderungen gewährleistet werden sollen. Diese Anforderungen werden hier zusammen mit einem abstrakten 4-Schichtenmodell der zugrunde liegenden Architektur thematisiert. Des Weiteren wird auf einige Vorkehrungen bzgl. der angedachten Software-Komponenten (*Rancher*, RKE, *Docker*, *Harbor*, *Ubuntu*) eingegangen. Anschließend folgt im Abschnitt 6.2 die Betrachtung möglicher Automatisierungen im Szenario, um die neu anfallende Komplexität der Technologien im Alltag eines

Rechenzentrums zu verringern. Gegenständlich werden die hier verantwortlichen Bestandteile der *vRealize Suite* für ein *Self-Service*-Portal und der Ausführung automatisierter *Workflows*, *Rancher* für die automatisierte Cluster-Bereitstellung sowie zentrale -Verwaltung und *GitLab* als *DevOps*-Plattform für die Anwendungsbereitstellung noch einmal gesondert angesprochen.

6.1 Sicherheit

Die Informationssicherheit – früher auch als Datensicherheit bezeichnet – hat für Unternehmen und Organisationen einen hohen Stellenwert. Informationen fallen dabei in vielen Bereichen an, sie können sowohl analog auf Papier, digital in Rechnern oder auch geistig in Köpfen vorliegen bzw. gespeichert sein. Zugegeben sind heutzutage die meisten Geschäftsprozesse ohne eine entsprechende IT-Unterstützung nicht mehr denkbar. Essentiell sind eine zuverlässig funktionierende Informationsverarbeitung sowie die dazugehörige Technik für die Aufrechterhaltung der Betriebsprozesse. Werden Informationen unzureichend geschützt, stellt dies oftmals einen unterschätzten Risikofaktor dar, der zumindest eine sinkende Reputation auslösen oder im schlimmsten Fall sogar die Existenz einer Unternehmung bedrohen kann. Sicherheitsmaßnahmen müssen sorgfältig geplant, umgesetzt und kontrolliert werden, um die verarbeiteten Daten und Informationen adäquat schützen zu können. Bei der ganzheitlichen Betrachtung der Informationssicherheit sollten auch die infrastrukturellen, organisatorischen und personellen Rahmenbedingungen herangezogen werden; nicht nur die Sicherheit von IT-Systemen steht hier im Vordergrund. Nicht vernachlässigt werden sollten u. a. die Sicherheit der gesamten Betriebsumgebung, die ausreichende Schulung der Mitarbeiter, die Verlässlichkeit von Dienstleistungen und die richtige Behandlung von zu schützenden Informationen. [BSI20a, vgl. S. 15]

Die Schutzziele oder auch Grundwerte der Informationssicherheit sind Vertraulichkeit, Integrität und Verfügbarkeit. Es können bei der Schutzbedarfsfeststellung natürlich noch weitere Grundwerte wie etwa die Authentizität, Verbindlichkeit, Zuverlässigkeit und Nichtabstreitbarkeit herangezogen werden, wenn dies in einem individuellen Anwendungsfall hilfreich erscheint. Nachfolgend werden die klassischen Schutzziele aufgeführt [BSI20a, vgl. S. 15, 33ff]:

- Verlust der Vertraulichkeit –
  „Vertraulichkeit ist der Schutz vor unbefugter Preisgabe von Informationen. Vertrauliche Daten und Informationen dürfen ausschließlich Befugten in der zulässigen Weise zugänglich sein“ [BSI20a, S. 44]. Gemeint sind hier z. B. personenbezogene Daten von Kunden, oder auch interne und vertrauliche Daten von Unternehmen. Eine Offenlegung dieser Informationen kann schwere Konsequenzen nach sich ziehen.

- Verlust der Integrität –
  „Integrität bezeichnet die Sicherstellung der Korrektheit (Unversehrtheit) von Daten und der korrekten Funktionsweise von Systemen“ [BSI20a, S. 38]. Daten müssen in diesem Zusammenhang vollständig und unverändert vorliegen. Beim weiter gefassten Begriff der Informationen müssen auch die zugehörigen Metadaten unverfälscht bleiben, um die Integrität zu gewährleisten. Durch den Verlust der Authentizität (Echtheit, Überprüfbarkeit) als ergänzendes Schutzziel, könnte in diesem Zusammenhang sogar eine digitale Identität gefälscht werden.

- Verlust der Verfügbarkeit –
  „Die Verfügbarkeit von Dienstleistungen, Funktionen eines IT-Systems, IT-Anwendungen oder IT-Netzen oder auch von Informationen ist vorhanden, wenn diese von den Anwendern

stets wie vorgesehen genutzt werden können“ [BSI20a, S. 44]. Bei einer Störung des Schutzziels wird das Fehlen von grundlegenden Informationen schnell ersichtlich, aber auch die Einschränkung von speziellen Informationen kann nachgeschaltete Prozesse beeinträchtigen.

Wie bereits erwähnt gibt das Bundesamt für Sicherheit in der Informationstechnik (BSI) jährlich ein IT-Grundschutz-Kompendium [BSI20a] heraus, in dem die drei Grundwerte der Informationssicherheit für unterschiedliche Einsatzumgebungen betrachtet werden. Mit dem IT-Grundschutz werden Organisationen praktische Maßnahmen und Sicherheitsanforderungen angeboten, mit denen sie in ihrem Zuständigkeitsbereich Informationen je nach Schutzbedarf angemessen schützen können. Lassen sich Unternehmen zertifizieren – in einigen Bereichen bzw. Branchen ist dies sogar notwendig, dann sollten diese Anforderungen konsequent eingehalten werden.

Mit den Container-Technologien beschäftigt sich das BSI konkret im Baustein „*SYS.1.6: Container*“ [BSI20b], welcher sich derzeit noch in der Entwurfsphase (*Community Draft*, Stand: 19.03.2020) befindet. Hier sind allerdings schon vielversprechende Ansätze zu finden, die u. a. auf dem Sicherheitsleitfaden für Container-Anwendungen [SMS17] des *National Institute of Standards and Technology* (NIST) aufbauen. Die Zielsetzung des Bausteins sieht den Schutz von Informationen im gesamten Lebenszyklus eines Containers und darüber hinaus vor. Er ist nach dem BSI immer anzuwenden, wenn Serverdienste und -anwendungen in Containern betrieben werden. Unabhängig von den verwendeten Produkten werden in diesem Baustein grundsätzliche Anforderungen zur Einrichtung, zum Betrieb und zur Orchestrierung von Containern sowie zur Verwaltung und Bereitstellung von Container-*Images* in einer *Image Registry* aufgestellt; eine Auflistung der Anforderungen ist nach aufsteigendem Schutzbedarf in den Tabellen 6.1, 6.2 und 6.3 vorzufinden. Die Container-Wirte (Hosts) sollten diese Bedingungen erfüllen, unabhängig davon, ob diese selbst auf physischen Servern ausgeführt werden oder virtualisiert vorliegen. Tendenziell müssen für eine umfassendere Betrachtung noch weitere Bausteine des IT-Grundschutz-Kompendiums herangezogen werden, die der Container-Virtualisierung zugrunde liegen (*SYS.1.1: Allgemeiner Server*, *SYS.1.5: Virtualisierung* etc.) oder auf der anderen Seite selbst auf diese Virtualisierungstechnologie in der Form von Container-Anwendungen aufbauen (*APP.3.2: Webserver* etc.).

Die in der Tabelle 6.1 gestellten Basis-Anforderungen müssen nach dem BSI für die Einhaltung des Bausteins „*SYS.1.6 Container*“ vorrangig erfüllt werden.

Tabelle 6.1: Basis-Anforderungen des Bausteins „*SYS.1.6 Container*“ (Stand: 19.03.2020) [BSI20b, vgl. S. 3ff]

Anforderung	**Bezeichnung**
SYS.1.6.A1	Planung des Container-Einsatzes
SYS.1.6.A2	Planung der Separierung der Anwendungen in Containern
SYS.1.6.A3	Planung der Verwaltung und Orchestrierung
SYS.1.6.A4	Härtung des Host-Systems
SYS.1.6.A5	Separierung der Container
SYS.1.6.A6	Verwendung sicherer Images
SYS.1.6.A7	Härtung der Software im Container
SYS.1.6.A8	Persistenz von Protokollierungsdaten
SYS.1.6.A9	Persistenz von Nutzdaten
SYS.1.6.A10	Speicherung von Zugangsdaten

Die Standard-Anforderungen aus der Tabelle 6.2 sollten grundsätzlich nach dem BSI erfüllt werden, um zusammen mit den Basis-Anforderungen den Stand der Technik bezogen auf die Einführung der Container-Virtualisierung in Organisationen zu erfüllen.

Tabelle 6.2: Standard-Anforderungen des Bausteins „*SYS.1.6 Container*“ (Stand: 19.03.2020) [BSI20b, vgl. S. 5ff]

Anforderung	Bezeichnung
SYS.1.6.A11	Richtlinie für Betrieb und *Images*
SYS.1.6.A12	Nur eine Anwendung bzw. ein Dienst pro Container
SYS.1.6.A13	Freigabe von *Images* und Konfigurationen
SYS.1.6.A14	Updates von Containern
SYS.1.6.A15	Unveränderlichkeit der Container
SYS.1.6.A16	Limitierung der Ressourcen pro Container
SYS.1.6.A17	Einbinden von Massenspeichern in Container
SYS.1.6.A18	Absicherung der Wirk- und Administrations-Netze
SYS.1.6.A19	Verwendung vorgelagerter Ein- und Ausgangssysteme
SYS.1.6.A20	Absicherung von Konfigurationsdaten und Automatisierung
SYS.1.6.A21	Container-Ausführung ohne Privilegien
SYS.1.6.A22	Absicherung von Hilfsprozessen der Automatisierung
SYS.1.6.A23	Administrativer Fernzugriff auf Container
SYS.1.6.A24	Identitäts- und Berechtigungsmanagement für die Container-Verwaltung
SYS.1.6.A25	Service-Accounts für Container
SYS.1.6.A26	Accounts der Anwendungsdienste in Containern
SYS.1.6.A27	Überwachung der Container
SYS.1.6.A28	Absicherung der *Registry* für *Images*

Die Tabelle 6.3 enthält Vorschläge für erhöhte Anforderungen, die über das dem Stand der Technik entsprechenden Schutzniveau hinausgehen und daraufhin in Betracht gezogen werden sollten. Im Rahmen einer Risikoanalyse würde eine konkrete Festlegung erfolgen.

Tabelle 6.3: Erhöhte Anforderungen des Bausteins „*SYS.1.6 Container*“ (Stand: 19.03.2020) [BSI20b, vgl. S. 7f.]

Anforderung	Bezeichnung
SYS.1.6.A29	Automatisierte Auditierung von Containern
SYS.1.6.A30	Eigene *Trusted Registry* für Container
SYS.1.6.A31	Erstellung erweiterter Richtlinien für Container
SYS.1.6.A32	*Host Based Intrusion Detection* für Container
SYS.1.6.A33	Mikro-Segmentierung von Containern
SYS.1.6.A34	Hochverfügbarkeit von Containern
SYS.1.6.A35	Verschlüsselte Datenhaltung bei Containern
SYS.1.6.A36	Verschlüsselung der Netzkommunikation zwischen Containern

Nach dem BSI sind die folgenden spezifischen Bedrohungen und Schwachstellen im Bereich der Container-Technologien von besonderer Bedeutung, die durch die aufgestellten Anforderungen des Bausteins „*SYS.1.6 Container*“ vermieden werden sollen [BSI20b, vgl. S. 2f.]:

- Schwachstellen in *Images* –
  Container basieren auf vorgefertigten *Images*, welche entweder aus dem öffentlichen Internet stammen oder selbst erstellt werden. Diese *Images* enthalten die zu betreibende Software oder sie wird durch das Hinzufügen einer weiteren Schicht ergänzt. Die enthaltene Software in einem solchen *Image* könnte verwundbar sein und somit Server-Dienste angreifbar machen. Oftmals passiert dies in Unkenntnis, da die Container-Abbilder und deren Schwachstellen

nur mit Hilfe zusätzlicher Werkzeuge erfasst werden können.

- Administrative Zugänge ohne Absicherung –
  Administratoren und die Tool-gestützte Orchestrierung benötigen für die Verwaltung von Container-Clustern entsprechende Zugänge, die gewöhnlich über ein IKT-Netzwerk erfolgen. Obwohl Methoden zur Authentisierung und Verschlüsselung dieser administrativen Zugänge zumeist existieren, sind sie nicht bei allen Produkten standardmäßig aktiviert. Unbefugte hätten demzufolge die Möglichkeit über ungeschützte Administrationszugänge Befehle auszuführen, die die Schutzziele der Informationssicherheit gefährden.

- Tool-basierte Orchestrierung ohne Absicherung –
  Für den Betrieb einer größeren Anzahl von Containern, wird zumeist zur Orchestrierung und Verwaltung dieser eine Software eingesetzt. Diese Komponente kann selbst über Schwachstellen verfügen oder nicht ausreichend gegen Unbefugte geschützt sein. Angreifer könnten auf diese Weise die Container-Hosts mit administrativen Berechtigungen kompromittieren und die drei erwähnten Grundwerte attackieren.

- Ausbruch aus dem Container –
  Angreifer könnten in der Lage sein, aus einem Container auszubrechen und die Kontrolle über den Container-Host oder andere Server im Netzwerk übernehmen, wenn für sie die Möglichkeit besteht, eigenen Code in einem Container auszuführen. Durch die gezielte Ausnutzung von Schwachstellen bspw. in CPUs, im Kernel oder in lokalen Diensten wie DNS (*Domain Name System*) oder SSH (*Secure Shell*) könnte dies geschehen und schwere Auswirkungen nach sich ziehen.

- Datenverluste durch fehlende Persistenz –
  Vom Aufbau her werden Container nur für eine temporäre Ausführungszeit erzeugt, zudem können sie jederzeit von den Verwaltungswerkzeugen abgeschaltet werden. Findet dies keine Beachtung, dann könnte eine Container-Anwendung ausschließlich Daten im Container speichern und im Zuge einer Neuinstantiierung, z. B. durch eine Aktualisierung des *Images*, all diese Daten unwiderruflich verlieren. Betrachtet werden sollten auch jene Dateien, die in der Verarbeitung nur zeitlich begrenzt im Container vorliegen und deren Ergebnisse bei Verlust unter Umständen nicht mehr nachvollzogen werden können.

- Vertraulichkeitsverlust von Zugangsdaten –
  Für den Aufbau und die Erstellung von Container-*Images* ist es oftmals notwendig, im Container über Zugangsdaten z. B. für Datenbanken zu verfügen. Diese sensiblen Informationen können dann ungeschützt im *Image*, in den Skripten zur Erstellung der Abbilder oder auch in der Versionsverwaltung dieser Skripte liegen und im schlimmsten Fall in unautorisierte Hände gelangen.

Ein bildlicher Ausgangspunkt für die systematische Sicherheitsbetrachtung eines *Kubernetes*-Clusters können die „*The 4C's of Cloud Native Security*“ sein, die in der Abbildung 6.1 dargestellt sind. Das Diagramm vermittelt einen mehrschichtigen Ansatz für die Sicherung von Software-Systemen in einer *Cloud Native*-Architektur. Die einzelnen Schichten sind hierbei: *Cloud / Co-Lo / Corporate Datacenter*, *Cluster*, *Container* und *Code*. Wie zu erkennen ist, besteht zwischen den ineinandergreifenden Schichten eine Abhängigkeit. Es wird bspw. schwierig, sich gegen geringe Sicherheitsmaßnahmen in den oberen Schichten zu schützen, wenn nur die Sicherheit auf der *Code*-Ebene betrachtet und umgesetzt wird. Werden hingegen diese zuvor nicht ausreichend betrachteten Schichten mit guten Sicherheitsstandards und *Best Practices* versehen, erweitert die Sicherheit im *Code* eine bereits starke Basis. Das abstrakte Modell versinnbildlicht das „*Defense in Depth*“-Konzept zur Sicherung von Softwaresystemen, welches absichtlich Redundanzen auf

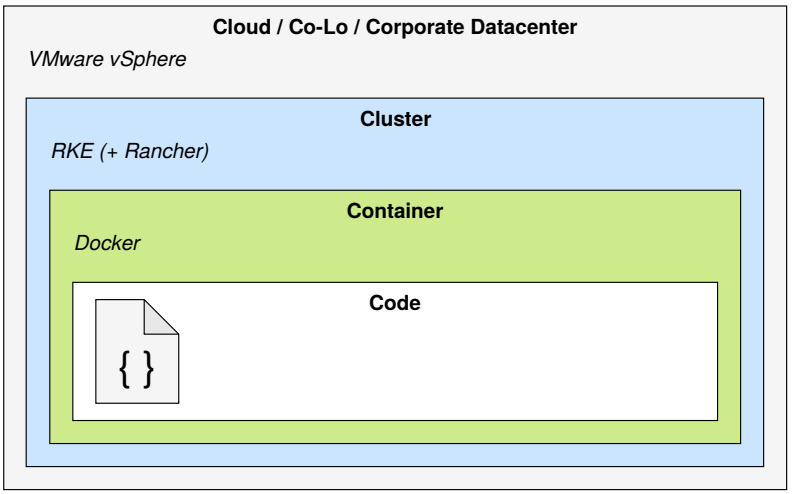


Abbildung 6.1: *The 4C's of Cloud Native Security* [Kub20g]

verschiedenen Ebenen erschafft, um dem Ausfall von Sicherheitskontrollen oder dem Ausnutzen von Schwachstellen entgegenzuwirken. [Kub20g]

Die Problemstellen und Maßnahmen für den Einsatz eines *K8s*-Clusters, die in den einzelnen Schichten der „*The 4C's of Cloud Native Security*“ anfallen, werden nachfolgend in den Unterabschnitten aufgezeigt. Begleitend dazu kann das *Kubernetes Security Whitepaper* [ECTP19] von *Trail of Bits* herangezogen werden, das Schlüsselaspekte der *K8s*-Angriffsvektoren und der Sicherheitsarchitektur aufzeigt. Im *Whitepaper* wird dabei zwischen externen Angreifern, die keinen Zugang zum Cluster haben, internen Angreifern, die bereits auf einen *Pod* / Container zugreifen können und böswilligen internen Benutzern differenziert, die ihre Privilegien innerhalb des Clusters missbrauchen. Darauf aufbauend werden Empfehlungen für Cluster-Administratoren und Anwendungsentwickler zur Verfügung gestellt.

6.1.1 *Cloud / Co-Lo / Corporate Datacenter*

Die erste Schicht des Modells ist die vertrauenswürdige Basis, auf die ein *Kubernetes*-Cluster basiert. Sind die hier involvierten Komponenten der Cloud, *Co-Located Servers* oder etwa eines eigenen Rechenzentrums (*Corporate Datacenter*) anfällig oder falsch konfiguriert, dann kann die Sicherheit der darauf aufbauenden Komponenten nicht gewährleistet werden. Für die eingesetzte *VMware*-Virtualisierungsumgebung gibt es direkt vom Hersteller verschiedene Sicherheitsempfehlungen und Leitfäden zur Sicherheitshärtung [VMw20a], die in einem Audit und durch entsprechende Protokollierungen behandelt werden können. Die dazugehörigen *Firewall*-Einstellungen für die *vSphere*-Umgebung und die notwendigen Aktionen für eine *ESXi*-Host-Absicherung können zudem dem Standardwerk [WAG+18, vgl. S. 756] entnommen werden. Auch Mikrosegmentierungen innerhalb eines SDDCs werden hier angesprochen, die auf die Netzwerkvirtualisierung via NSX aufbauen [WAG+18, vgl. S. 477ff]. *Firewalls* können auf dieser Ebene durch sogenannte *Intrusion Detection Systems* (IDS) ergänzt werden und so Angriffe auf IT-Systeme und IKT-Netzwerke erkennen. Um Attacken darüber hinaus automatisiert und aktiv abzuwehren bieten sich IPS-Produkte (*Intrusion Prevention System*) an. Diese Maßnahmen und Komponenten sollen an dieser Stelle allerdings nicht weiter betrachtet werden, da von einer bereits abgesicherten Umgebung ausgegangen wird. Dennoch lassen sich allgemeine Problembereiche bezogen auf die Sicherheit einer zugrunde liegenden *K8s*-Infrastruktur auflisten [Kub20g]:

- Netzwerkzugriff auf den API-Server (*Control Plane*) –
  Der Zugriff auf die Steuerungsebene eines *Kubernetes*-Clusters sollte nicht öffentlich im Internet möglich und über Netzwerk-Zugriffskontrolllisten bzw. entsprechende *Firewalls* beschränkt sein. Nur die für die Verwaltung eines Clusters erforderliche Menge an IP-Adressen sollte freigegeben werden.

- Netzwerkzugriff auf die Knoten (*Nodes*) –
  Auch die einzelnen Knoten, auf denen letztendlich die Container-Anwendungen bereitgestellt werden, sollten wenn möglich nicht vollständig im öffentlichen Internet stehen. Nur geregelte Verbindungen sollten über Netzwerk-Zugriffskontrolllisten bzw. *Firewalls* von der Steuerungsebene aus an die angegebenen Ports akzeptiert werden. Die externen Zugriffe auf Anwendungen in einem *K8s*-Cluster sollten dennoch vom Typ „*NodePort*“ und „*LoadBalancer*“ gewährleistet sein.

- Zugang zum Schlüsselwertspeicher (*etcd*) –
  Die Verbindungen auf den Datenspeicher eines *Kubernetes*-Clusters sollten im besten Fall nur von der Steuerungsebene aus möglich sein. Außerdem sollte *etcd* über TLS (*Transport Layer Security*) – sprich nur über einen verschlüsselten Kommunikationskanal – angesprochen werden.

- Verschlüsselung des Schlüsselwertspeichers (*etcd*) –
  Bezogen auf die Informationssicherheit ist es in der Praxis durchaus sinnvoll, Laufwerke zu verschlüsseln, die sich im Ruhezustand befinden. Der Schlüsselwertspeicher (*Key–Value Store*) *etcd* hält den Zustand des gesamten Clusters einschließlich der *Secrets* vor, daher sollte gerade das Laufwerk dieser Komponente im Ruhezustand verschlüsselt und besonders vertraulich behandelt werden.

Im Allgemeinen sollte der Zugriff auf die Bestandteile eines Software-Systems, wie *Kubernetes* eines ist, möglichst nach dem „*Principle of Least Privilege*“ (PoLP) und in fein granulierten Abstraktionsschichten erfolgen. Das Prinzip verwirklicht den Zugriff einer Partei nur auf die Informationen und Ressourcen, für die diese einen legitimen Zweck vorweisen kann. Beschränkungen können hierfür direkt mit Hilfe von *Firewalls* auf den jeweiligen Hosts des Clusters konfiguriert werden. Des Weiteren gibt es die Möglichkeit *Firewalls* in Netzsegmenten zu integrieren, die nicht nur den Zugriff auf einen bestimmten Host, sondern auch Verbindungen zu kompletten Netzwerken blockieren bzw. zulassen können.

6.1.2 *Cluster*

In der *Cluster*-Schicht sind zwei Bereiche für die Sicherheit eines *Kubernetes*-Clusters von entscheidender Bedeutung. Dies ist zum einen die Sicherung der eigentlichen Komponenten vor versehentlichem oder böswilligem Zugriff, aus denen ein *K8s*-Cluster besteht [Kub20j] ...

- Steuerung des Zugriffs auf die *Kubernetes*-API –
  K8s ist vollständig API-gesteuert, daher sollten der Zugriff und die davon abhängigen Aktionen der API mit den folgenden Maßnahmen kontrolliert bzw. eingeschränkt werden:

  – Verwendung von TLS im gesamten API-Verkehr: Standardmäßig wird die gesamte API-Kommunikation in einem *K8s*-Cluster mit Hilfe von TLS verschlüsselt; *Kubernetes* setzt dies sogar voraus. Viele Installationsmethoden ermöglichen bereits die Erstellung und Verteilung der erforderlichen Zertifikate an die jeweiligen Cluster-Komponenten, dennoch kann es bei einigen Methoden vorkommen, dass lokale Ports noch über HTTP angeboten werden. Der ungesicherte Datenverkehr muss dann umgestellt werden.

  - API-Authentifizierung: Im Cluster muss ein Mechanismus zur Authentifizierung ausgewählt werden, der dem üblichen Zugriffsmuster der beteiligten Akteure entspricht. Bei kleinen Einzelbenutzer-Clustern kann dies vereinfacht durch ein Zertifikat oder einen statischen *Bearer*-Token geschehen. Bei größeren Clustern bietet sich bspw. die Integration eines LDAP-Verzeichnisdienstes (*Lightweight Directory Access Protocol*) an, der die Unterteilung der Nutzer in Gruppen ermöglicht. Generell müssen jedoch alle API-Clients authentifiziert werden, auch die, die der Infrastruktur (*Nodes*, *Proxies*, *Scheduler* etc.) angehören. Gewöhnlich werden hierfür beim Cluster-Start oder bei der -Installation automatisch Dienstkonten oder *x509*-Client-Zertifikate erstellt.

  - API-Autorisierung: Jeder API-Aufruf muss nach der Authentifizierung auch die Autorisierung bestehen. Hierfür verwendet *Kubernetes* eine integrierte rollen-basierte Zugriffskontrolle (RBAC), die eingehende Benutzer oder auch Gruppen einem Satz von Berechtigungen zuordnet, die wiederum in Rollen gebündelt vorliegen. Diese Berechtigungen können auf einem *Namespace* fixiert oder auch Cluster-übergreifend Anwendung finden und kombinieren im Wesentlichen Aktionen (*get*, *create*, *delete*) mit Ressourcen (*Pods*, *Services*, *Nodes* etc.). Die dabei als Organisationseinheit verwendeten Rollen können speziell auf einen Anwendungsfall zugeschnitten werden, oder es wird sich mit den bereits vorhandenen „*Out of the box*“-Rollen arrangiert. Generell sollte eine Rolle sorgfältig geprüft werden, um eine versehentliche Eskalation durch zu weitreichende Berechtigungen zu vermeiden.

- Kontrolle des *kubelet*-Zugangs –
  In einem *K8s*-Cluster gewährt der *kubelet*-Agent durch die Bereitstellung eines HTTPS-Endpunktes eine mächtige Kontrolle über den jeweiligen Knoten und dessen Container. Standardmäßig wird der nicht authentifizierte Zugriff auf diesen Endpunkt erlaubt, deshalb sollten Produktiv-Cluster die Authentifizierung und Autorisierung an der *kubelet*-Komponente aktiviert haben und auch nur diesen gesicherten Weg erlauben.

- Kontrolle der Berechtigungen eines *Workloads* oder Benutzers zur Laufzeit –
  Die Autorisierung in *Kubernetes* findet auf einer hohen Abstraktionsschicht statt und spezifiziert grobe Aktionen an den involvierten Ressourcen. Mächtigere Kontrollen bieten die nachfolgend aufgelisteten Richtlinien (*Policies*) an, die *K8s*-Objekte im Cluster-Verbund je nach Anwendungsfall zusätzlich einschränken:

  - Begrenzung der Cluster-Ressourcennutzung: Mit Hilfe eines Ressourcenkontingents (*ResourceQuota*) kann die Kapazität der in einem *Namespace* zur Verfügung stehenden Hardware-Ressourcen und die Anzahl von *Pods* und *Services* begrenzt werden. Des Weiteren kann ein Grenzwertbereich (*LimitRange*) definiert werden, der maximale und / oder minimale Werte an Host-Ressourcen (CPU, RAM, *Storage* etc.) in einem *Namespace* vorgibt, um zu verhindern, dass Benutzer unangemessen hohe oder niedrige Kapazitäten anfordern. Außerdem ist hierüber die Angabe von Standardwerten möglich.

  - Kontrolle, mit welchen Privilegien Container laufen: *Pod*-Definitionen können einen Sicherheitskontext (*securityContext*) enthalten, welcher Berechtigungs- und Zugriffssteuerungseinstellungen definiert. Generell ist es möglich Container auf der Host-Ebene mit der Berechtigung des *Root*-Benutzer auszuführen. In Anbetracht der damit verbundenen Privilegien und des möglichen Ausbrechens aus einem Container-Kontext, sollten Container grundsätzlich mit einem Nicht-*Root*-Benutzer ausgeführt werden. Restriktive *Pod*-Sicherheitsrichtlinien (PSPs) sollten diesbezüglich auf globaler Cluster-Ebene eine Verwendung finden.

  - Verhindern des Ladens unerwünschter Kernel-Module durch Container: Unprivilegierte Prozesse können bestimmte Kernel-Module bezogen auf das Netzwerkprotokoll vom Host-System automatisch nachladen lassen und daraufhin mögliche Sicherheitslücken ausnutzen. Um dies zu verhindern, können die Module vom Knoten entfernt oder lokale Regeln im Betriebssystem aufgestellt werden, die das Laden blockieren. Generischer geht es mit Hilfe der LSMs (*Linux Security Modules*), die Containern die Berechtigung für einen solchen „*module_request*“ vollständig entziehen können; der Kernel kann dann auf diesem Host keine Module mehr für die Container laden. Manuell geladene Module auf dem Host und privilegierte Container sind hiervon jedoch ausgenommen.
  - Beschränkung des Netzzugangs: Innerhalb eines *Namespace* können mit Netzwerkrichtlinien (*NetworkPolicy*) der Zugriff von *Pods* auf *Pods* und Ports in anderen *Namespaces* geregelt werden. Durch die Verwendung der *Service*-Ressource vom Typ „*NodePort*“ oder auch „*LoadBalancer*“ kann in diesem Zusammenhang gesteuert werden, ob eine bestimmte Anwendung auch außerhalb des Clusters sichtbar sein soll. Zusätzliche Schutzvorkehrungen bezogen auf den Netzzugang können z. B. über *Firewalls* auf den *Nodes* oder durch physisch getrennte Cluster-Knoten erreicht werden.
  - Kontrolle, auf welche Knoten *Pods* einen Zugang haben: Standardmäßig gibt es unter *Kubernetes* keine Einschränkungen, welche Knoten einen *Pod* ausführen dürfen. Dennoch gibt es die Möglichkeit über Richtlinien eine kontrollierte Platzierung von *Pods* auf Knoten vorzunehmen. Darüber hinaus gibt es das Konzept der *Taints* und *Tolerations*. Ein oder mehrere *Taints* (dt. Anstriche) werden auf einen Knoten aufgetragen. Dies hat zur Folge, dass der Knoten grundsätzlich keine *Pods* annehmen sollte, die die *Taints* nicht tolerieren.

- Schutz der Cluster-Komponenten vor Kompromittierung –
  Einige verbreitete Schutzvorkehrungen, die die Komponenten eines *K8s*-Clusters betreffen, werden nachstehend aufgelistet:
  - Zugang zu *etcd*-Instanzen einschränken: Ein Schreibzugriff auf das *etcd-Backend* mit der API ist äquivalent mit dem Erlangen von *Root*-Berechtigungen auf dem gesamten Cluster. Auch ein entsprechender Lesezugriff auf die sensiblen Informationen kann schnell zu einer Eskalation führen. Von den API-Servern zu den *etcd*-Instanzen sollten daher immer starke Berechtigungsnachweise, wie z. B. gegenseitige Authentisierung über TLS-Client-Zertifikate, verwendet werden. Außerdem sollte der verteilte Schlüsselwertspeicher möglichst hinter einer *Firewall* abgegrenzt werden, auf die nur die API-Server einen Zugriff erlangen.
  - Aktivieren der Audit-Protokollierung: Im Falle einer Kompromittierung des Clusters können die von der API durchgeführten Aktionen für eine spätere Analyse durch einen sogenannten Audit-*Logger* aufgezeichnet werden. Obwohl dieser noch als Beta-Funktion in *Kubernetes* deklariert ist, wird empfohlen die Protokollierung zu aktivieren und die Audit-Datei auf einen sicheren Server zu archivieren. Generell ist die Überwachung der einzelnen *K8s*-Komponenten über eine geeignete Protokollierung empfehlenswert.
  - Zugang zu Alpha- oder Beta-Funktionen einschränken: Die Verwendung von *Kubernetes*-Funktionen, die sich in einem Alpha- und Beta-Status befinden, kann aufgrund von Einschränkungen oder Fehlern zu Sicherheitslücken im Betrieb führen. Zuvor sollte immer der Mehrwert verbunden mit einem einhergehenden Sicherheitsrisiko abgeschätzt werden; im Zweifelsfall sollten derartige Funktionen deaktiviert bleiben.
  - Häufiges Rotieren der Infrastruktur-Zugangsdaten: Je kürze die Laufzeit von Berechtigungsnachweisen ist, desto schwieriger wird es für Eindringlinge diesen Angriffsvektor

auszunutzen. Zertifikate in der Umgebung sollten kurze Laufzeiten besitzen und eine automatisierte Rotation verwenden. Auch die ausgegebenen Tokens sollten im besten Fall nur kurzfristig bestehen und möglichst über einen Authentifizierungsanbieter kontrolliert sowie regelmäßig erneuert werden.

- Überprüfung der Integration von *3rd Parties*: Die Integration von Drittanbietern in *Kubernetes* kann ungeahnte Auswirkungen auf das Sicherheitsprofil eines Clusters haben. Bevor einer solchen Erweiterung der Zugriff gewährt wird, sollten immer die angeforderten Berechtigungen überprüft werden. Im Zweifelsfall sollte die Integration auf das Funktionieren in einem einzigen *Namespace* beschränkt werden. Berechtigungen für den „*kube-system*"-*Namespace* oder die Verwendung von freizügigen PSPs müssen gesondert betrachtet werden.
- *Secrets* im Ruhezustand verschlüsseln: Der Schlüsselwertspeicher *etcd* enthält alle relevanten Informationen, die über die *K8s*-API zugänglich sind. Angreifer könnten hierüber sensible Einblicke in den Zustand eines Clusters erlangen. Erstellte Sicherungen dieser Datenbank sollten grundsätzlich verschlüsselt werden, wenn möglich ist eine vollständige Festplattenverschlüsselung zu bevorzugen. Obwohl *Kubernetes* die Verschlüsselung im Ruhezustand momentan nur als Beta-Funktion unterstützt, bietet sie eine zusätzliche Verteidigungsebene an, falls die Sicherungen nicht verschlüsselt sind oder ein Angreifer Lesezugriff auf *etcd*-Instanzen erhält.

... und zum anderen das Sichern der Container-Anwendungen (*Workloads*), die letztendlich im Cluster ausgeführt werden. Diesbezüglich werden die Sicherheitsbedenken und relevante Themen nachfolgend aufgelistet [Kub20g]:

- RBAC-Autorisierung (Zugriff auf die *K8s*-API) –
  Unter der rollen-basierten Zugriffskontrolle (RBAC) wird eine Methode zur Regulierung des Zugriffs auf Computer- oder Netzwerkressourcen auf der Grundlage einer Rolle des einzelnen Benutzers innerhalb einer Organisation verstanden. Die RBAC-Autorisierung verwendet dabei konkret die API-Gruppe „*rbac.authorization.k8s.io*", um Autorisierungsentscheidungen zu treffen. Über die *Kubernetes*-API lassen sich entsprechende Richtlinien für den Zugriff auf *Workloads* dynamisch konfigurieren, falls die standardmäßigen RBAC-Richtlinien nicht ausreichen.
- Authentifizierung –
  Auf die API eines *Kubernetes*-Clusters kann mit Hilfe von Client-Bibliotheken, REST-Anfragen oder auch der *kubectl* zugegriffen werden. Für einen autorisierten API-Zugriff können sowohl menschliche Benutzer als auch *K8s*-Service-Konten involviert sein. Erreicht eine Anfrage schließlich die API, dann müssen vor der Bearbeitung der Anfrage im Cluster drei Stufen nacheinander überwunden werden: Authentifizierung, Autorisierung und die Zugriffskontrolle. Auf allen Stufen muss die jeweilige Anfrage akzeptiert werden, ansonsten wird sie verworfen und Interaktionen mit den Cluster-Ressourcen bleiben verwehrt.
- *Secrets*-Verwaltung von Anwendungen (*etcd*-Verschlüsselung im Ruhezustand) –
  Mit den *Secret*-Objekten in *Kubernetes* lassen sich vertrauliche Informationen wie bspw. Passwörter, *OAuth*-Token oder auch SSH-Schlüssel im verteilten Schlüsselwertspeicher *etcd* speichern und verwalten. Die Verwendung von *Secrets* für sensible Daten ist sicherer und flexibler als das Ablegen der Informationen in Klartext innerhalb eines *Pod*-Manifests oder eines Container-*Images*. Wie bereits angesprochen ist es – gerade bei der *Secrets*-Verwendung – von großer Wichtigkeit, die *etcd*-Datenbank im Ruhezustand zu verschlüsseln.
- *Pod*-Sicherheitsrichtlinien (PSPs) –
  Bezogen auf die *Pod*-Sicherheit gibt es in *Kubernetes* traditionell das Definieren eines

Sicherheitskontextes (*securityContext*), welcher *Pods* und Container zur Laufzeit konfiguriert und dementsprechend Teil eines *Pod*-Manifests ist. Ein solcher Sicherheitskontext wird individuell für einen *Pod* angelegt und ermöglicht daher keine globale Durchsetzung konsistenter Sicherheitsrichtlinien. PSPs hingegen ermöglichen eine feingranulare Autorisierung bezogen auf eine *Pod*-Erstellung und -Aktualisierung auf der Ebene der *Control Plane* [Kub20i]. Diese Richtlinien kontrollieren sicherheitsrelevante Aspekte der *Pod*-Spezifikationen und geben eine Reihe von Bedingungen an, unter denen ein *Pod* im Cluster letztendlich ausgeführt werden muss. Standardmäßig gibt es drei vordefinierte *Policy Types*, die das Sicherheitsspektrum von stark eingeschränkt bis hochflexibel abdecken:

  - *Privileged* (dt. privilegiert): „Uneingeschränkte Richtlinie, die ein größtmögliches Maß an Berechtigungen unterstützt. Diese Richtlinie lässt bekannte Privilegieneskalationen zu“ [Kub20i, Übers.].
  - *Baseline / Default* (dt. Basislinie / Standard): „Minimal restriktive Richtlinie bei gleichzeitiger Verhinderung bekannter Privilegieneskalationen. Erlaubt die standardmäßige (minimal festgelegte) Pod-Konfiguration“ [Kub20i, Übers.].
  - *Restricted* (dt. eingeschränkt): „Stark eingeschränkte Richtlinie, die den aktuellen Best Practices für die Pod-Härtung folgt“ [Kub20i, Übers.].

- *Quality of Service* (QoS) und Cluster-Ressourcenmanagement –
  Pods können sogenannte QoS-Klassen zugewiesen werden. *Kubernetes* selbst verwendet diese Eigenschaften im Cluster-Verbund, um Entscheidungen über das *Scheduling* und Entfernen von *Pods* zu treffen. Einem *Pod* wird bei der Erstellung in Abhängigkeit der Ressourcenzuweisung eine der folgenden QoS-Klassen automatisch zugewiesen:
  - *Guaranteed* (dt. garantiert): Jeder im *Pod* enthaltene Container muss eine Speicher- und CPU-Begrenzung (*limits*) sowie eine Speicher- und CPU-Anforderung (*requests*) besitzen, die zudem identisch sein müssen.
  - *Burstable* (dt. aufbrechbar): Der *Pod* erfüllt die Kriterien der QoS-Klasse „*Guaranteed*“ nicht. Mindestens ein Container im *Pod* besitzt allerdings eine Speicher- oder CPU-Begrenzung (*limits*) bzw. -Anforderung (*requests*).
  - *BestEffort* (dt. größte Bemühung): Die Container innerhalb eines *Pods* dürfen keine Speicher- oder CPU-Begrenzungen bzw. -Anforderungen besitzen.

- Netzwerkrichtlinien –
  Richtlinien für das Netzwerk spezifizieren, wie Gruppen von *Pods* miteinander und mit anderen Netzwerk-Endpunkten kommunizieren dürfen. Das *K8s*-Objekt „*NetworkPolicy*“ verwendet dazu *Labels*, um *Pods* auszuwählen und daraufhin Regeln durchzusetzen, welcher Datenverkehr zu den betroffenen *Pods* zulässig ist. Sobald in einem *Namespace* ein „*NetworkPolicy*“-Objekt existiert und einen *Pod* auswählt, lehnt dieser alle Verbindungen ab, die nicht explizit von der Richtlinie zugelassen werden. Netzwerkrichtlinien sind zueinander additiv, d. h. wenn mehrere Richtlinien einen *Pod* auswählen, dann wird die Vereinigung des Regelsatzes aktiv – die Reihenfolge hat dementsprechend keinen weiteren Einfluss. Standardmäßig liegen *Pods* nicht isoliert vor, sie akzeptieren also Netzwerkzugriffe von jeder Quelle; diese zugrunde liegende Standardrichtlinie ist anpassbar.

- TLS für *Kubernetes-Ingress* –
  Ingress ist eine *K8s*-Ressource, die HTTP- und HTTPS-Routen von außerhalb des Clusters auf intern-befindliche Dienste ermöglicht. Darüber hinaus kann diese Ressource so konfiguriert werden, dass Anwendungen extern erreichbare URLs zugeteilt, eine Lastverteilung des Datenverkehrs, die Terminierung von TLS und namensbasiertes virtuelles *Hosting* betrieben

werden kann. Um auch andere Ports bzw. Protokolle als HTTP und HTTPS für einen externen Zugriff freizugeben, müssen unweigerlich *K8s-Services* vom Typ „*NodePort*“ bzw. „*LoadBalancer*“ verwendet werden. Generell gilt auch hier, dass eine Verschlüsselung des Transportkanals zur Sicherung der Informationssicherheit erfolgen sollte.

Durch den beabsichtigten Einsatz von *Rancher* mit RKE als zentrale *Kubernetes*-Verwaltungsplattform in einer *On-Premises*-Umgebung werden bereits einige der hier aufgeführten sicherheitskritischen Bestandteile gelöst [Ran20g, Ran20e]. Als Basis agieren Cluster der *K8s*-Distribution RKE, welche komplett auf die *Docker*-Virtualisierung im Gegensatz zu einer *Bare Metal*-Variante aufbauen. Daher treffen nicht alle – aber die meisten – der hier thematisierten Angriffsvektoren eines „*Vanilla Kubernetes*“ zu. Mit Hilfe von zuvor erstellten und gehärteten *Templates* lassen sich durch *Rancher* diesbezüglich „sichere“ Cluster- und Host-Konfigurationen automatisiert erzeugen. Zudem gibt es für die zentrale *Cluster Management*-Plattform die Möglichkeit eine hochverfügbare Architektur (HA) zu konstruieren, um das Schutzziel der Verfügbarkeit zu garantieren. Interessant ist in diesem Zusammenhang auch der Ansatz einer sogenannten „*Air Gapped Environment*“, die verwaltete RKE-Cluster in abgeschotteten Netzwerksegmenten samt eigener *Private Registry* thematisiert; dies wird noch einmal im Kapitel 7 aufgegriffen. In dem im Jahr 2019 durch *Cure53* durchgeführten Penetrationstest [HKH+19] hat *Rancher* die erfolgten Angriffe und Kompromittierungsversuche gut überstanden. Begründet durch den relativ großen und komplexen Aufbau war die Plattform zu diesem Zeitpunkt nicht frei von Schwachstellen. Die Untersuchung zeigte allerdings durch die enge Zusammenarbeit der Entwickler mit den Testern, dass *Rancher Labs* eine hohe interne Motivation zur Erreichung aller Sicherheitsziele besitzt und Software-Schwachstellen schnellstmöglich schließt.

6.1.3 *Container*

Erst durch die Einführung der *Namespaces*, die der *Linux*-Kernel seit der Version *2.6.32* mitbringt, sind die heutigen Container-Implementierungen möglich geworden [Lie19, vgl. S. 80ff]. Ressourcen des Kernsystems (Kernels) können hiermit voneinander isoliert und unter bestimmten Voraussetzungen anderen Prozessen zur Verfügung gestellt werden. „*Namespaces* limitieren, was (pid, net, ipc, mount, uts, user) von welchen Prozessen gesehen werden kann bzw. darf (*Isolation*)“ [Lie19, S. 86]. Im Kontext dieser Analyse muss grundsätzlich folgende Aussage getroffen werden: „Container – tolle Sache, aber die Sicherheit von Containern ist nach wie vor ein zentrales Problem“ [Lie19, S. 84]. Eine entscheidende Rolle spielen dabei zwei Aspekte. Zum einen die Sicherheit und Vertrauenswürdigkeit der *Images*, die den auszuführenden Containern als Basis dienen und zum anderen die bereits benannten *Namespaces*. Letztere isolieren zwar die direkten Zugriffe auf Kernel-Ressourcen, dennoch sind die Container-Prozesse schon mal auf der niedrigen Ebene des Kernels verankert.

Einen besseren Überblick zu dieser Problematik verschafft die Abbildung 6.2, in der der unterschiedliche Zugriff auf die CPU in VMs und Containern gegenübergestellt wird. Auf dem rechts dargestellten Host, der zwei VMs ausführt, teilen sich wiederum zwei komplett getrennte Betriebssysteme die Hardware. Unterhalb der VMs ist ein *Hypervisor* platziert, der letztendlich die tatsächlichen Hardware-Ressourcen auf virtuelle Ressourcen geringeren Umfangs abbildet. Systemaufrufe werden innerhalb der VMs an den Kernel des Gastbetriebssystems gerichtet. Genau dieser Kernel führt anschließend Anweisungen über den *Hypervisor* auf der physikalischen CPU des Hosts aus. Bei Containern hingegen ist diese strenge Kapselung, wie auf der linken Seite zu sehen, nicht gegeben. Hier werden Systemaufrufe direkt in dem Kernel ausgeführt, der schließlich auch die Anweisungen auf der CPU des Hosts bewirkt. Die CPU braucht für die Container-Bereitstellung daher auch keine Erweiterung der Prozessor-Virtualisierung. [Luk18, vgl. S. 11f.]

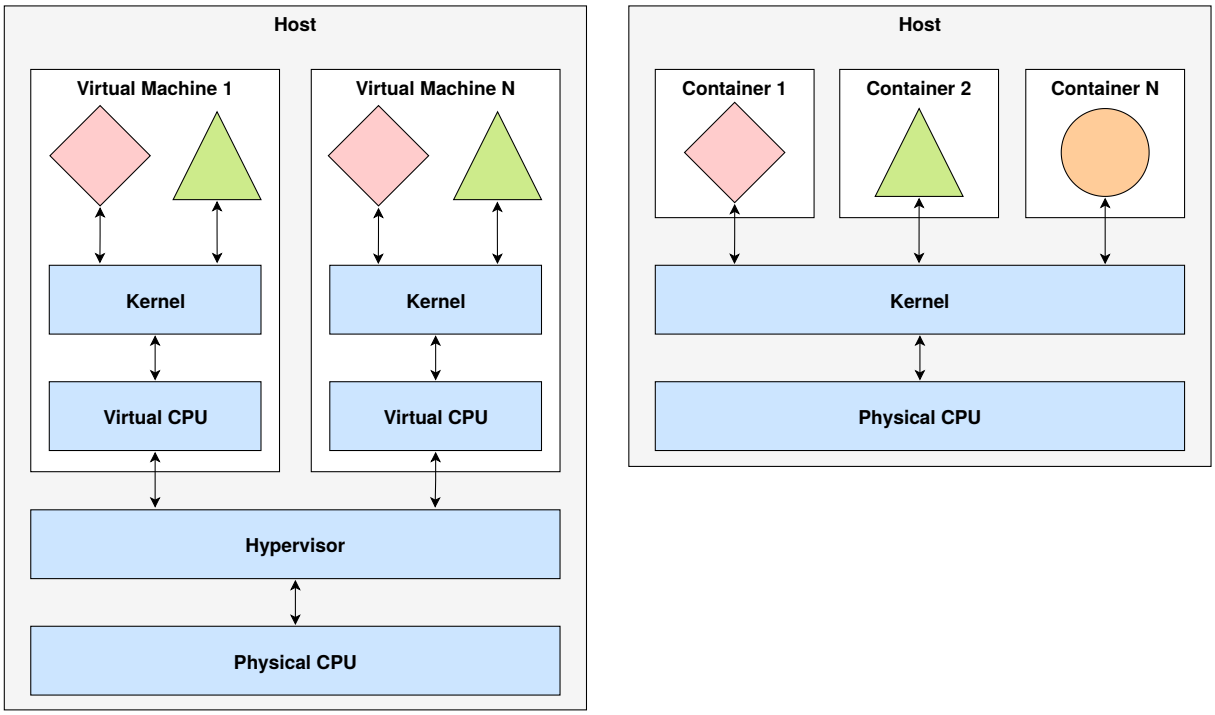


Abbildung 6.2: Unterschiedlicher Zugriff auf die CPU in VMs und Containern [Luk18, vgl. S. 12]

Mit unter verlangen Container-Anwendungen erhöhte Berechtigungen auf dem Host-System. Es können dafür sogenannte „*priviledged*"-Containers verwendet werden, die bereits mit erhöhten Berechtigungen starten. Sicherer ist jedoch die Verwendung von *Capabilities* des *Linux*-Kernels, denn jeder zugrunde liegende Prozess eines Containers arbeitet üblicherweise nur mit einer reduzierten Menge an Privilegien; vollumfängliche Berechtigungen und die damit einhergehenden Angriffsvektoren werden hierdurch vermieden. Um einen Container-Host in einem zufriedenstellenden Maße zu härten, sind die Verwendung des *Secure Computing Modes* (*seccomp*) sowie die *Linux Security Modules* (LSMs) wie *AppArmor* oder *Security-Enhanced Linux* (*SELinux*) ratsam. Mit Hilfe festgelegter Profile dieser *Frameworks* lassen sich die im Container zulässigen Aktionen limitieren. Schon die dabei standardmäßig aktiven Einstellungen schützen das Betriebssystem und seine Anwendungen vor Sicherheitsbedrohungen. [Lie19, vgl. S. 85f., 199ff], [Doc20f]

Wie bereits erwähnt erfolgt die Isolation von Container-Umgebungen durch die Verwendung der nativen *Linux*-Kernel-Funktionen. Neben den *Namespaces* und *Capabilities* existiert noch das wichtige Konzept der sogenannten *cgroups* (*Control Groups*). „*Cgroups* limitieren Prozesse/Prozessgruppen/Ressourcen wie Memory, CPU, Block, Network (*Limitierung*)" [Lie19, S. 86]. Hierüber können Container vor *DoS*-Angriffen (*Denial of Service*) geschützt werden, indem für beteiligte Prozesse ein eindeutiges Ressourcen-Limit definiert wird. Bezogen auf einen *DoS*-Angriff, könnten dann Überbeanspruchungen der Host-Ressourcen vermieden werden.

Ergänzend zu den genannten Sicherheitsmechanismen kann ein Container-*Sandboxing* angewendet werden, um die Isolation von Containern zu erhöhen. Dies kann bspw. durch die Einführung einer weiteren Abstraktionsschicht geschehen. Die Abbildung 6.3 zeigt diesen Mechanismus basierend auf einer VM, welche schließlich als Container-Host agiert; bei erhöhten Anforderungen könnte in einer VM auch nur ein Container betrieben werden (*Kata Containers*). Ein zweiter Ansatz sieht bezogen auf einen Container das Hinzufügen eines eigenen Kernels vor. Eine konkrete Umsetzung kann z. B. durch die Software *gVisor* erfolgen, die 2018 von *Google* vorgestellt wurde. Letztendlich wird auch hiermit eine zusätzliche Zwischenschicht erzeugt, die den direkten Kernel-Aufruf weiter

separieren soll und aufgrund der nicht vorhandenen VM ressourcenschonender arbeitet. [Lie19, vgl. S. 86f., 89ff, 114, 509]

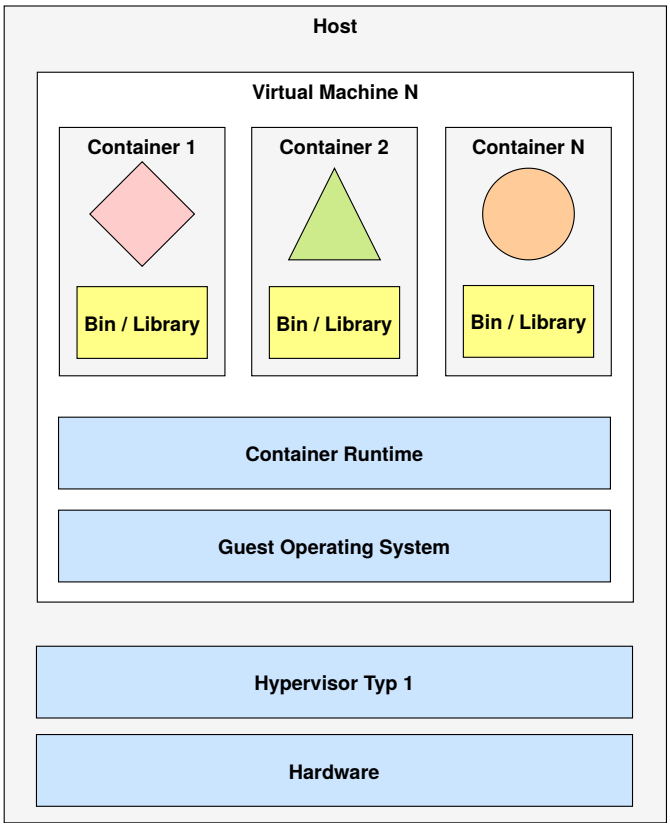


Abbildung 6.3: Container-Virtualisierung in virtuellen Maschinen [Lie19, vgl. S. 91, 509]

In einem *Kubernetes*-Cluster wird die bereitzustellende Software letztendlich in einem Container, genauer gesagt innerhalb eines *Pods*, ausgeführt. Ein *Pod* repräsentiert eine Gruppe von Containern und *Volumes*, die in der gleichen Ausführungsumgebung – also immer zusammen auf dem selben Host – laufen [AD19, vgl. S. 147]. *Pods* sind die kleinste Bereitstellungseinheit in einem *K8s*-Cluster. Auch in dieser *Container*-Schicht gibt es noch einmal zusammengefasst allgemeine Empfehlungen für die Gewährleistung der Sicherheit [Kub20g]:

- Sicherheit des zugrunde liegenden Betriebssystems –
  Das Betriebssystem eines Container-Hosts muss entsprechend „sicher“ konfiguriert sein (*Firewall*, SSH-Zugang etc.) und sollte die neuesten Sicherheitsaktualisierungen besitzen. Aufgrund der angesprochenen Problematik der Container-Isolation, sollte ein aktueller *Linux*-Kernel mit *Namespaces* und *cgroups* sowie aktiviertem *seccomp* und weiteren LSMs eingesetzt werden.

- *Scannen* von Container-Schwachstellen –
  Es sollten Maßnahmen vorherrschen, die Container-*Images* auf bekannte Schwachstellen untersuchen und zuständige Entwickler bei Auffälligkeiten benachrichtigen. Hierfür bieten sich CVE-*Scanner* an, die sogar vor dem Hinzufügen eines *Images* zu einer zentralen *Registry* vollautomatisiert eine Schwachstellenanalyse basierend auf den einzelnen *Image Layern* durchführen können. Auch die Überwachung bereits existierender Abbilder könnte durch wiederkehrende Routinen abgedeckt werden. [Lie19, vgl. S. 387ff]

- *Image*-Signierung –
  Container-*Images* sollten digital signiert und in einer zentralen *Registry* abgelegt werden,

um ein Vertrauenssystem für den Inhalt der Abbilder aufrechtzuerhalten. In diesem Zusammenhang wird dann auch von „*Trusted Images*“ gesprochen, die u. a. auch komplett selbst erstellt werden können [Lie19, vgl. S. 165ff].

- Privilegierte Benutzer sperren –
  Anwendungen innerhalb eines Containers sollten möglichst mit unprivilegierten Benutzern ausgeführt werden, d. h. sie sollten über die geringsten Betriebssystemprivilegien verfügen, die für eine funktionstüchtige Ausführung des Containers erforderlich sind. [Lie19, vgl. S. 199ff]

Im Wesentlichen sind Container vom Host-System und anderen Container-Anwendungen weniger stark isoliert als es bei VMs der Fall ist. Dennoch lassen sich unter Zuhilfenahme der aufgezeigten *Linux*-Kernel-Funktionalitäten relativ sichere Container implementieren. Vor allem wenn unprivilegierte Benutzer, ein gehärtetes Host-Betriebssystem und sichere *Images* zum Einsatz kommen, kann *Docker* als *Container Runtime* zufriedenstellend verwendet werden [Doc20f]. Als Betriebssystem bietet sich *Ubuntu* mit vordefinierten Profilen für den *seccomp* und das LSM *AppArmor* (*docker-default*) an. Des Weiteren unterstützt *Harbor* [Har20] als *Image Registry* die Integration der CVE-*Scanner Clair* oder *Trivy* und *Notary* zur digitalen Signierung von *Images*. Letztendlich ist dies eine gute Basis für die Gewährleistung von *Trusted Images* an einer zentralen Stelle in der Infrastruktur. Überdies ermöglicht *Harbor* eine automatische Replizierung der abgelegten Ressourcen mit weiteren *Registry*-Instanzen, um einem Ausfall als „*Single Point of Failure*“ (SPoF) vorzubeugen. Der im Jahr 2019 durch *Cure53* durchgeführte Penetrationstest [HWW+19] war überwiegend positiv, zeigte allerdings auch zwei gravierende Schwachstellen auf, die wiederum von den Entwicklern zeitnah geschlossen worden sind. Es zeigt sich, dass konsequent an einer sicheren *Open Source-Registry* gearbeitet wird, welche mittlerweile mehrere größere Aktualisierungen erhalten hat.

6.1.4 *Code*

Der *Code* in Container-Anwendungen stellt die niedrigste Schicht im Modell dar und kann als primäre Angriffsfläche durch die darüberliegenden Abhängigkeiten gesehen werden. Es gibt diverse Empfehlungen, die vorwiegend im Entwicklungsprozess einer Software herangezogen werden sollten und somit nicht maßgeblich in dieser Ausarbeitung verfolgt werden [Kub20g]:

- Zugang nur über TLS –
  Vorab sollte für eine im Code zu implementierende TCP-Kommunikation (*Transmission Control Protocol*) ein TLS-*Handshake* mit dem Client erfolgen. Mit Ausnahme weniger Fälle sollte alles verschlüsselt werden, was sich auf dem Transportweg befindet. Generell verstärkt es die Informationssicherheit, den Netzwerkverkehr auch zwischen zwei Diensten zu verschlüsseln. Eine zweiseitige Überprüfung der Kommunikation von Zertifikatsinhabern kann dabei bspw. durch *mTLS* (*mutual* TLS) geschehen.

- Begrenzung der Port-Bereiche für die Kommunikation –
  Für eine Anwendung sollten nur die Ports geöffnet werden, die tatsächlich für eine Kommunikation oder die Erfassung von Metriken erforderlich sind. Alle weiteren Port-Bereiche sollten für einen späteren Gebrauch keine Anwendung finden.

- Sicherheit der „*3rd Party*“-Abhängigkeiten –
  Bibliotheken von Drittanbietern sollten in regelmäßigen Abständen auf bekannte Sicherheitslücken überprüft und zudem aktuell gehalten werden. Die meisten Programmiersprachen verfügen heutzutage über entsprechende Werkzeuge zur automatischen Durchführung dieser zur Code-Sicherheit beitragenden Überprüfung.

- Statische Code-Analyse –
  Entwickelter Quellcode sollte auf unsichere Kodierungstechniken der spezifischen Programmiersprache analysiert werden. Diese Überprüfungen sollten wenn immer möglich mit automatisierten Werkzeugen durchgeführt werden, die Code-Basen auf häufige Sicherheitsfehler auswerten können. Das OWASP (*Open Web Application Security Project*) stellt hierfür eine Auflistung [OWA20] einiger Tools bereit.

- *Dynamic Probing Attacks* –
  Mit Hilfe dieses Verfahrens können automatisiert Tests bzw. bekannte Angriffe gegen ein eigenes Software-Produkt veranlasst werden. Dazu gehören u. a. SQL *Injection*-, CSRF-(*Cross-Site Request Forgery*) und XSS-Attacken (*Cross-Site Scripting*). Ein beliebtes dynamisches *Open Source*-Analysewerkzeug ist der vom OWASP zur Verfügung gestellte *Zed Attack Proxy* (ZAP).

Für die *Code*-Ebene gilt abschließend: Einfallstore durch nicht relevante Ports oder veraltete Bibliotheken vermeiden, eine durchgängig verschlüsselte Kommunikation möglichst mit der Authentifizierung beider Teilnehmer einsetzen und automatisierte Schwachstellenanalysen im Entwicklungsprozess integrieren, dann sind gute Voraussetzungen für solide und sichere Code-Basen auch im Umfeld von skalierbaren Container-Clustern geschaffen.

6.2 Automatisierung

IT-Optimierungen und das meistens damit einhergehende ökonomische Handeln spielen neben der Informationssicherheit und in Anbetracht der Bereitstellung von Ressourcen in Rechenzentren eine wichtige Rolle. Nachdem das Virtualisieren von Server- und Desktop-Systemen zum etablierten Standard geworden ist, entwickeln sich Rechenzentren durch die digitale Transformation zunehmend in Richtung des *Software-Defined Datacenters* (SDDC) [WAG+18, vgl. S. 53ff, 1235], welches weitere Bestandteile wie das Netzwerk und die Speichersysteme virtualisiert. Bereits im Unterabschnitt 3.1.1 wurde diese Entwicklung bezogen auf die Virtualisierungsumgebung *VMware vSphere* thematisiert. Ein SDDC sieht zudem operative Erleichterungen wie die Automatisierung und Orchestrierung von IT-Diensten bei immer komplexer werdenden Strukturen, wie sie für skalierbare Container-Anwendungen notwendig sind, vor. Auch Liebel verstärkt dies durch seine Aussage: „Kernstichwort für erfolgreiche Container-Landschaften ist und bleibt aber die *Automation*“ [Lie19, S. 91]. Im Allgemeinen wird unter IT-Automatisierung, „die Verwendung von Software zur Erstellung wiederholbarer Anweisungen und Prozesse, die eine menschliche Interaktion mit IT-Systemen ersetzen oder reduzieren“ [Red20a], verstanden. Eine Vielzahl von Werkzeugen existieren in diesem Themenbereich, was teilweise zu anspruchsvollen Lernkurven für Administratoren führen kann. Nichtsdestotrotz ist davon auszugehen, dass in zukünftigen Software-Systemen eine größere Autonomie vorhanden sein und demzufolge auch eine höhere Intelligenz eingebettet wird, wie es bereits in Ansätzen bei *Kubernetes* zu sehen ist. Entscheidungen können automatisiert vom System selbst ohne direkte Kontrollen und Unterbrechungen des Betriebs getroffen werden. [WAG+18, vgl. S. 1071], [Red20a]

Gerade in *Enterprise*-Umgebungen bestehen oft eine Reihe von Wünschen und Anforderungen hinsichtlich der Automation der IT-Landschaft. Die automatisierte Bereitstellung von Ressourcen nimmt im Gegensatz zur gewohnten manuellen Provisionierung von VMs eine immer größere Bedeutung ein. Die nachstehende Auflistung soll einen kompakten Überblick über mögliche Anwendungsfälle geben [WAG+18, vgl. S. 1071ff], [Red20a]:

- Bereitstellung von Ressourcen (Basis-Infrastruktur) –
  Eine gute Basis für Automatisierungen stellen Virtualisierungsumgebungen bzw. SDDCs

dar, die dynamisch IT-Ressourcen virtualisiert zur Verfügung stellen und automatisiert um neue Hardware-Bestandteile skaliert werden können. Konkret wird mit *VMware vSphere* solch eine in großen Teilen Software-definierte Umgebung bereitgestellt, in der mit geringen Aufwand eine automatisierte Integration von neuen Komponenten, wie bspw. *ESXi*-Hosts, erfolgen kann. Die dafür benötigten Tools werden von *VMware* bereitgestellt. Ist erst einmal eine solche virtualisierte Umgebung aufgebaut, können diverse Vorteile, wie bereits im Unterabschnitt 3.1.1 ersichtlich wurde, genutzt und schließlich virtuelle Ressourcen des Rechenzentrums u. a. über isolierte VMs bereitgestellt werden.

- Konfigurationsmanagement –
  Bei der stetig wachsenden Zunahme von automatisierten und komplexer werdenden Bereitstellungen in einem Rechenzentrum ist es von Vorteil, ein zentrales Konfigurationsmanagement einzusetzen, über das die zur Verfügung gestellten IT-Ressourcen (*Bare Metal*-Hosts, VMs, Container etc.) definiert werden können. Letztendlich wird hierüber eine wiederholbare und robuste Umgebung geschaffen, die eine effektivere Verwaltung der IT-Systemlandschaft ermöglicht. Als *Open Source*-Software für das Konfigurationsmanagement kann an dieser Stelle *Puppet* genannt werden, welche auch am AWI eingesetzt wird. *VMware* hingegen bietet die „*Host Profiles*“-Funktion an, um Konfigurationen auf neue *ESXi*-Hosts zu übertragen. Weitergehende Maßnahmen in der *vSphere*-Umgebung können mit Hilfe der *PowerCLI* via *Scripting* oder über *Workflows* durch *vRO* durchgeführt werden [WAG+18, vgl. S. 1086f.]. Auch *Rancher* bietet die Möglichkeit an, Konfigurationen in Form von RKE-Cluster- und Knoten-*Templates* für automatisierbare Provisionierungen abzulegen, sowie konsistente Richtlinien-Definitionen (RBAC, PSPs, *Network Policies*) global zu spezifizieren [Ran20f]. Im *Cloud Native*-Bereich muss hier noch einmal auf das *Cloud-init*-Werkzeug verwiesen werden, welches sich für die Konfiguration von Betriebssystemen – u. a. die im Abschnitt 3.2 erwähnten *Ubuntu Cloud Images* – eignet.

- Automatisiertes Erzeugen von Anwendungsumgebungen (VMs, Container) –
  Neben der Automatisierung der Basis-*vSphere*-Infrastruktur rückt auch das Erzeugen von komplexeren Anwendungsumgebungen in den Vordergrund, um letztendlich Bereitstellungszeiten für den Endanwender zu verkürzen. Nicht nur einfache VMs, sondern ganze vorinstallierte Server-Anwendungen mit mehrschichtigen Architekturen inklusive Netzwerkkomponenten wie LBs müssen dafür automatisiert in die bestehende Infrastruktur mit den vorhandenen Management-Tools integriert werden. Zu erzeugende Anwendungsumgebungen können z. B. auch auf skalierbaren Container-Infrastrukturen basieren und die Verwaltung aller beweglichen Bestandteile somit komplexer ausfallen. Generell wird es dann für umfangreichere Vorgänge sinnvoll eine Orchestrierung aufzubauen, die weitestgehend autonome (Cluster-)Umgebungen ermöglicht. Konkret arbeitet bspw. *Rancher* als *Orchestrator* und stellt RKE-Container-Cluster auf der Grundlage von provisionierten VMs zur Verfügung. Für diese Automation in der Virtualisierungsumgebung verwendet *Rancher* einen integrierten *vSphere*-Treiber [Ran20f, Ran20c]. Auch auf der *VMware*-Seite gibt es u. a. den *Orchestrator vRO*, mit dem über automatisierbare *Workflows* komplexe Bereitstellungen direkt in der *Cloud*-Management-Plattform der *vRealize Suite* vorgenommen werden können [WAG+18, vgl. S.].

- Bereitstellung von Anwendungen (CI / CD) –
  In der traditionellen Anwendungsentwicklung aber auch wenn in Unternehmen eine *DevOps*-Philosophie (s. Abschnitt 2.4) mit CI / CD zum Einsatz kommt, sind effiziente Entwicklungspipelines – insbesondere während der Testphase – auf robuste und automatisierte Betriebsprozesse angewiesen, um die vollumfänglichen Erwartungen der Entwickler (*Dev*) sowie Betriebsteams (*Ops*) zu erfüllen. Vorausgesetzt wird die Automatisierung der Bereitstellungsvorgänge in der zugrunde liegenden IT-Infrastruktur. Hierbei müssen unter

Umständen VMs, separate logische Netzwerke, zusätzliche Netzwerkkomponenten oder auch Container provisioniert werden. Automatisierte Prozesse können dabei helfen, vom *Commit* zum *Build* über *Tests* bis hin zum *Deployment* auf eine zuverlässige, transparente Art zu gelangen. Schlussendlich kann über automatisierte Anwendungsbereitstellungen die Möglichkeit von menschlichen Fehlern reduziert und gleichzeitig die Effizienz und der Durchsatz verbessert werden. Gegenständlich können mit Hilfe der *GitLab* CI / CD kontinuierliche Bereitstellungen über eine Pipeline aufgebaut und darüber Container-Anwendungen in *Kubernetes*-Clustern platziert werden [Git20a].

- *Security* und *Compliance* –
  Bei der zunehmenden Komplexität von Rechenzentren kann die Automatisierung auch beim Thema *Security* helfen. Automatisierte Vorgänge verringern potentielle Fehlerherde, die zuvor durch manuelle Konfigurationen entstehen konnten. Des Weiteren werden durch standardisierte Sicherheitsprozesse und automatisierte *Workflows* einfacher *Compliance* und Audits ermöglicht. Neue Prozessanforderungen können schneller verifiziert und leichter in die gesamte IT-Infrastruktur konsistent integriert werden. Hilfreich hierfür ist eine strukturierte Überwachung der beteiligten IT-Ressourcen durch Protokollierungen. Mit *vRealize Log Insight* und den OPA sind Produkte für das *VMware*-Virtualisierungsumfeld vorhanden [WAG+18, vgl. S. 53, 945], die verknüpft mit *Icinga2* über Vorfälle automatisiert benachrichtigen können. Auch *Rancher* besitzt ein aggregiertes *Logging* seiner *Downstream*-RKE-Cluster und gestattet somit eine zentrale Überwachung der bereitgestellten Container-Cluster [Ran20f]. Zudem verwendet die *Image Registry Harbor* CVE-*Scanner*, um automatisiert Schwachstellen in den hinzugefügten Abbildern zu finden und daraufhin verantwortliche Benutzer zu kontaktieren [Har20].

- *Governance* –
  In Organisationen steht neben der automatisierten Provisionierung von Ressourcen zusätzlich die sogenannte *Governance* bezogen auf die strategische Führung der IT eines Unternehmens im Fokus. Hierüber soll der komplette Lebenszyklus von IT-Ressourcen über Richtlinien und Prozesse gesteuert werden, u. a. sind mehrstufige Genehmigungsprozesse oder auch spezielle Richtlinien über die Bereitstellungsdauer von Ressourcen involviert. Um solchen Anwendungsfällen gerecht zu werden, sind *Cloud*-Management-Plattformen wie etwa *vRealize Automation* (*vRA*) von *VMware* auf dem Markt erhältlich; *vRA* ist in der *vRealize Suite* enthalten und wurde bereits in Unterabschnitt 3.1.2 ausführlicher thematisiert.

- *Self-Service*-Portale –
  Damit Endkunden wie etwa Entwickler IT-Ressourcen anfordern können, werden in vielen Firmen zusätzlich zu der angesprochenen Automatisierung von *Workflows*, interne *Self-Service*-Portale errichtet, die zumeist im Stil einer *Public Cloud* erscheinen. Bei *VMware* ist dieser Funktionsumfang in der *vRealize Suite* hauptsächlich durch die Produkte *vRA* und *vRO* abgedeckt (s. Unterabschnitt 3.1.2). Sie ermöglichen den Aufbau einer *Cloud*-Management-Plattform, in der gewünschte Ressourcen in einem webbasierten *Self-Service*-Katalog bestellt werden können. Anschließend werden die IT-Ressourcen mittels integrierter Automatisierungstechniken [WAG+18, vgl. S. 1073ff] bereitgestellt und sind somit meistens innerhalb weniger Minuten für den Anforderer verfügbar.

7 Realisierung der Infrastruktur

In diesem Kapitel wird die praktische Umsetzung des angedachten Szenarios beschrieben. Konkret wird hier versucht, die funktionalen und nicht-funktionalen Anforderungen aus dem Kapitel 4 bezogen auf die involvierten Interaktionsplattformen abzubilden. Wie sich bereits in den vorherigen Kapiteln herauskristallisiert hat, wird die Umsetzung des Szenarios wesentlich durch die Installation bzw. Konfiguration und Integration der folgenden vier Plattformen getragen: *Harbor* als *Image Registry*, *Rancher* mit RKE als *Cluster Management*, die *VMware vRealize Suite* (*vRA*, *vRO*) als *Self-Service Portal* und *GitLab* als *CI / CD Pipeline*.

Die Grundlage für die Umsetzung der Ausarbeitung ist das Vorhandensein einer *Cloud Native*-Virtualisierungsumgebung im Rechenzentrum des AWIs. Die nachfolgende Abbildung 7.1 ist eine Konkretisierung der Abbildung 4.1 und zeigt schematisch die Bestandteile des Szenarios in der *VMware vSphere*-Umgebung basierend auf dem *Bare Metal-Hypervisor ESXi*. Generell kommen mehrere *Hypervisor* zum Einsatz, die in einem auf zwei Standorte verteilten Cluster-Verbund betrieben werden. Hierdurch kann eine hohe Verfügbarkeit gewährleistet und der Ressourcen-Pool durch eine Lastverteilung effizient ausgenutzt werden; auf die *VMware*-Produkte wird ausführlicher im Abschnitt 3.1 eingegangen.

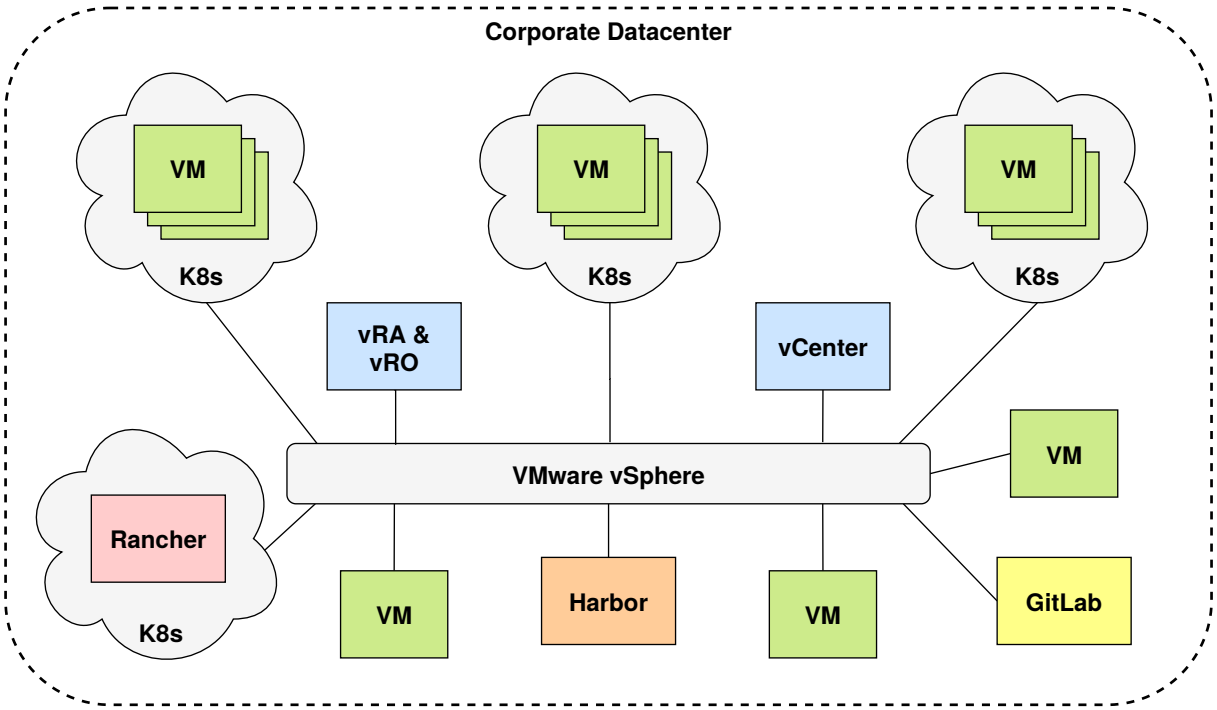


Abbildung 7.1: Virtualisierungsumgebung des Szenarios

Mit Hilfe von *vSphere* können VMs *On-Premise* bereitgestellt werden, die in diesem Kontext auch als Cluster-Knoten agieren sollen. Die einzelnen Knoten eines *K8s*-Clusters – also die *Control Plane*, als auch die nur Container-Anwendungen ausführenden *Nodes* – können dabei durch einzelne VMs und dem *Cloud Native*-Betriebssystem *Ubuntu* realisiert werden. Anschließend

besteht die Möglichkeit Container-Anwendungen (*Pods*) in den zur Verfügung stehenden Clustern zu orchestrieren. Die eigentlichen *Kubernetes*-Cluster in Ausprägung der *K8s*-Distribution RKE können mit dem *Enterprise Cluster Management*-Tool *Rancher* erstellt werden. Zudem bietet dieses Werkzeug an einer zentralen Stelle einfache Interaktionsmöglichkeiten (Administration, Überwachung, Anwendungsbereitstellung etc.) mit einem aus mehreren Knoten bestehenden Cluster an. Für die Bereitstellung in Produktivumgebungen wird vom Hersteller eine redundante Auslegung in einem eigenen *Kubernetes*-Cluster empfohlen. Des Weiteren ist ein *Self-Service Portal* mit automatisierten *Workflows* in Form der *vRealize Suite*-Bestandteile *vRA* und *vRO* erforderlich, über die der „Bestellvorgang“ einer Cluster-Ressource vollzogen und die dazugehörige Abrechnung im Organisationskontext erfolgen kann. Bereitzustellende Container-*Images* können dabei in die vorgesehene *Image Registry Harbor* abgelegt bzw. dann wieder innerhalb der Infrastruktur abgerufen werden. Um darüber hinaus eine kontinuierliche Integration (CI) und Bereitstellung (CD) von Software-Anwendungen zu ermöglichen, wird die Integration eines *GitLab*-Servers erforderlich.

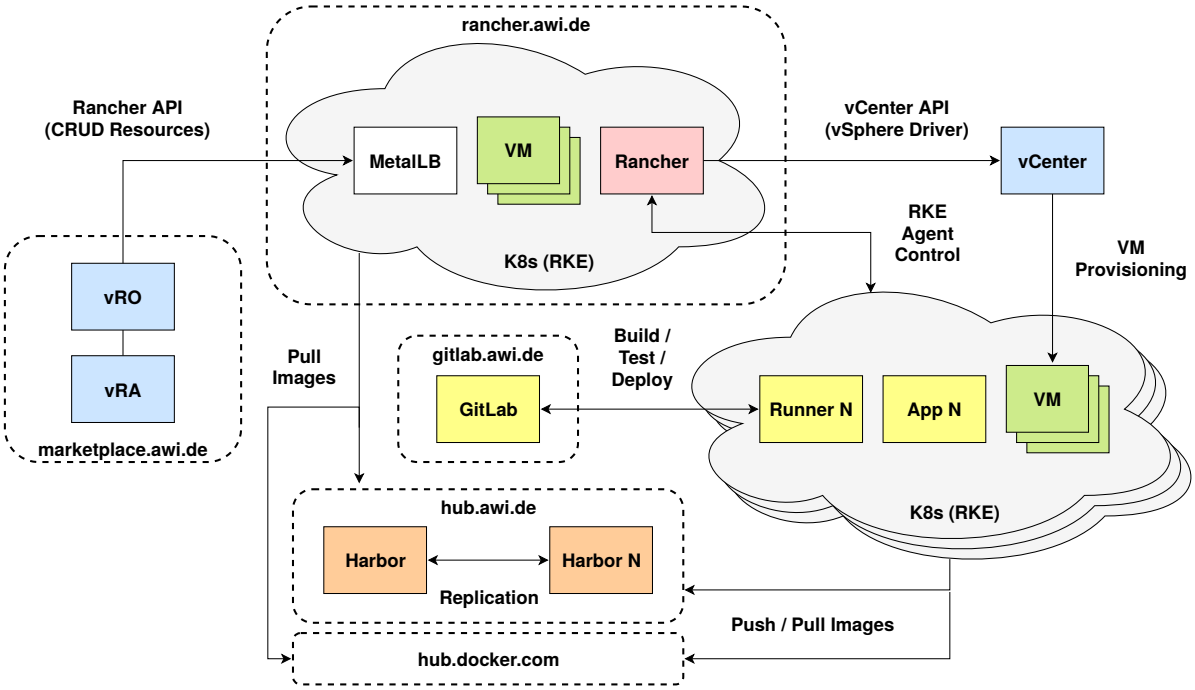


Abbildung 7.2: Abhängigkeiten der Interaktionsplattformen innerhalb des Szenarios

Für ein noch besseres Verständnis und einen klareren Gesamtzusammenhang werden in der Abbildung 7.2 die verschiedenen Abhängigkeiten der beteiligten Interaktionsplattformen vereinfacht dargestellt. Das *Self-Service Portal* ist unter der URL „`https://marketplace.awi.de`“ zu finden und leitet eine hier veranlasste Cluster-Bestellung an das *Cluster Management*-Tool *Rancher* durch CRUD-Ressourcen-Operationen (*Create*, *Read*, *Update*, *Delete*) weiter. Der externe Cluster-Zugriff auf die hochverfügbare *Rancher*-Anwendung mit der URL „`https://rancher.awi.de`“ erfolgt dabei über den *Load Balancer MetalLB*. Anschließend provisioniert *Rancher* über das *vCenter* der *vSphere*-Virtualisierungsumgebung und mit Hilfe zuvor angelegter *Templates* eine definierte Anzahl von VMs, die letztendlich zusammen einen neuen Cluster bilden. Danach veranlasst *Rancher* wieder automatisiert die Installation und Konfiguration der notwendigen Bestandteile eines RKE-Clusters. Nach der erfolgreichen Erstellung einer neuen Cluster-Ressource kann in *Rancher* selbst die Verwaltung des jeweiligen RKE-*Downstream*-Clusters über die in RKE integrierten *Cluster / Node Agents* stattfinden. Die für den Aufbau zuvor benötigten Container-*Images* können der *Image Registry Harbor* unter der URL „`https://hub.awi.de`“ entnommen

bzw. entsprechend hinzugefügt werden. In diesem Kontext wird es auch möglich sein *Images* aus der öffentlichen *Docker Hub Registry* (`https://hub.docker.com`) zu verwenden. Darauf aufbauend können Container-Anwendungen (*App*) dann kontinuierlich über die *DevOps*-Plattform *GitLab*, welche über die URL „`https://gitlab.awi.de`“ erreichbar ist, voll automatisiert bereitgestellt werden. In diesem Zusammenhang werden eine Pipeline-Verarbeitung (*Build / Test / Deploy*) und die sogenannten *Runner* verwendet. Manuell kann eine Anwendungsbereitstellung in einem *Kubernetes*-Cluster auch direkt über die *Rancher*-Benutzeroberfläche oder nativ mit der *kubectl* erfolgen. Alle auftretenden Bestandteile sind wie bereits erwähnt in der *VMware vSphere*-Umgebung angesiedelt und werden möglichst automatisiert aufgebaut.

In den nachstehenden Abschnitten werden die vier Interaktionsplattformen bezogen auf die praktische Umsetzung noch einmal genauer betrachtet.

7.1 *Image Registry*

Die Basis für die skalierbare Container-Infrastruktur bildet die vollständig containerisiert vorliegende *Image Registry Harbor*. In ihr können Container-*Images* abgelegt und entsprechend wieder für Software-Bereitstellungen in Clustern heruntergeladen werden. Zudem stehen bereits nach der Installation einige Sicherheitsfunktionen wie CVE-*Scanner* und das digitale Signieren von *Images* zur Verfügung. Ausführlichere Informationen zu *Harbor* sind im Abschnitt 3.4 zu finden.

Harbor wurde in einer VM installiert, um die angestrebte HA-Architektur, wie sie der Abbildung 7.2 zu entnehmen ist, umzusetzen. Es kann eine Replikation zwischen verschiedenen *Image Registries* – es muss dafür nicht unbedingt eine weitere *Harbor*-Instanz sein – konfiguriert werden, damit die einzelnen *Repositories* mit ihren *Images* möglichst ausfallsicher betrieben werden können. Die Spezifikation der VM ist der Tabelle 7.1 zu entnehmen. Zur Lastverteilung und der Verwendung mehrerer *Registries* unter einer zentralen Adresse wurde der DNS-Eintrag „*hub.awi.de*“ angelegt, der wiederum auf die virtuellen Instanzen verweist. Für die Sicherung des Hosts wurde u. a. eine lokale *Firewall* mit dem Paket *ufw* (*uncomplicated firewall*) aufgebaut, die nur die notwendigen Ports (22, 80, 443, 4443) nach außen freigibt, und mit Hilfe von *fail2ban* der SSH-*Daemon* vor *Brute-Force*- bzw. *DoS*-Angriffen geschützt. Des Weiteren müssen *Docker* als *Container Runtime*, *Docker Compose* und *Openssl* installiert vorliegen.

Tabelle 7.1: Spezifikation der *Harbor*-VM

Hostname	*hub.awi.de* → *hub-reg1.awi.de*
OS	*Ubuntu 18.04.4* LTS
CPU	4
RAM	8 GB
Storage	250 GB (*/opt/harbor* (*Configuration*), */data* (*Data / Database*))
Software	*Docker Engine* (*v19.03*), *Docker Compose* (*v1.26*), *Harbor* (*v2.0.1*)

Die einzelnen Schritte der *Harbor*-Installation innerhalb einer bereitgestellten VM sind in der Auflistung 7.1 aufgeführt. Als erstes müssen die Installationsdateien aus dem offiziellen *GitHub-Repository* heruntergeladen und die Archivdatei entpackt werden. Optional kann im Vorfeld die heruntergeladene Datei mit Hilfe einer Prüfsumme und einer digitalen Signatur verifiziert werden. Danach erfolgt die Konfiguration des HTTPS-Zugriffs auf *Harbor*. Hierfür müssen TLS-Zertifikate mit dem entsprechenden FQDN (*Fully Qualified Domain Name*) angelegt und durch eine vertrauenswürdige CA (*Certificate Authority*) signiert werden. Die Zertifikatsdateien „*.crt*“, „*.cert*“ und „*.key*“ werden daraufhin für *Harbor* und *Docker* in die richtigen Dateipfade

kopiert. An dieser Stelle ist wichtig zu erwähnen, dass der *Nginx*-Webserver, der von *Harbor* eingesetzt wird, die komplette Zertifikatskette als öffentliches TLS-Zertifikat benötigt. *Docker* hingegen erwartet dieses Zertifikat zwingend mit der Dateiendung „*.cert*“. Als nächstes wird die Hauptkonfigurationsdatei „*harbor.yml*“ der *Registry*-Anwendung angepasst. Hier erfolgen auf der einen Seite allgemeine Einstellungen zu den Zertifikaten und dem Host-Zugriff. Auf der anderen Seite können in der YAML-Datei (*YAML Ain´t Markup Language*) aber auch diverse Konfigurationen an den einzelnen *Harbor*-Komponenten vorgenommen werden, bspw. lassen sich die zu verwendenden Datenbanken oder CVE-*Scanner* von extern – ohne auf dem selben Host zu laufen – anbinden. Darüber hinaus kann in einem weiteren Schritt und unter Zuhilfenahme eines *Docker*-Containers eine verschlüsselte Kommunikation zwischen den einzelnen *Harbor*-Komponenten durch die Erstellung von zusätzlichen TLS-Zertifikaten konfiguriert werden. In der letzten Anweisung der Auflistung 7.1 wird das umfassende Installationsskript aufgerufen, dem Parameter für die Installation von optionalen Komponenten mitgegeben werden können. Hier lassen sich direkt *Notary* für die digitale Signierung von *Docker Images*, *Clair* und *Trivy* als CVE-*Scanner* sowie *ChartMuseum* als *Helm Chart Repository* mitinstallieren.

Auflistung 7.1: Installation der *Image Registry Harbor*

```
#!/bin/bash
# Download the installer
curl -LO "https://github.com/goharbor/harbor/releases/download/v2.0.1/harbor-online-installe
↪  r-v2.0.1.tgz"

# Unpack the installer
tar -xvf harbor-online-installer-v2.0.1.tgz

# Configure HTTPS access to Harbor
cp ca.crt hub-reg1.awi.de.crt hub-reg1.awi.de.key /data/cert
sudo cp ca.crt hub-reg1.awi.de.cert hub-reg1.awi.de.key /etc/docker/certs.d/
sudo systemctl restart docker

# Configure the Harbor YML file
vim harbor.yml

# Configure internal TLS communication between Harbor components
docker run -v /:/hostfs goharbor/prepare:v2.0.1 gencert -p /data/cert/intern

# Run the installer script
sudo ./install.sh --with-notary --with-clair --with-trivy --with-chartmuseum
```

In der Auflistung 7.2 sind die verschiedenen Komponenten mit den benötigten Netzwerk-Ports zu sehen, die für den vollen Funktionsumfang von *Harbor* benötigt werden. Sie stammen letztendlich aus dem öffentlichen *Docker Hub-Repository* „*goharbor*“ und werden im Zuge der Bereitstellung auf dem Host-System ausgeführt.

Auflistung 7.2: *Microservice*-Komponenten der *Image Registry Harbor*

```
$ docker-compose ps
Name                Ports
------------------------------------------------------------------------------------
chartmuseum         9999/tcp
clair               6060/tcp, 6061/tcp
clair-adapter       8080/tcp
harbor-core
harbor-db           5432/tcp
harbor-jobservice
```

```
harbor-log          127.0.0.1:1514->10514/tcp
harbor-portal       8080/tcp
nginx               0.0.0.0:4443->4443/tcp, 0.0.0.0:80->8080/tcp, 0.0.0.0:443->8443/tcp
notary-server
notary-signer
redis               6379/tcp
registry            5000/tcp
registryctl
trivy-adapter       8080/tcp
```

Nach der Installation wird *Harbor* bzw. die einzelnen Container-Komponenten mit Hilfe von *Docker Compose* verwaltet (s. Auflistung 7.3). *Docker Compose* agiert in diesem Zusammenhang als ein lokaler *Orchestrator*, welcher *Microservices* mit definierten Konfigurationen aus einer „*docker-compose.yml*“-Datei starten und wieder stoppen kann. Schlussendlich lassen sich hierüber komplexe Container-Anwendungen aus mehreren Komponenten von einer zentralen Stelle aus dirigieren.

Auflistung 7.3: Verwalten der *Image Registry Harbor* via *Docker Compose*

```
#!/bin/bash
# Stop and restart Harbor
sudo docker-compose stop
sudo docker-compose start

# Reconfigure Harbor
sudo docker-compose down -v
vim harbor.yml
sudo ./prepare --with-notary --with-clair --with-trivy --with-chartmuseum
sudo docker-compose up -d
```

Um die *Harbor*-Installation abzuschließen, müssen noch einige Konfigurationen in der webbasierten Benutzeroberfläche getätigt werden. Nach der Anmeldung mit dem lokalen Administrator-Konto sollte als erstes das dazugehörige Passwort durch eine stärkere Zeichenkette angepasst werden. Anschließend können globale Konfigurationen wie etwa die Benutzer-Authentifizierung über den LDAP-Verzeichnisdienst des AWIs, die Verwendung eines E-Mail-Servers für Benachrichtigungen und weitere Systemeinstellungen vorgenommen werden. Wichtig unter den erweiterten Einstellungen sind u. a. das Definieren einer Projekt-Quota, die den verwendbaren Speicherplatz der *Repositories* einschränkt, das Aktivieren der automatischen Schwachstellenanalyse der *Open Source*-CVE-*Scanner Clair* von *CoreOS* oder *Trivy* von *Aqua Security*, sowie die automatische *Garbage Collection*, um nicht mehr benötigte *Image*-Bestandteile in der *Registry* zu bereinigen. Des Weiteren können an dieser Stelle andere *Registry*-Instanzen angebunden werden, mit denen schließlich eine vollständige oder teilweise Replikation erfolgen soll. Hierfür müssen zuerst *Registry*-Endpunkte hinzugefügt und danach aufbauende Replikationsregeln (*Push-based*, *Pull-based*) konfiguriert werden.

Damit *Images* in *Harbor* hochgeladen werden können, muss initial eine Projekt-Ressource angelegt werden, in der alle *Repositories* einer zugehörigen Container-Anwendung platziert werden. Wurde ein Projekt erstellt und Benutzer mit einer entsprechenden Rolle (RBAC) diesem zugewiesen, dann kann, wie in der Auflistung 7.4 zu sehen ist, ein lokales *Image* mit einem *Tag* versehen und letztendlich hochgeladen (*push*) werden. Auf anderen Hosts kann dieses Image dann wieder heruntergeladen (*pull*) werden. Ist ein Projekt nicht öffentlich zugänglich, dann muss im Vorfeld eine Anmeldung an der *Registry* erfolgen. Diverse Einstellungen lassen sich an einem *Harbor*-Projekt vornehmen, u. a. kann eine automatische Schwachstellenanalyse direkt nach dem

Hochladen eines *Images* gestartet, oder auch nur signierte Abbilder (*Trusted Images*) bei einem „*pull*“-Vorgang erlaubt werden. Außerdem können sogenannte *Helm Charts* in einem Projekt bereitgestellt werden, die in *Kubernetes*-Clustern für die Anwendungsbereitstellung Verwendung finden können. *Helm* ist ein Paket-Manager für *K8s* und benutzt das Paketformat *Chart* als Manifest, in dem der Aufbau einer Anwendung durch *K8s*-Ressourcen beschrieben ist.

Auflistung 7.4: Interaktionsmöglichkeiten mit der *Image Registry Harbor* in der *Bash*

```
#!/bin/bash
# Build an image from a dockerfile
docker build -f DOCKERFILE . #-t hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Create a target image tag that refers to a source image
docker tag SOURCE_IMAGE[:TAG] hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Log in to a Docker registry
docker login hub.awi.de

# Push an image or a repository to a registry
docker push hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Sign an image - a simple push is also sufficient
docker trust sign hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Pull an image or a repository from a registry
docker pull hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Remove trust for an image
docker trust revoke hub.awi.de/PROJECT/REPOSITORY[:TAG]

# Log out from a Docker registry
docker logout hub.awi.de
```

Um *Images* beim Hochladen via *Notary* zu signieren, müssen die Umgebungsvariablen aus der Auflistung 7.5 verwendet werden. Für eine permanente Integration können die Befehle auch in der „*.bashrc*“- oder der „*.profile*“-Datei im Nutzerverzeichnis hinzugefügt werden. Ein *Signed-Push* in die *Registry* erfolgt komplett automatisiert, es sind lediglich Passworteingaben (*root key*, *repository key*) für die Signatur des Images erforderlich. Sobald diese *Notary*-spezifischen Direktiven gesetzt sind, kann es bei der Interaktion mit anderen *Registries* allerdings zu Fehlern kommen. Dann müssen die Umgebungsvariablen wieder mit einem „*unset*“ bereinigt werden.

Auflistung 7.5: Aktivieren von *Docker Content Trust* für das Signieren von *Docker Images*

```
#!/bin/bash
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=https://hub.awi.de:4443
```

7.2 *Cluster Management*

Als zentrale Verwaltungsplattform für die verteilten *Kubernetes*-Cluster wird *Rancher* mit der dazugehörigen *K8s*-Distribution RKE eingesetzt. *Rancher* konsolidiert die skalierbaren Container-Infrastrukturen in einem gebündelten *Cluster Management*, welches u. a. Cluster-Operationen,

globale Richtlinien und eine RBAC-Benutzerverwaltung ermöglicht. Ausführlichere Informationen zu den Software-Produkten von *Rancher Labs* sind im Abschnitt 3.6 zu finden.

Nach Empfehlung des Herstellers wurde *Rancher* selbst innerhalb eines RKE-Clusters bereitgestellt, um die Anforderungen der Hochverfügbarkeit und Ausfallsicherheit zu gewährleisten [Ran20d]. Dieser Cluster basiert wiederum auf drei VMs, die nach der aus der Tabelle 7.2 zu entnehmenden Spezifikation aufgebaut wurden. Als eindeutige URL bzw. für einen zentralen Nutzerzugang wurde der DNS-Eintrag „*rancher.awi.de*“ angelegt, der von allen später zu provisionierenden *Downstream*-Clustern aufzulösen sein muss. Dieser Eintrag verweist auf den im Cluster befindlichen *Load Balancer MetalLB* [Met20], der für eine Lastverteilung sorgt und letztendlich den Netzwerkverkehr auf die virtuellen Instanzen verteilt. Für die initiale Einrichtung der drei *Ubuntu*-Hosts wurde diesmal das Hilfswerkzeug *Cloud-init* in der *vSphere*-Umgebung verwendet (s. Auflistung A.8). Hierdurch werden pro VM-Instanz vollautomatisiert u. a. Nutzerkonten erstellt, der SSH-*Daemon* konfiguriert, *Docker* installiert und die *Firewall ufw* mit entsprechenden Richtlinien aktiviert.

Tabelle 7.2: Spezifikation der drei *Rancher*-VMs

Hostname	*rancher.awi.de* → *rancher1.awi.de*, *rancher2.awi.de*, *rancher3.awi.de*
OS	*Ubuntu 18.04.4* LTS
CPU	4
RAM	8 GB
Storage	30 GB
Software	*Docker Engine* (*v19.03*), *Kubernetes*/RKE (*v1.15*), *Rancher* (*v2.4.5*), *MetalLB* (*v0.9.3*), *Ingress-Nginx* (*v0.34*)

Für die administrative Verwaltung der zugrunde liegenden Hosts des Clusters dient der SSH-Zugang ausschließlich über SSH-Schlüssel, die die auf elliptischen Kurven basierende Verschlüsselungstechnik *Ed25519* anwenden. Die Auflistung 7.6 zeigt, wie entsprechende SSH-Schlüssel generiert und dem lokalen Authentifizierungsagenten hinzugefügt werden. Auch die rudimentäre Host-Verwaltung der späteren *Downstream*-Cluster wird nur über diese *Keys* möglich sein. Die Verwendung von herkömmlichen Passwörtern wird dementsprechend nicht erlaubt, wie es der *Cloud-init*-Konfiguration in der Auflistung A.8 zu entnehmen ist.

Auflistung 7.6: Verwendung von *Ed25519*-SSH-Schlüsseln für die Container-Cluster

```
#!/bin/bash
# Authentication key generation
ssh-keygen -t ed25519 -C ubuntu -f ~/.ssh/id_ed25519_ubuntu
ssh-keygen -t ed25519 -C rke -f ~/.ssh/id_ed25519_rke

# Adds private key identities to the authentication agent
ssh-add ~/.ssh/id_ed25519_ubuntu ~/.ssh/id_ed25519_rke
```

Der in diesem Abschnitt betrachtete *Rancher Server* kann auch in einer sogenannten „*Air Gapped Environment*“ installiert werden, d. h. es wird eine *Offline*-Installation, hinter einer *Firewall* oder einem *Proxy* realisiert. Für dieses Szenario ist eine *Private Registry* wie *Harbor* unabdingbar, mit der *Docker Images* in der vom Internet abgeschotteten Umgebung zugänglich gemacht werden. Für die Umsetzung bedarf es einiges an Mehraufwand, da sämtliche Container-*Images* – auch die *Rancher*-System-*Images* – für einen *Offline*-Zugriff gepflegt werden müssen. Die Auflistung 7.7 zeigt, wie die *Rancher*-Abbilder eines bestimmten Releases mit Hilfe bereitgestellter Skripte heruntergeladen und in die eigene *Registry* wieder hochgeladen werden können; die entsprechenden

Projekte / *Repositories* müssen dafür vorher in *Harbor* erstellt sein. Dieses Szenario wurde aus Test-Zwecken ausprobiert, für den späteren produktiven Aufbau aufgrund des Mehraufwands und der zugrunde liegenden Anforderungen aber nicht weiter verfolgt. Dennoch sind in den nachstehenden Auflistungen immer wieder Hinweise auf Maßnahmen bzgl. der „*Air Gapped Environment*“ vermerkt.

Auflistung 7.7: Bereitstellen der *Rancher-Images* bei der ausschließlichen Verwendung einer *Private Registry*

```
#!/bin/bash
# Save the images to your workstation
curl -LO "https://github.com/rancher/rancher/releases/download/v2.4.5/{rancher-images.txt,rancher-save-images.sh}"
chmod +x rancher-save-images.sh
./rancher-save-images.sh --image-list rancher-images.txt

# Populate the private registry
curl -LO "https://github.com/rancher/rancher/releases/download/v2.4.5/rancher-load-images.sh"
chmod +x rancher-load-images.sh
./rancher-load-images.sh --image-list rancher-images.txt --registry hub.awi.de
```

Nach der Bereitstellung der VMs folgt die Installation der *Kubernetes*-Distribution RKE, die schließlich aus den verteilten Hosts einen Cluster-Verbund erzeugt. In der Auflistung 7.8 ist das dafür verwendete RKE-*Template* „*rancher-cluster.yml*“ ersichtlich, welches *K8s* mit der Version *1.15.12* in einem Verbund aus drei Hosts bereitstellt. Die einzelnen Hosts nehmen dabei alle drei Rollen (*controlplane*, *etcd*, *worker*) eines RKE-Clusters ein und werden mit Hilfe des Benutzerkontos „*rke*“ und dem zuvor erstellten SSH-Schlüssel eingerichtet. Laut *Rancher Labs* sollten für einen optimalen Betrieb neben *Rancher* keine zusätzlichen *Workloads* im Cluster ausgeführt werden. Für die spätere Konfiguration des *Load Balancers MetalLB* muss zudem die standardmäßig installierte *Ingress*-Ressource deaktiviert werden. Der *etcd*-Dienst wird zudem mit einem separaten Benutzerkonto (*52034*) ausgeführt.

Auflistung 7.8: RKE-*Template* für die Installation des *Cluster Managements Rancher*

```
cluster_name: rancher
kubernetes_version: v1.15.12-rancher2-2
ssh_key_path: ~/.ssh/id_ed25519_rke

nodes:
- address: ...
hostname_override: rancher1
user: rke
role: ['controlplane', 'etcd', 'worker']
- address: ...
hostname_override: rancher2
user: rke
role: ['controlplane', 'etcd', 'worker']
- address: ...
hostname_override: rancher3
user: rke
role: ['controlplane', 'etcd', 'worker']

# Scenario: Air Gapped Environment
#private_registries:
#  - url: hub.awi.de
#    is_default: true
```

```
23
    services:
25  etcd:
    gid: 52034
27  uid: 52034
    snapshot: true
29  creation: 6h
    retention: 24h
31
    ingress:
33  provider: none
```

Die eigentliche Bereitstellung des *Rancher*-RKE-Clusters, wie sie in der Auflistung 7.9 zu sehen ist, wird mit der RKE CLI in Verbindung mit dem RKE-*Template* „*rancher-cluster.yml*“ aus der Auflistung 7.8 durchgeführt. Mit Hilfe der *kubectl* kann dann im Nachgang der Cluster-Status überprüft werden. Für einen erfolgreichen Zugriff auf die *Kubernetes*-API wird die bei der *K8s*-Installation automatisch erzeugte „*kubeconfig*“-Datei „*kube_config_rancher-cluster.yml*“ im Suchpfad benötigt. Zusammen mit der „*rancher-cluster.rkestate*“-Datei sollte die „*kubeconfig*“-Datei zudem gesichert werden, um auch zukünftig den vollen Cluster-Zugriff zu behalten. Die Installationsmaßnahmen der immer wieder auftauchenden Tools können im Abschnitt A.1 nachvollzogen werden.

Auflistung 7.9: Bereitstellen des *Rancher*-RKE-Clusters

```
1  #!/bin/bash
   # Run RKE
3  rke up --config rancher-cluster.yml

5  # Test your cluster
   mkdir -p ~/.kube && cp kube_config_rancher-cluster.yml ~/.kube/config
7  kubectl [--kubeconfig ...] get nodes

9  # Check the health of your cluster pods
   kubectl [--kubeconfig ...] get pods --all-namespaces
```

Nachdem der RKE-Cluster erfolgreich aufgebaut wurde, kann die eigentliche *Rancher*-Installation, wie in der Auflistung 7.10 dargestellt, durchgeführt werden. Mit Hilfe von *Helm* wird *Rancher* in dem *K8s-Namespace* „*cattle-system*“ bereitgestellt. Mit *Helm Chart Repositories* können *Kubernetes*-Manifeste von Anwendungen einfacher verwaltet und letztendlich in einem Container-Cluster installiert werden [Lie19, vgl. S. 1129ff]. Darüber hinaus muss für einen verschlüsselten Zugriff auf das *Cluster Management Rancher* ein signiertes TLS-Zertifikat mit der vollen Zertifikatskette (*tls.crt*, *tls.key*) und dem entsprechenden FQDN zugänglich gemacht werden; hierfür wird dem *K8s*-Cluster eine *Secret*-Ressource hinzugefügt. Stehen keine eigenen vertrauensvollen Zertifikate bereit, dann lässt sich auch der „*jetstack/cert-manager*“ im Cluster installieren, der auf selbst-signierte TLS-Zertifikate oder die CA *Let's Encrypt* aufbaut. Anschließend kann mit der *kubectl* der Bereitstellungsstatus des *Rancher Servers* kontrolliert werden. *Rancher* selbst lässt sich für die spätere Wartung auch ohne eine Neuinstallation aktualisieren, indem die *Helm Chart Repositories* mit „*helm repo update*“ aktualisiert und ein „*helm upgrade rancher ...*“ mit den selben Parametern wie bei der Installation ausgeführt werden.

Auflistung 7.10: Installation des *Cluster Managements Rancher* im RKE-Cluster

```
#!/bin/bash
# Add the Helm chart repository
helm repo add rancher-stable https://releases.rancher.com/server-charts/stable

# Create a namespace for Rancher
kubectl [--kubeconfig ...] create namespace cattle-system

# Scenario: Air Gapped Environment
#helm fetch rancher-stable/rancher
#helm template rancher ./rancher-v2.4.5.tgz --output-dir . --namespace cattle-system --set
↪ hostname=rancher.awi.de --set rancherImage=hub.awi.de/rancher/rancher --set
↪ ingress.tls.source=secret --set systemDefaultRegistry=hub.awi.de --set
↪ useBundledSystemChart=true
#kubectl [--kubeconfig ...] -n cattle-system apply -R -f ./rancher

# Install Rancher with Helm and your chosen certificate option
helm [--kubeconfig ...] install rancher rancher-stable/rancher --namespace cattle-system
↪ --set hostname=rancher.awi.de --set ingress.tls.source=secret

# Adding TLS secrets
kubectl [--kubeconfig ...] -n cattle-system create secret tls tls-rancher-ingress
↪ --cert=tls.crt --key=tls.key

# Verify that the Rancher server is successfully deployed
kubectl [--kubeconfig ...] -n cattle-system rollout status deploy/rancher
```

Wie weiter oben bereits erwähnt, kommt *MetalLB* [Met20] als LB innerhalb des *Kubernetes*-Clusters zum Einsatz. Dieser Ansatz hat den Vorteil, dass keine externe LB-Instanz für den *Rancher Server* verwaltet werden muss und ein ansonsten in *K8s* nicht vorgesehener vollständiger *Layer 2-Load Balancer* im *Bare Metal Environment*-Cluster genutzt werden kann [Kub20e]. Damit *MetalLB* mit Hilfe der *kubectl* und der herunterzuladenden YAML-Manifeste erfolgreich ausgeführt werden kann, muss zuvor eine *Ingress*-Ressource wie bspw. der *Nginx-Ingress-Controller* in *Kubernetes* ausgeführt werden. Wichtig hierbei ist, dass im Manifest des *Controllers* anstelle des „*ServiceTypes*“ „*NodePort*“ der Typ „*LoadBalancer*“ gesetzt wird. Die Konfiguration des LBs wird abschließend mit dem Hinzufügen einer *Kubernetes-Config Map* im *Namespace* „*metallb-system*“ abgeschlossen. Diese *Config Map* „*metalLB.yml*“ enthält u. a. die IP-Adressen, für die der LB letztendlich aktiv werden soll.

Auflistung 7.11: Integration des *Layer 2-Load Balancers MetalLB* in *Kubernetes*

```
#!/bin/bash
# Installation of Nginx ingress controller (bare metal environment)
curl -LO "https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v0.34.0/depl
↪ oy/static/provider/baremetal/deploy.yaml"
sed -i 's/NodePort/LoadBalancer/g' deploy.yaml
kubectl [--kubeconfig ...] apply -f deploy.yaml

# Installation of MetalLB by manifest
kubectl [--kubeconfig ...] apply -f
↪ https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/namespace.yaml
kubectl [--kubeconfig ...] apply -f
↪ https://raw.githubusercontent.com/metallb/metallb/v0.9.3/manifests/metallb.yaml
# On first install only
kubectl [--kubeconfig ...] create secret generic -n metallb-system memberlist
↪ --from-literal=secretkey="$(openssl rand -base64 128)"
```

```
# Layer 2 configuration of MetalLB
kubectl [--kubeconfig ...] apply -f metalLB.yml
```

Die Auflistung 7.12 gibt einen Überblick über alle aktiven *Pods* mit ihren *Namespaces* innerhalb des hier thematisierten *Rancher*-RKE-Clusters: *Rancher* (*cattle-system*), *Nginx-Ingress-Controller* (*ingress-nginx*), *Kubernetes* (*kube-system*) und *MetalLB* (*metallb-system*).

Auflistung 7.12: *Kubernetes-Pods* mit *Namespaces* des *Rancher*-RKE-Clusters

```
$ kubectl [--kubeconfig ...] get pods -A --field-selector=status.phase==Running
NAMESPACE        NAME
cattle-system    cattle-cluster-agent-6c5dd5ffbd-knvzw
cattle-system    cattle-node-agent-5fkrg
cattle-system    cattle-node-agent-bw7kf
cattle-system    cattle-node-agent-xpmg9
cattle-system    rancher-674c4dd9fb-b28f4
cattle-system    rancher-674c4dd9fb-b6xgm
cattle-system    rancher-674c4dd9fb-v7rmj
ingress-nginx    ingress-nginx-controller-8864f7cf9-vf596
kube-system      canal-dv6g6
kube-system      canal-pj2s8
kube-system      canal-w5vsg
kube-system      coredns-799dffd9c4-btpgc
kube-system      coredns-799dffd9c4-wc4sp
kube-system      coredns-autoscaler-7868844956-mfgvh
kube-system      metrics-server-59c6fd6767-qjzkc
metallb-system   controller-7686dfc96b-z2bzf
metallb-system   speaker-9rvjz
metallb-system   speaker-ffk7g
metallb-system   speaker-tj8js
```

Für einen erfolgreichen Abschluss der *Rancher*-Installation müssen noch weitere Konfigurationen in der webbasierten Benutzeroberfläche der Management-Anwendung absolviert werden. Dies ist zu aller erst die Passwort-Änderung des lokalen Administrator-Kontos durch eine starke Zeichenkette und die Eingabe der vollständigen *Rancher Server*-URL „*https://rancher.awi.de*“, welche im Netzwerk von allen beteiligten Komponenten erreicht werden muss. Danach erfolgt die Einbindung des LDAP-Verzeichnisdienstes des AWIs für die Authentifizierung von Benutzern im Organisationskontext. Zu beachten ist, dass nach der Integration des Verzeichnisdienstes lokale *Rancher*-Benutzerkonten nicht auf Daten des LDAP-Dienstes zugreifen können. Sollen demzufolge weitere Benutzerkonten in diesem Zusammenhang verwendet werden, müssen entsprechend Konten auf der Ebene des Verzeichnisdienstes angelegt werden. Außerdem sind die Standard-Cluster-Rollen (RBAC) für Benutzerkonten anzupassen bzw. weiter einzuschränken und *Cloud Credentials* für die Benutzung der *vSphere*-Virtualisierungsumgebung im Zuge einer automatisierten VM-Bereitstellung zu erstellen. Zuvor kann noch ein neues Konto im Verzeichnisdienst erstellt werden, das für den automatischen *Workflow* letztendlich agiert und mit weniger Berechtigungen (*Standard User*) im *Cluster Management Rancher* ausgestattet wird. Des Weiteren wurden *Templates* für die RKE-Cluster-Erstellung sowie der automatisierten Knoten-Bereitstellung angelegt. Für die RKE-*Downstream*-Cluster wird ein gehärtetes RKE-*Template* verwendet, welches in der Auflistung A.9 in voller Länge vorzufinden ist. Hier fließen sowohl *Best Practices* [Ran20e] des Herstellers als auch Analyse-Erkenntnisse aus dem Kapitel 6 mit ein. In den *Node-Templates* wird der Zugriff auf die *vSphere*-Umgebung konfiguriert, VM-Host-Ressourcen definiert und die Möglichkeit gegeben Einstellungen via *Cloud-init* (s. Auflistung A.8) für jede Knoten-Instanz im Bereitstellungsvorgang zu definieren.

Für den *vSphere-Node-Driver* von *Rancher* sind auf der Seite der *VMware vSphere*-Umgebung noch die Erstellung eines Benutzerkontos und die Vergabe spezieller Berechtigungen im *vCenter* notwendig, damit die Provisionierung der VMs im Bereitstellungskontext – sprich im Rechenzentrum des AWIs – geschehen kann [Ran20b]. Außerdem sind im *vCenter* VM-*Templates* anzulegen, auf die in den besagten *Node-Templates* der *Cloud Management*-Plattform wiederum referenziert wird. Als Grundlage dienen hierfür die stetig aktualisierten *Ubuntu Cloud Images* [Ubu20b] als ursprüngliches OVF-*Template* (*Open Virtualization Format*). Dieses Format hat den Vorteil, dass keine Betriebssysteminstallation veranlasst werden muss, sondern lediglich ein bereits installiertes System importiert wird. Mit Hilfe der „*vApp Options Properties*“ der VM und optionalen *Cloud-init*-Anweisungen lassen sich dann pro Instanz speziellere Einstellungen vornehmen. Um auch auf Betriebssystemebene auf die *Properties* zugreifen zu können, muss der „OVF *Environment Transport*“ aktiviert sein.

7.3 *Self-Service Portal*

Damit IT-Ressourcen durch interne Kunden des AWIs – sprich WissenschaftlerInnen – selbstständig angefordert und auf die entsprechenden Kostenstellen gebucht werden können, wird das vorhandene *Self-Service Portal* basierend auf der *VMware vRealize Suite* eingebunden. Maßgeblich sind hierfür auch die zu hinterlegenden *Workflows*, die eine automatisierte Zurverfügungstellung der Ressourcen, bspw. in der Form eines *K8s*-Clusters, erst ermöglichen. Ausführlichere Informationen der Bestandteile der *vRealize Suite* sind im Unterabschnitt 3.1.2 zu finden.

In der Abbildung 7.2 wird bereits ersichtlich, dass das *Self-Service Portal* als Interaktionsplattform in zwei Bestandteile untergliedert werden kann. Auf der einen Seite ist es die *Hybrid Cloud*-Automatisierungsplattform *vRA*, die unter der URL „`https://marketplace.awi.de`“ einen Service-Katalog über IT-Dienstleistungen und -Ressourcen für Endanwender bereitstellt. Auf der anderen Seite ist dies der *vRO*, mit dessen Hilfe automatisierbare *Workflows* entwickelt bzw. vorgehalten und wiederum in *vRA* integriert werden können. Beide Produkte liegen nach der Installation der *VMware vRealize Automation Appliance* in der Version *7.5* vor und stehen bezogen auf das zu realisierende Szenario in unmittelbarer Abhängigkeit mit der API des *Rancher Servers*. Nachfolgend werden zunächst die notwendigen Maßnahmen im *vRO* und anschließend die Konfigurationen in der *vRA*-Plattform erläutert.

Der *vRealize Orchestrator* [WAG+18, vgl. S. 1086f.] beinhaltet bereits nach der Installation eine Menge vorkonfigurierter *Workflows* für eine *vSphere*-Umgebung. Auch viele weitere Hersteller bieten bereits fertige *Plug-ins* für ihre Produkte an, um sie in das *VMware*-Ökosystem zu integrieren. Für die Interaktion mit einem *Rancher Server* sind diesbezüglich noch keine Inhalte zu finden, daher müssen benutzerdefinierte *Workflows* entwickelt werden. Der *vRO*-Zugriff wird dabei über einen *Java*-basierten Client realisiert und die individuellen *Workflows* in der „*Design*“-Rubrik des Tools mit der Skriptsprache *JavaScript* (JS) umgesetzt. Bevor mit der Entwicklung der einzelnen *Workflows* gestartet werden kann, muss ein sogenannter „*Dynamic Type*“ mit zugehörigem *Namespace* angelegt werden. Instanzen dieser Objekte repräsentieren im späteren Verlauf innerhalb des *Self-Service Portals* eine angeforderte Cluster-Ressource, mit der durch die Verwendung des *Dynamic Types* schließlich interagiert werden kann. Mit den beiden vorhandenen *Workflows* „*Define Namespace*“ und „*Define Type*“ wird im ersten Schritt der Namensraum „*DynamicTypes:Rancher*“ und anschließend der *Dynamic Type* „*DynamicTypes:Rancher.Cluster*“ angelegt. Damit die Bestandteile der *vRealize Suite* mit diesem neuen und unbekannten Cluster-Objekt vollwertig interagieren können, müssen noch die nachstehenden, automatisch generierten *Workflows* mit Inhalt gefüllt werden: „*Find All Rancher-Cluster*“, „*Find Rancher-Cluster By Id*“, „*Find Relation Rancher-Cluster*“ und „*Has Rancher-Cluster Children In Relation*“. Maßgeblich

ist hierbei der *Workflow* „*Find Rancher-Cluster By Id*“, damit die Cluster-Objekte des genannten Typs im Kontext richtig aufgelöst werden können. Hier ist zu beachten, dass das benutzerdefinierte Cluster-Objekt mit dem folgenden Aufruf innerhalb des *Workflows* neu erstellt und zurückgegeben wird: „*resultObj = new DynamicTypesManager().makeObject(namespace, type, id , name, [props ...]);*“. Erst nach diesen Vorkehrungen wurden die eigentlichen *Workflows* durch die Verwendung des Bausteins „*Scriptable Tasks*“ und dem Hinzufügen von JS-Code entwickelt, um letztendlich mit der *Rancher*-API zu kommunizieren. In der Tabelle 7.3 sind die zu automatisierenden *Workflows* aufgeführt, die im *vRealize Orchestrator* erarbeitet wurden.

Tabelle 7.3: Übersicht der entwickelten *Workflows* im *vRO*

Workflow	**Kurzbeschreibung**	**Quellcode**
rancher_create_cluster	Einen *K8s*-Cluster erstellen.	Auflistung 7.14
rancher_delete_cluster	Einen *K8s*-Cluster entfernen.	Auflistung A.11
rancher_create_cluster_user	Einen Benutzer auf einer *K8s*-Cluster-Ebene erstellen.	Auflistung A.12
rancher_delete_cluster_user	Einen Benutzer auf einer *K8s*-Cluster-Ebene entfernen.	Auflistung A.13
rancher_create_project	Eine *Rancher-Project*-Ressource in einem *K8s*-Cluster erstellen.	Auflistung A.14
rancher_delete_project	Eine *Rancher-Project*-Ressource in einem *K8s*-Cluster entfernen.	Auflistung A.15
rancher_create_project_user	Einen Benutzer auf einer *Rancher-Project*-Ebene erstellen.	Auflistung A.16
rancher_delete_project_user	Einen Benutzer auf einer *Rancher-Project*-Ebene entfernen.	Auflistung A.17
rancher_create_worker	Weitere *Worker*-Knoten in einem *K8s*-Cluster erstellen.	Auflistung A.18
rancher_delete_worker	Weitere *Worker*-Knoten in einem *K8s*-Cluster entfernen.	Auflistung A.19

Die *Rancher*-API ist unter der URL „*https://rancher.awi.de/v3*“ erreichbar und benötigt für einen erfolgreichen Zugriff die Authentifikation des Anfragenden durch eine HTTP *Basic Authentication* unter der Verwendung eines API-*Keys*. Dieser Schlüssel wird in der Benutzeroberfläche des *Cluster Managements Rancher* mit einer zu wählenden Gültigkeitsdauer angelegt. Nachdem dies geschehen ist, kann im *vRO* ein globales Objekt vom Typ „*REST:RESTHost*“ angelegt werden, in dem die REST-Zugriffsinformationen für den *Rancher Server* enthalten sind. Die Auflistung 7.13 zeigt die *vRO-Action* „*excuteRequest*“, welche HTTP-*Requests* an einem bestimmten REST-Host übermittelt und die Antwort im JSON-Format (*JavaScript Object Notation*) zurückgibt. Diese und weitere Hilfsfunktionen (s. Auflistung A.10) wurden für einen globalen Zugriff im Modul „*de.awi.rancher*“ abgelegt, um u. a. Code-Redundanzen zu vermeiden. *vRO-Actions* sind äquivalent zu JS-Funktionen. Generell werden in allen aufgezeigten *Workflows* Veränderungen an den beteiligten Ressourcen der *Rancher*-RKE-Cluster mit der CRUD-Semantik verwirklicht.

Auflistung 7.13: *Action* „*executeRequest*“ des Moduls „*de.awi.rancher*“ im *vRO*

```
1 const RANCHER = System.getModule('de.awi.rancher');

3 function executeRequest(restHost, requestType, operationUrl, requestContent) {
        var request = restHost.createRequest(requestType, operationUrl,
        ↪  JSON.stringify(requestContent));
```

```
        var response = request.execute();
        if(!(response.statusCode >= 200) || !(response.statusCode < 300)) {
                throw 'Error calling the API! Status code: ' + response.statusCode;
        }
        return (RANCHER.isEmptyString(response.contentAsString))? '' :
        ↪  JSON.parse(response.contentAsString);
}
```

In der Auflistung 7.14 wird der komplette JS-Code des *Workflows* „*rancher_create_cluster*“ dargestellt, der für die Erstellung eines neuen *K8s*-Clusters ausgeführt werden muss. Konkret wird durch HTTP-*Requests* der *Rancher Server* beauftragt, eine neue Cluster-Ressource mit einer zugehörigen „*NodePool*“-Ressource anzulegen. Letztendlich wird ein neuer *Kubernetes*-Cluster bestehend aus drei Knoten (VMs) mit allen drei *K8s*-Funktionen (*Control Plane*, *etcd* sowie *Worker*) in der *vSphere*-Virtualisierungsumgebung provisioniert. Zur Verringerung des Konfigurationsaufwands kommen hier die zuvor erstellten Cluster- und *Node-Templates* der *Rancher*-Plattform zum Einsatz. Im Vorfeld wird noch ein einzigartiger Cluster-Name generiert, indem u. a. der aktuelle UNIX *Timestamp* (*Uniplexed Information and Computing System*) verwendet wird. Wichtig zu erwähnen ist, dass als Ausgabeparameter des *Workflows* das in *vRO* eigens definierte Cluster-Objekt vom Typen „*DynamicTypes:Rancher.Cluster*“ zurückgegeben werden muss, um die spätere Integration in *vRA* abschließen zu können.

Auflistung 7.14: *Workflow „rancher_create_cluster“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get New Cluster Name */
var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters', null);
var clusterNames = resObj.data.map(function(value, index, array) {
        return value.name;
});

var newClusterName = null;
do {
  newClusterName = RANCHER.getUid(uid) + '-k8s-' + Date.now();
} while(clusterNames.indexOf(newClusterName) !== -1);

/* Create Cluster */
var requestContent = {
    "clusterTemplateRevisionId": RANCHER.getClusterTemplateRevisionId(clusterTemplate),
    "name": newClusterName
};
resObj = RANCHER.executeRequest(restHost, 'POST', '/clusters', requestContent);
clusterId = resObj.id;

/* Create Node Pool */
requestContent = {
    "clusterId": clusterId,
    "controlPlane": true,
    "etcd": true,
    "hostnamePrefix": newClusterName + '-',
    "nodeTemplateId": RANCHER.getNodeTemplateId(nodePoolTemplate),
    "quantity": 3,
    "worker": true
};
resObj = RANCHER.executeRequest(restHost, 'POST', '/nodepools', requestContent);
nodePoolId = resObj.id;
```

```
35 /* Get DynamicTypesDynamicObject */
rancherCluster = new DynamicTypesManager().getObject('Rancher' , 'Cluster' , clusterId);
```

Im Allgemeinen wurde darauf geachtet alle wesentlichen Eingabeparameter der *Workflows* mit Hilfe von regulären Ausdrücken (*Regular Expressions*, *Regex*) sowohl in den *vRO- / vRA*-Formularen (*Presentation*) als auch im JS-Code durch den Funktionsaufruf von „*String.match('Regex')*“ zu validieren. Die Tabelle 7.4 gibt einen Überblick über die verwendeten regulären Ausdrücke. Des Weiteren werden ungültige Eingabewerte durch die Verwendung des „*default*“-Schlüsselworts in den *Switch-Case*-Blöcken mit Standardwerten ersetzt. Darüber hinaus wird das Abbrechen von *Workflows* durch *Exceptions* zugelassen, um eine fehlerhafte Ausführung eines *Workflows* zu unterbinden; hierzu werden auch eigene *Exceptions* mit entsprechenden Nachrichten geworfen. Weitere Code-Auflistungen sind im Anhang unter Abschnitt A.4 aufgeführt.

Tabelle 7.4: Validierung der Eingabeparameter mit *Regex* in *vRO / vRA*

Eingabeparameter	***Regex***	**Beispielwert**
uid	`^[a-z0-9-]{1,8}@dmawi\.de$`	fmangels@dmawi.de
username	`^\D{1,}$`	Fabian Mangels
clusterName	`^[a-z0-9-]{1,8}-k8s-[0-9]{1,}$`	fmangels-k8s-1596026499416
nodePoolName	`^[a-z0-9-]{1,8}-k8s-[0-9]{1,}-worker-[0-9]{1,}-$`	fmangels-k8s-1596026499416-worker-1596027175973-
projectName	`^(?!System)([a-zA-Z0-9- ]{1,})$`	Default

Einzelne zusammengeschaltete *Workflows* können wiederum einen übergeordneten *Workflow* ergeben, was die Abbildung 7.3 verdeutlicht. So ist es möglich direkt nach der Erstellung eines *K8s*-Clusters, einen Benutzer auf der Cluster-Ebene hinzuzufügen und diesen danach dem RKE-Standard-Projekt „*Default*“ zuzuordnen. Kleine *Workflows* können somit für größere Abläufe genutzt und bspw. mit vorgegebenen Eingabeattributen (*In Attributes*) versehen werden. Die jeweiligen Eingaben und Ausgaben in Form von *Parameters* und *Attributes* werden im *vRO* zur besseren Übersicht visuell miteinander verknüpft. Sollen Ergebnisse eines *Workflows* an einen nachgeschalteten *Workflow* weitergegeben werden, dann müssen dafür die globalen Ausgabeattribute (*Out Attributes*) genutzt werden. Wieder ist es wichtig darauf zu achten, dass für die spätere Zuordnung in der *vRA*-Plattform alle *Workflows* das Cluster-Objekt vom Typen „*DynamicTypes:Rancher.Cluster*“ entweder als Eingabe erwarten oder als Ausgabe produzieren.

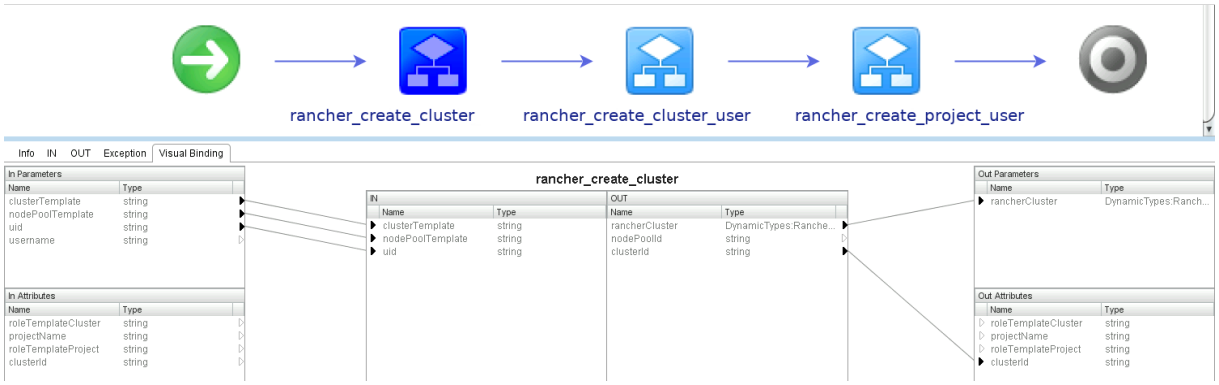


Abbildung 7.3: Schema mit Parameter- und Attribut-Zuweisungen des übergeordneten *Workflows* „*rancher_create_cluster_default*“ im *vRO*

In der Automatisierungsplattform *vRealize Automation* müssen nun die erstellten *vRO-Workflows* integriert und über einen Katalogeintrag dem Nutzer zugänglich gemacht werden. Der Zugriff auf *vRA* erfolgt dabei webbasiert über die URL „`https://marketplace.awi.de`“, um Konfigurationen vorzunehmen sowie den Ressourcen-Katalog einzusehen. Die ersten Maßnahmen werden unter der Rubrik „*Design*“ und dort unter dem Eintrag „*XaaS*“ ausgeführt. Hier wird zunächst eine sogenannte „*Custom Resource*“ (*Rancher Cluster*) mit dem im *vRO* erschaffenden Cluster-Objekt „*Dynamic Types:Rancher.Cluster*“ in Beziehung gesetzt. Mit Hilfe dieser *Custom Resource* wird es dann möglich angeforderte Ressourcen separaten Ressourcen-Operationen zuzuordnen. Danach erfolgt das Anlegen eines *XaaS Blueprints* – in diesem Fall mit dem Namen „*K8s Cluster*“ – mit dem letztendlich die angedachte K8s-Cluster-Ressource provisioniert werden soll. Im Erstellungsprozess wird der *vRO-Workflow* „*rancher_create_cluster_default*“ herausgesucht und ein Formular (*Blueprint Form*) für die im *Workflow* betroffenen Eingabeparameter erstellt. Dann wird die in diesem *XaaS Blueprint* produzierte Ressource „*Rancher Cluster*“ mit dem entsprechenden Ausgabeparameter des ausgewählten *Workflows* in Beziehung gesetzt. Abschließend können noch weitere *Workflows* bezogen auf die Ressourcen-Löschung oder -Aktualisierung (*Component Lifecycle*) hinzugefügt werden. Um auch Ressourcen-spezifische Operationen (*Day-2 Operations*) später vorzunehmen, bedarf es darüber hinaus die Erstellung von sogenannten „*Resource Actions*“. An dieser Stelle wurden für alle restlichen im *vRO* hinterlegten *Workflows* entsprechende *Resource Actions* angelegt. Inbegriffen ist wieder die Erstellung eines Formulars (*Action Form*) für die Eingabeparameter und die Zuordnung der jeweiligen *Resource Action* zur *Custom Resource* „*Rancher Cluster*“ anhand eines Eingabeparameters des *vRO-Workflows*. Um den Erstellungsprozess des *XaaS Blueprints* und der *Ressource Actions* abzuschließen, müssen diese nur noch durch Betätigen der Schaltfläche „*Publish*“ veröffentlicht werden.

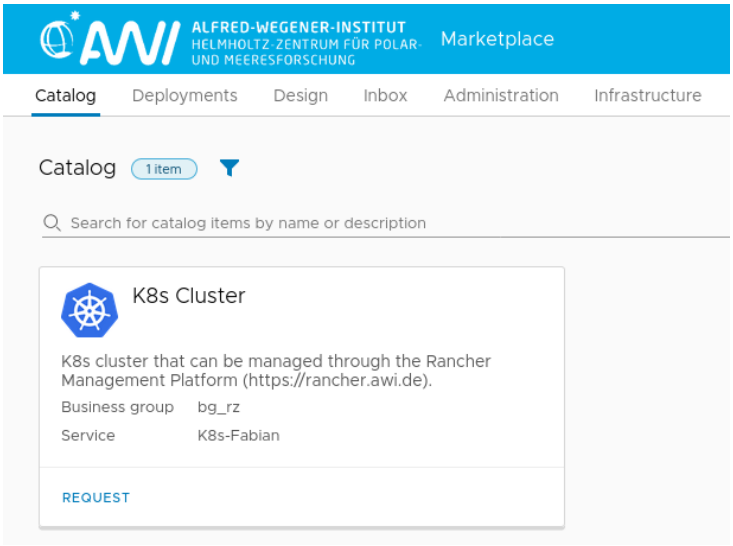


Abbildung 7.4: Katalogeintrag für einen *K8s*-Cluster im *Self-Service Portal*

Damit der neu erstellte *XaaS Blueprint* „*K8s Cluster*“ auch im Ressourcen-Katalog des *Self-Service Portals* angezeigt wird, müssen unter der Rubrik „*Administration*“ und dort unter dem Eintrag „*Catalog Management*“ noch Einstellungen getätigt werden. Als erstes wird ein *Service* (*K8s-Fabian*) angelegt, dem im zweiten Schritt *Catalog Items* bzw. ein *Blueprint* (*K8s Cluster*) zugeordnet werden. Dieser Katalogeintrag wird dann nur noch aktiviert und steht denjenigen Nutzern somit zur Verfügung, denen entsprechende Berechtigungen (*Entitlements*) zugeordnet wurden. In der Abbildung 7.4 wird das Ergebnis des erstellten Katalogeintrags ersichtlich. Die IT-Ressource „*K8s Cluster*“ kann nun über das *Self-Service Portal* angefragt bzw. „bestellt“ werden. Nach einem „*REQUEST*“ der Ressource, wird der hinterlegte *vRO-Workflow*

„*rancher_create_cluster_default*“ für die Provisionierung des *Kubernetes*-Clusters ausgeführt und unter der Rubrik „Deployments“ als Artefakt hinterlegt. Anschließend können auf das *vRA-Deployment* weitere Ressourcen-Operationen (*Day-2 Operations*) angewendet werden.

7.4 CI / CD Pipeline

Die Automatisierung der Bereitstellungsvorgänge von Anwendungen ist ein wichtiger Faktor bei der *DevOps*-Bewegung. Durch die automatisierten Prozesse einer *CI / CD Pipeline* können u. a. menschliche Fehler reduziert und dazu parallel die Effizienz der Software-Entwicklung gesteigert werden. *GitLab* CI / CD verwirklicht die Schaffung einer solchen Pipeline-Verarbeitung mit der Verknüpfung eines dazugehörigen Quellcode-*Repositories* und lässt nahezu unendliche Gestaltungsmöglichkeiten bezogen auf *Build*, *Test* oder auch *Deploy Stages* zu. Ausführlichere Informationen zu *GitLab* sind im Abschnitt 3.7 zu finden.

Die *DevOps*-Plattform *GitLab*, welche in der Version *12.9* vorliegt, ist unter der URL „`https://gitlab.awi.de`“ erreichbar und dient den Entwicklern vorwiegend zur Versionierung des Quellcodes und als Werkzeug der kollaborativen Anwendungsentwicklung. Wie die Abbildung 7.2 zeigt, steht die benutzerdefinierte *CI / CD Pipeline* eines *GitLab*-Projektes unmittelbar mit einem oder mehreren *GitLab Runnern* in Beziehung. Ein *Runner* wird verwendet, um Pipeline-Aufträge auszuführen und die Ergebnisse wieder an *GitLab* zurückzuliefern. Solch ein *Runner* kann passenderweise innerhalb eines *Kubernetes*-Clusters durch den „*Kubernetes Executor*“ ausgeführt werden [Git20b]. Für jeden neuen Pipeline-Auftrag wird daraufhin ein eigener *Pod* im angegebenen Namensraum des Clusters erschaffen, der die zuvor definierten CI- / CD-Anweisungen ausführt.

Das Einrichten eines *GitLab Runners* kann dafür, wie in der Auflistung 7.15 zu sehen ist, mit Hilfe des *Helm Chart Repositories* „`https://charts.gitlab.io`“ geschehen. Damit der *Runner* in einem *K8s*-Cluster bereitgestellt werden kann, muss zuvor eine *Rancher-Project*-Ressource mit privilegierten Berechtigungen (PSP: *unrestricted*) und einem zugeordneten *Namespace* (*gitlab-runner*) angelegt werden. Weitere Konfigurationen können entweder durch eine YAML-Datei, oder direkt als Parameter beim Installationsaufruf mitgegeben werden. Für die Einrichtung sind unbedingt die vollständige URL des *GitLab*-Servers „`https://gitlab.awi.de/`“ (*gitlabUrl*) sowie ein Token (*runnerRegistrationToken*) zur Registrierung des *Runners* anzugeben. Letzteres wird von der *GitLab*-Instanz unter „*Settings*“, dann „*CI / CD*“ und dort unter „*Runners – Set up a specific Runner manually*“ abgerufen. Damit auch automatisch ein Service-Konto für einen *Runner* bei aktiven RBAC-Richtlinien angelegt wird, muss der Parameter „*rbac.create=true*“ gesetzt werden. Mit „*runners.privileged=true*“ wird es in diesem Kontext zudem möglich, Container im privilegierten Modus ausführen zu lassen; u. a. werden in der *Build Stage* „*Docker-in-Docker*“-Konstrukte (*dind*) verwendet, für die diese Anforderung notwendig ist. Allerdings bringt die Vergabe erhöhter Berechtigungen wieder mehrere Risiken bis hin zu einer möglichen Host-Kompromittierung mit sich. *Images* können auch ohne privilegierten Modus gebaut werden, als Beispiel-Projekt ist hier *kaniko* [Goo20b] zu erwähnen. Eine weitere Thematisierung des Tools würde an dieser Stelle zu weit führen, dennoch sollte es zukünftig und über die Ausarbeitung hinaus noch einmal betrachtet werden. Für eine auf der Seite des Nutzers erleichterte Umsetzung dieser *GitLab Runner*-Installation, könnte ein zusätzlicher *Workflow* über das *Self-Service Portal* angeboten werden. Dieser würde als einzigen Eingabeparameter den individuellen „*runnerRegistrationToken*“ haben und alle Einstellungen am *K8s*-Cluster automatisiert vornehmen. Das Vorgehen der *Runner*-Zurverfügungstellung ist zu diesem Zeitpunkt noch nicht geklärt, da es auch noch berechtigte Alternativen zur Abwägung gibt.

Auflistung 7.15: Manuelles Einrichten eines *GitLab Runners* innerhalb eines *K8s*-Clusters

```
#!/bin/bash
# Add the Helm chart repository
helm repo add gitlab https://charts.gitlab.io

# Create a namespace for the GitLab Runners
kubectl [--kubeconfig ...] create namespace gitlab-runner

# Install GitLab Runner with Helm and specific GitLab settings
helm [--kubeconfig ...] install --namespace gitlab-runner gitlab-runner gitlab/gitlab-runner
↪  --set gitlabUrl="https://gitlab.awi.de/" --set runnerRegistrationToken="..." --set
↪  rbac.create=true --set runners.privileged=true
```

Alternativ zum manuellen Hinzufügen eines *Runners*, gibt es die Möglichkeit der direkten Einbindung eines gesamten *Kubernetes*-Clusters innerhalb eines *GitLab*-Projektes. Hiermit würde die Installation eines *Runners* dann automatisch vorgenommen werden. Diese Variante erfordert allerdings noch tiefgreifendere Berechtigungen im eigentlichen *K8s*-Cluster und veranschlagt einen höheren Konfigurationsaufwand. Darüber hinaus gibt es noch die mögliche Integration der sogenannten „*Shared Runners*“, die allerdings nur durch einen Administrator der *GitLab*-Plattform eingerichtet werden können. Dies wäre eine denkbare Alternative, da der Nutzer selbst keinen größeren Konfigurationsaufwand hätte und die Gefahr einer privilegierten *Runner*-Ausführungsumgebung durch die Administratoren dementsprechend besser gekapselt und – was noch viel wichtiger ist – auch unter deren Kontrolle stehen würde.

Eine Pipeline für die kontinuierliche Integration (CI) und Bereitstellung (CD) von Software, die durch die erstellten *Runner* abgearbeitet wird, kann im Wurzelverzeichnis eines *Git-Repositories* durch das Anlegen der YAML-Datei „*.gitlab-ci.yml*“ definiert werden. In der Auflistung 7.16 ist ein Ausschnitt einer beispielhaften *Build Stage* aufgeführt, in der der Quellcode einer Software-Anwendung als ein *Docker Image* gebaut und schließlich für weitere Anwendungsfälle in eine *Image Registry* hochgeladen wird. Damit der *Docker Daemon* auch innerhalb eines Containers verwendet werden kann, muss ein „*Docker-in-Docker*“-Konstrukt als zusätzlicher Service verwendet werden. Normalerweise ist der *Docker Daemon* außerhalb eines Containers unter „*/var/run/docker.sock*“ zu erreichen. Innerhalb eines Containers ohne weitere Konfigurationen allerdings nicht, daher muss der *Daemon* durch die TCP-Verbindung „*tcp://localhost:2375*“ über den zusätzlichen Service „*docker:dind*“ aufgerufen werden. Erst dies ermöglicht die Verwendung der *Docker Runtime* innerhalb eines Containers im *K8s*-Cluster. Um den Client dazu zu bringen, TCP zur Kontaktaufnahme zu verwenden, müssen im *Build*-Container die folgenden Umgebungsvariablen gesetzt sein: „*DOCKER_DRIVER*“, „*DOCKER_TLS_CERTDIR*“ und „*DOCKER_HOST*“.

Auflistung 7.16: *Build Stage* einer *CI / CD Pipeline* basierend auf *Docker Images*

```
build:
  stage: build
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_DRIVER: overlay2
    DOCKER_TLS_CERTDIR: ""
    DOCKER_HOST: tcp://localhost:2375
    IMAGE_NAME: "$REGISTRY_SERVER/$REGISTRY_PATH/$IMAGE_NAME:$IMAGE_TAG"
  before_script:
    - docker login -u "$REGISTRY_USER" -p "$REGISTRY_TOKEN" "$REGISTRY_SERVER"
  script:
```

```
    - docker build --pull -t "$IMAGE_NAME" .
    - docker push "$IMAGE_NAME"
```

Für den Bereitstellungsvorgang einer Container-Anwendung in einen *Kubernetes*-Cluster, kann beispielhaft der Ausschnitt einer *Deploy Stage* in der Auflistung 7.17 herangezogen werden. Hier wurde das besonders schlanke *Alpine Linux-Image* verwendet, in dem zu Beginn der *Deploy Stage* das Hilfswerkzeug *kubectl* nachinstalliert wird. In der eigentlichen Ausführungsphase werden mit Hilfe der *kubectl* ein *Namespace*, dann ein *Docker Registry-Secret* und schließlich eine vermeintlich ausgerollte Anwendungsinstanz entfernt und wieder neu bereitgestellt. Der dabei involvierte Cluster wird durch eine *K8s*-Konfigurationsdatei als *GitLab*-Umgebungsvariable (*K8S_CONFIG*) bestimmt. Die zu erstellenden *Kubernetes*-Ressourcen müssen in diesem Beispiel vor dem Ausführen der *Deploy Stage* in der Manifest-Datei „*k8s.yml*“ definiert vorliegen. In der Auflistung 8.3 ist der schematische Aufbau einer solchen YAML-Datei dargestellt.

Auflistung 7.17: *Deploy Stage* einer *CI / CD Pipeline* in einen *K8s*-Cluster

```
deploy:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --update --no-cache curl
    - curl -L "https://storage.googleapis.com/kubernetes-release/release/`curl -s https://st
    ↪  orage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl"
    ↪  -o /usr/local/bin/kubectl
    - chmod +x /usr/local/bin/kubectl
  script:
    - sed -i "s|<IMAGE>|$IMAGE_NAME|g" k8s.yml
    - sed -i "s|<REGISTRY>|$K8S_DOCKER_REGISTRY_SECRET|g" k8s.yml
    - kubectl create namespace "$K8S_NAMESPACE" --dry-run=client -o yaml | kubectl
    ↪  --kubeconfig "$K8S_CONFIG" apply -f -
    - kubectl -n "$K8S_NAMESPACE" create secret docker-registry
    ↪  "$K8S_DOCKER_REGISTRY_SECRET" --docker-server="$REGISTRY_SERVER"
    ↪  --docker-username="$REGISTRY_USER" --docker-password="$REGISTRY_TOKEN"
    ↪  --dry-run=client -o yaml | kubectl --kubeconfig "$K8S_CONFIG" apply -f -
    - kubectl --kubeconfig "$K8S_CONFIG" -n "$K8S_NAMESPACE" delete --ignore-not-found=true
    ↪  -f k8s.yml
    - kubectl --kubeconfig "$K8S_CONFIG" -n "$K8S_NAMESPACE" apply -f k8s.yml
```

Wie erkenntlich wird, können die Umgebungen der einzelnen Aufträge einer Pipeline individuell zusammengestellt werden. Der Vorteil an solchen containerisierten Vorgängen ist, dass immer das gleiche und definierte Ausführungsumfeld vorliegt und keine verbliebenen Artefakte bzw. Randbedingungen darauf Einfluss nehmen. In einer klassischen *CI / CD Pipeline* wäre vor einer Anwendungsbereitstellung zumindest noch eine *Test Stage* vorhanden, in der automatisiert Überprüfungen der Software durchgeführt werden und abhängig vom Ergebnis die *Deploy Stage* aktiviert wird. Außerdem könnte eine statische Code-Analyse als weitere *Stage* hinzugefügt werden, um die Sicherheitsaspekte in der Software-Entwicklung nicht zu vernachlässigen.

8 Evaluation

An dieser Stelle erfolgt die möglichst objektive Bewertung der Ergebnisse dieser Ausarbeitung. Zunächst werden im Abschnitt 8.1 die Sicherheitsanforderungen eines im Szenario neu angeforderten *Kubernetes*-Clusters überprüft. Dafür wird eine Anwendung herangezogen, die eine *K8s*-Cluster-Installation automatisch kontrolliert, indem sie in diesem Kontext anerkannte und dokumentierte Prüfungen der CIS *Kubernetes Benchmark* [CIS19] durchführt und anschließend einen Ergebnisbericht wiedergibt. Im zweiten Abschnitt 8.2 wird die realisierte *Kubernetes*-Infrastruktur bezogen auf die im Kapitel 4 definierten Anwendungsfälle kritisch bewertet und mit Hilfe potentieller Endkunden (WissenschaftlerInnen) ausprobiert. Zu realisierende *Use Cases* sind u. a. der „Bestellvorgang“ eines Kunden über das *Self-Service Portal* und die anschließende Bereitstellung von Container-Anwendungen in dem neu ausgerollten Cluster über das *Cluster Management* und einer *CI / CD Pipeline*. Alle involvierten Interaktionsplattformen werden hier noch einmal kurz bezogen auf die Verwendbarkeit und möglichen Einschränkungen thematisiert.

8.1 Sicherheit

Wie der Titel dieser Ausarbeitung bereits aussagt, steht vor allem die Sicherheit bzw. Informationssicherheit eines im *On-Premise*-Umfeld angesiedelten *Kubernetes*-Clusters im Fokus. Um eine möglichst objektive und auch umfassende Aussage darüber treffen zu können, wurde die Container-Anwendung *kube-bench* [Aqu20a] von dem *Cloud Native*-Sicherheitsunternehmen *Aqua Security* innerhalb eines neu bereitgestellten Clusters ausgeführt. Diese in der Programmiersprache *Go* geschriebene Anwendung überprüft, ob *K8s* mit all seinen Bestandteilen sicher eingesetzt wird, indem die in der CIS *Kubernetes Benchmark* [CIS19] dokumentierten Prüfungen durchgeführt werden. Das CIS [CIS20] ist eine gemeinnützige Organisation, welche weltweit anerkannte *Best Practices* zur Sicherung verschiedener IT-Systeme der Öffentlichkeit zur Verfügung stellt. Die herausgegebenen Standards werden von einer globalen Gemeinschaft von IT-Fachleuten ständig weiterentwickelt und durch angebotene Produkte sowie Dienstleistungen unterstützt, um sich gegen aufkommende Bedrohungen proaktiv zu schützen. Die Anwendung *kube-bench* bietet sich demnach an, in regelmäßigen Abständen automatisierte *Benchmark*-Tests in den vorhandenen *K8s*-Clustern durchzuführen. Schließlich können hierdurch bekannte Fehlkonfigurationen aufgedeckt und resultierende Angriffsvektoren vermieden werden.

Die Tabelle 8.1 gibt einen groben Überblick über die durchzuführenden Kontrollen (*Controls*) in der Version *1.5.0* der CIS *Kubernetes Benchmark* und deren bewertete Einteilung (*Scored / Not Scored*). Betrachtet werden die einzelnen Maßnahmen bezogen auf *Kubernetes* in der Version *1.15*. Insgesamt sind 122 Prüfungen vorhanden, von denen 92 in die Bewertung aufgenommen werden und 30 entsprechend nicht. Werden die Aussagen [Ran20a] des RKE-Herstellers *Rancher Labs* herangezogen, dann sind grundlegend nur die bewerteten Kontrollen (*Scored*) relevant, von denen wiederum 23 durch die charakteristische RKE-Architektur nicht durchführbar sind (*Not Applicable*). Das liegt vorwiegend daran, dass ein RKE-*Downstream*-Cluster vollständig durch die Container-Virtualisierung *Docker* realisiert wird und daher von einem ursprünglichen „*Vanilla Kubernetes*“ abweicht. Schließlich bleiben 69 einzelne Überprüfungen übrig, die die folgenden Bestandteile eines *Kubernetes*-Clusters betreffen: *Control Plane Components*, *etcd*, *Control Plane*

Configuration, *Worker Nodes* und wesentliche *Policies*. Eine vollständige Kontrollübersicht mit allen konkreten Einzelheiten liefert die im Anhang vorliegende Tabelle A.1.

Tabelle 8.1: Kontrollübersicht der CIS *Kubernetes Benchmark* (*v1.5.0*) bezogen auf einen *Rancher*-verwalteten RKE-Cluster [CIS19, Ran20a]

ID	***Controls*** ($\sum 122$)	***Scored*** ($\sum 92$)	***Not Scored*** ($\sum 30$)	***Not Applicable*** ($\sum 23$)
1	***Control Plane Components***	**58**	**7**	**14**
1.1	*Master Node Configuration Files*	19	2	14
1.2	*API Server*	30	5	0
1.3	*Controller Manager*	7	0	0
1.4	*Scheduler*	2	0	0
2	***etcd***	**6**	**1**	**0**
3	***Control Plane Configuration***	**1**	**2**	**0**
3.1	*Authentication and Authorization*	0	1	0
3.2	*Logging*	1	1	0
4	***Worker Nodes***	**20**	**3**	**9**
4.1	*Worker Node Configuration Files*	10	0	8
4.2	*Kubelet*	10	3	1
5	***Policies***	**7**	**17**	**0**
5.1	*RBAC and Service Accounts*	1	5	0
5.2	*Pod Security Policies*	4	5	0
5.3	*Network Policies and CNI*	1	1	0
5.4	*Secrets Management*	0	2	0
5.5	*Extensible Admission Control*	0	1	0
5.6	*General Policies*	1	3	0

Die Auflistung 8.1 zeigt, wie die Container-Anwendung *kube-bench* in einem *Kubernetes*-Cluster ausgeführt werden kann. Zuerst wird ein *Namespace* – hier ebenfalls „*kube-bench*“ – erstellt, dann wird die Anwendung als *K8s-Job* durch die im *Repository* zur Verfügung gestellte YAML-Datei „*job.yaml*“ im Cluster ausgerollt und letztendlich die entsprechende Protokolldatei des ausgeführten *Pods* mit dem Ergebnisbericht betrachtet. Für die Ausführung der Anwendung werden privilegierte Berechtigungen benötigt, die auch einen Host-Zugriff durch den Container mit einschließen. Daher wurde in *Rancher* ein separates Projekt angelegt, in dem die PSP „*unrestricted*“ gilt und der erstellte *Namespace* „*kube-bench*“ zugeordnet ist.

Auflistung 8.1: Ausführung von *kube-bench* in einem *Kubernetes*-Cluster [Aqu20a]

```
#!/bin/bash
# Create a namespace for kube-bench
kubectl [--kubeconfig ...] create namespace kube-bench

# Apply the kube-bench resource configuration
kubectl [--kubeconfig ...] -n kube-bench apply -f job.yaml

# Get all pods in the namespace kube-bench
kubectl [--kubeconfig ...] -n kube-bench get pods

# Print the logs for a pod / container
kubectl [--kubeconfig ...] -n kube-bench logs kube-bench-kphgm
```

Das Ergebnis eines Tool-Durchlaufs mit der CIS *Kubernetes Benchmark* in der Version *1.5.0* ist in einem gehärteten RKE-Cluster mit dem *Template* aus der Auflistung A.9 nahezu vollständig positiv (*PASS*). Einige Warnungen (*WARN*) tauchen unter dem fünften Punkt „*Policies*“ auf, die darauf hinweisen, dass die „*default*“-*Service Accounts* und der -*Namespace* innerhalb eines *K8s*-Clusters nicht verwendet werden sollten. Alle weiteren Kontrollen können durch den Einsatz von restriktiven RBAC-Richtlinien und den standardmäßig aktiven PSPs und *Network Policies* im gehärteten RKE-*Template* weitestgehend eingehalten werden. Dennoch tauchen auch nicht bestandene Prüfungen (*FAIL*) auf, die aber allesamt auf die RKE-Architektur zurückzuführen sind und somit vernachlässigt werden können. Es ist also ein durchweg positives Resultat durch die eingesetzten Sicherheitskonfigurationen festzuhalten. Durch die Verwendung des *Benchmark*-Tests konnte die Cluster-Sicherheit verifiziert werden.

In *Rancher* selbst gibt es auch die Möglichkeit, eine bereits integrierte Cluster-Sicherheitsüberprüfung mit vorkonfigurierten Profilen (*RKE-CIS-1.4 Permissive*, *RKE-CIS-1.4 Hardened*) zu starten. Allerdings wird hier noch die veraltete *Kubernetes Benchmark* in der Version *1.4* verwendet, die einen anderen Aufbau der durchzuführenden Kontrollen besitzt. Insgesamt gibt es im Profil „*RKE-CIS-1.4 Hardened*“ 97 bewertete (*Scored*) Überprüfungen, von denen 19 nicht durchzuführen sind (*Not Applicable*). Wird ein sogenannter „*CIS Scan*“ über die Benutzeroberfläche des *Cluster Managements* gestartet, dann werden unter der Verwendung des bereits erwähnten gehärteten RKE-*Templates* (s. Auflistung A.9) alle durchgeführten Kontrollen in dieser Version mit einem „*PASS*“ bestanden.

Für die „sichere“ Gestaltung eines *Kubernetes*-Clusters gehört es darüber hinaus dazu, die verteilten Knoten mit dem installierten Betriebssystem (*Ubuntu*) und die verwendete *Container Runtime* (*Docker*) entsprechend abzusichern. Das CIS bietet auch hierfür anerkannte *Benchmarks* an, die an dieser Stelle allerdings keine Berücksichtigung finden. Eine rudimentäre Härtung der Hosts wurde in der *Cloud-init*-Konfiguration (s. Auflistung A.8) verankert, die u. a. den SSH-*Daemon* und eine *Firewall* mit einigen *K8s*-spezifischen Richtlinien aktiviert. Des Weiteren werden die stetig aktualisierten *Ubuntu Cloud Images* verwendet, die in der Form von OVF-*Templates* ausgeliefert und deshalb nur noch in die *vSphere*-Umgebung importiert werden müssen. Im späteren produktiven Einsatz wird es zudem wichtig die Integration eines Konfigurationsmanagement, wie etwa *Puppet*, in diesem Umfeld zu aktivieren, um Betriebssystemaktualisierungen und sich verändernde Konfigurationen auch in bereits ausgerollten *K8s*-Knoten anwenden zu können. Auch die dann anstehende Integration der im Rechenzentrum vorhandenen *Monitoring*- und *Logging*-Werkzeuge gewährleistet eine bessere Überwachung der Host-Sicherheit.

8.2 Verwendbarkeit

Neben der Sicherheitsbetrachtung eines *Kubernetes*-Clusters kommt es auch auf die spätere produktive Verwendbarkeit der realisierten Infrastruktur durch die Nutzerschaft an. Die zuvor definierten funktionalen Anforderungen sollen im Zusammenspiel durch die einzelnen Interaktionsplattformen umzusetzen sein und mögliche Einschränkungen, bspw. durch die konfigurierten Sicherheitsmaßnahmen oder der eingesetzten Plattformen selbst, in Erfahrung gebracht werden. Für eine systematische Bewertung der Infrastruktur wurde eine interaktive Umfrage mit Hilfe von *Google Forms* [Goo20a] entwickelt und AWI-intern durch ausgewählte Nutzer in einem virtuellen Treffen gemeinschaftlich durchgeführt. Eine Kopie des vollständigen Umfrageformulars ist im Anhang unter Abschnitt A.7 abgedruckt. Das Formular berücksichtigt alle vier wesentlichen Bestandteile (*Self-Service Portal*, *Cluster Management*, *Image Registry*, *CI / CD Pipeline*) des in dieser Ausarbeitung aufgebauten Szenarios und teilt hierzu die Umfrage nacheinander in Aufgaben

und anschließenden Fragestellungen auf. Die jeweils vorher platzierten kleinen Arbeitsaufträge, sollen eine bewusste Auseinandersetzung mit den verschachtelten Plattformen bewirken.

Die ersten beiden Abschnitte bis zur sechsten Frage der Umfrage informieren über die zu betrachtende Infrastruktur und beinhalten allgemeine Fragen zum Kenntnisstand von Container-Technologien und den vielleicht schon vorher bekannten Software-Produkten (*VMware vRealize Suite*, *Rancher*, *Harbor*, *GitLab*). In den nachfolgenden Abschnitten werden dann die einzelnen Plattformen betrachtet, die auch an dieser Stelle jeweils kurz aufgeführt werden.

Im dritten Abschnitt der Umfrage wird der Ressourcen-Katalog des *Self-Service Portals* aufgerufen, ein *K8s*-Cluster angefordert und daraufhin weitere Ressourcen-spezifische Operationen (*Day-2 Operations*) durchgeführt. In der Abbildung 8.1 ist die Repräsentation (*Deployment*) eines angeforderten *Kubernetes*-Clusters in der *vRA*-Oberfläche dargestellt, auf den weitere Operationen (s. Tabelle 7.3) angewendet werden können. Der „Bestellvorgang“ einer solchen Cluster-Ressource muss hierfür im Vorfeld, wie bereits in der Abbildung 7.4 zu sehen ist, durch das Abschicken einer Anfrage abgewickelt und die automatisierte Bereitstellung im Hintergrund vollzogen sein.

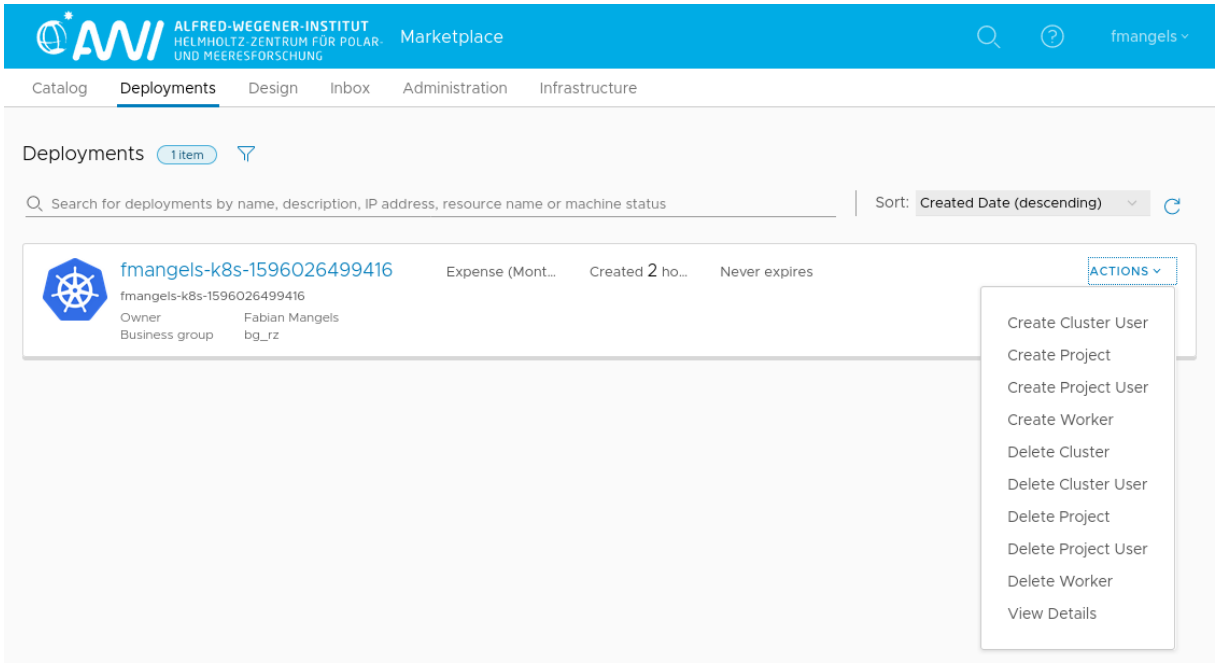


Abbildung 8.1: Anwenden von weiteren Operationen auf ein *K8s*-Cluster-*Deployment* im *Self-Service Portal*

Der darauffolgende vierte Abschnitt befasst sich mit dem *Cluster Management Rancher*. Vorwiegend sollen hier Aufgaben bezogen auf die Interaktion mit dem zuvor angeforderten Cluster ausgeführt werden. Als Beispiel kann die Abbildung 8.2 herangezogen werden, in der das *Dashboard* eines *Rancher*-verwalteten Clusters zu sehen ist. Für eine native Kommunikation mit dem Cluster ist die auf dieser Seite aufzurufende Konfigurationsdatei für das Tool *kubectl* durch den *Button* „*Kubeconfig File*“ besonders nützlich oder eben die vollständig im Webbrowser stattfindende *Session* durch Betätigen des *Buttons* „*Launch kubectl*“. Außerdem soll über die grafische Benutzeroberfläche eine Anwendung bereitgestellt werden. In der Abbildung 8.3 ist eine Übersicht bereitgestellter *Workloads* – u. a. auch die Bestandteile des weiter unten aufgeführten Minimalbeispiels über die *CI / CD Pipeline* und ein *Nginx*-Webserver (s. Auflistung 8.3) – innerhalb einer Projekt-Ressource der *Rancher*-Plattform dargestellt. Als kleine Test-Anwendung bietet sich hier das *Docker Image* „*hello-world*“ an, welches als *Workload Type* „*Job*“ über den „*Deploy*“-*Button* im *K8s*-Cluster bereitgestellt werden kann. Anschließend lassen sich in der

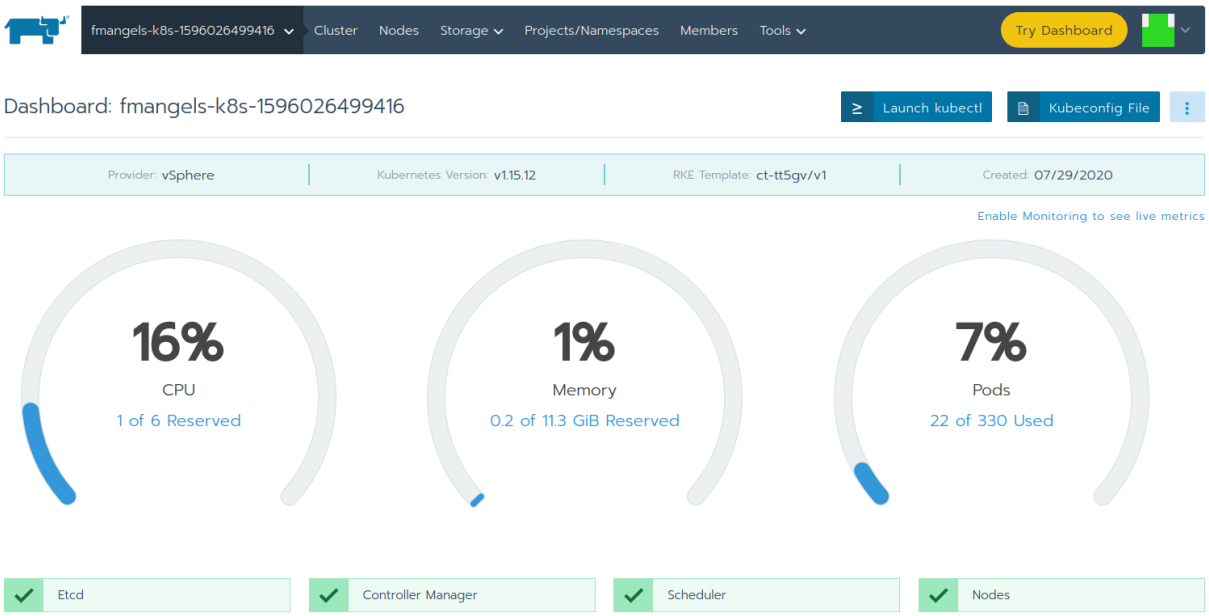


Abbildung 8.2: *Dashboard* eines im *Cluster Management Rancher* verwalteten *K8s*-Clusters

Rancher-Benutzeroberfläche die *Logging*-Informationen oder auch das automatisch angelegte *Kubernetes*-YAML-Manifest der Test-Anwendung durch die erweiterten *Workload*-Optionen betrachten. Diese kleinen Aufgaben helfen, auch unerfahreneren Nutzern einen ersten Einblick im Umgang mit *K8s*-Clustern zu bekommen.

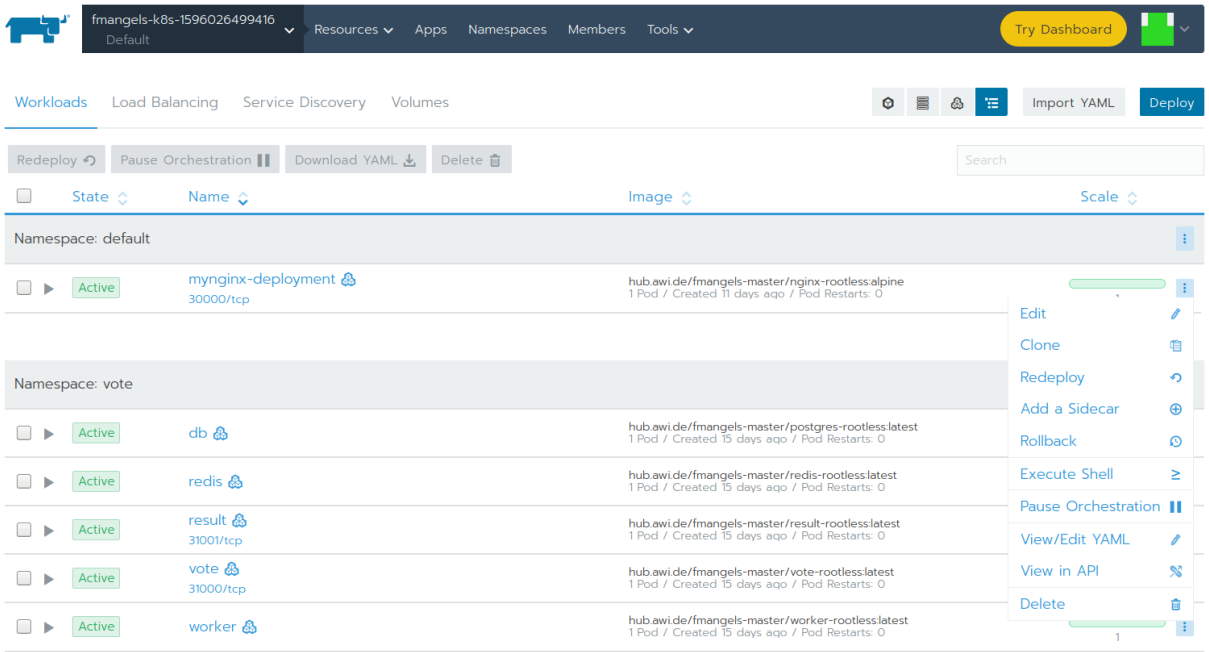


Abbildung 8.3: Übersicht bereitgestellter *Workloads* im *Cluster Management Rancher*

Die *Image Registry Harbor* wird im fünften Abschnitt thematisiert. Die Abbildung 8.4 stellt hierfür einen Einblick auf die *Harbor*-Benutzeroberfläche bereit. Konkret ist das *Repository* „*worker-rootless*“ – ein Bestandteil des weiter unten aufgeführten Minimalbeispiels – mit einem hochgeladenen *Image* dargestellt, welches das „*latest*“-*Tag* verwendet. Zudem werden durch einen

automatisch stattgefundenen CVE-*Scan* Schwachstellen des *Images* grafisch bewertet und in einem weiteren Untermenü entsprechende Handlungsempfehlungen gegeben. Die Arbeitsaufträge der Umfrage enthalten auf der Kommandozeilenebene u. a. das Bauen bzw. richtige *Taggen* und Hochladen von *Docker Images* in die *Private Registry* des AWIs sowie den Umgang und das Erforschen der grafischen Benutzeroberfläche im Webbrowser der *Registry*. Außerdem sollen signierte Images durch das Setzen zweier Umgebungsvariablen hochgeladen werden, um auch die automatische Verwendbarkeit des *Docker Content Trusts* durch *Harbors Notary*-Integration aufzuzeigen.

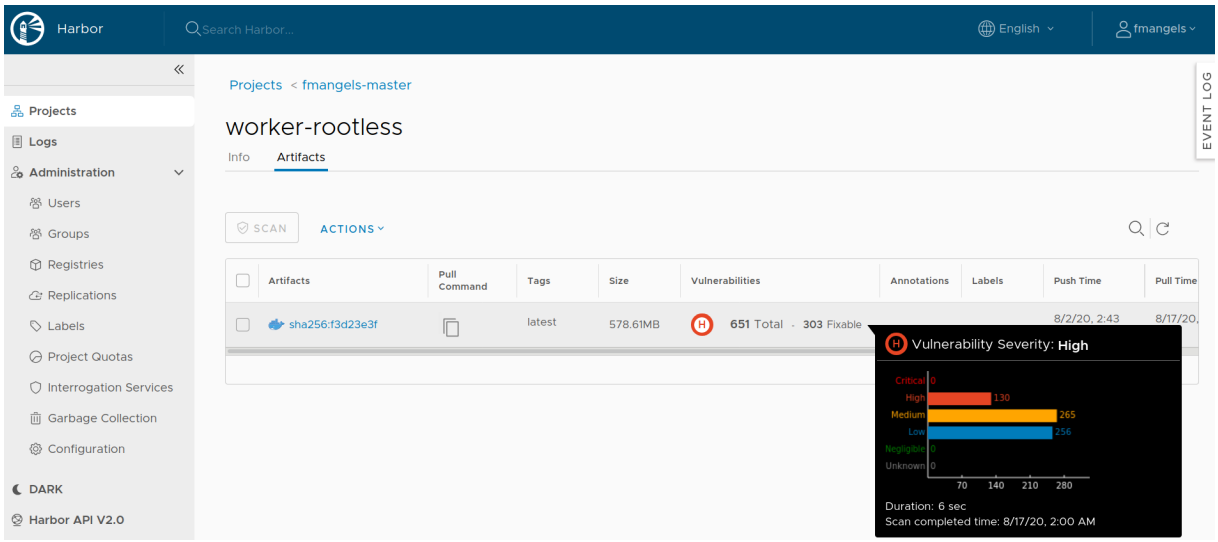


Abbildung 8.4: Schwachstellenanalyse eines *Docker Images* in der *Registry Harbor*

Im sechsten und letzten Abschnitt der Umfrage, wird sich der *DevOps*-Plattform *GitLab* und den integrierten *CI / CD Pipelines* gewidmet. Zu den Arbeitsaufträgen zählen u. a. die Erstellung eines neuen *GitLab*-Projektes, die Installation und Konfiguration eines *GitLab Runners* (s. Auflistung 7.15) innerhalb des zuvor bereitgestellten *K8s*-Clusters, das Hinzufügen von Quellcode eines Minimalbeispiels und die Konfiguration einer hierzu aufbauenden *CI / CD Pipeline* bestehend aus mindestens einer *Build* und *Deploy Stage*. Für die Bereitstellung einer *Microservice*-basierenden Container-Anwendung, wurde an dieser Stelle die verteilte Abstimmungsanwendung „*Voting App*“ [Doc20g] herangezogen, die aus mehreren *Docker*-Containern aufgebaut ist. Die Wahl der Anwendung wurde aufgrund des heterogenen Aufbaues und des Vorhandenseins der Anwendungsdefinition für *Kubernetes* getroffen. Außerdem ist der Quellcode des Projektes öffentlich zugänglich und eignet sich daher für Vergleichszwecke.

In der Abbildung 8.5 sind die einzelnen Komponenten bzw. *Microservices* des Minimalbeispiels „*Voting App*“ mit ihren Beziehungen untereinander dargestellt. Nach außen betrachtet stellt die Anwendung zwei Weboberflächen bereit, auf denen eine Abstimmung über „Hund“ oder „Katze“ (*dog* oder *cat*) stattfinden kann bzw. die Auswertung ersichtlich wird. Das *Frontend* für die Abstimmung zwischen den zwei Optionen wird dabei über eine *Python*-Anwendung bereitgestellt (blau, *voting-app*). Durch eine *Redis*-Datenstruktur werden daraufhin die Stimmen in einer Warteschlange gesammelt (grün, *redis*). Der nachgeschaltete *.NET*-Service (orange, *worker*) holt die Stimmabgaben ab und legt sie aufbereitet in eine *POSTgreSQL*-Datenbank wieder ab (gelb, *db*). Abschließend wird in einer *Node.js*-Webanwendung das Ergebnis der Abstimmung in Echtzeit angezeigt (rot, *result-app*). [Doc20g]

Die verwendete *CI / CD Pipeline* des Minimalbeispiels wird aufgrund des Umfangs im Anhang in der Auflistung A.20 aufgeführt. Sie besteht aus den drei *Stages* „*build_org*“, „*build_rootless*“

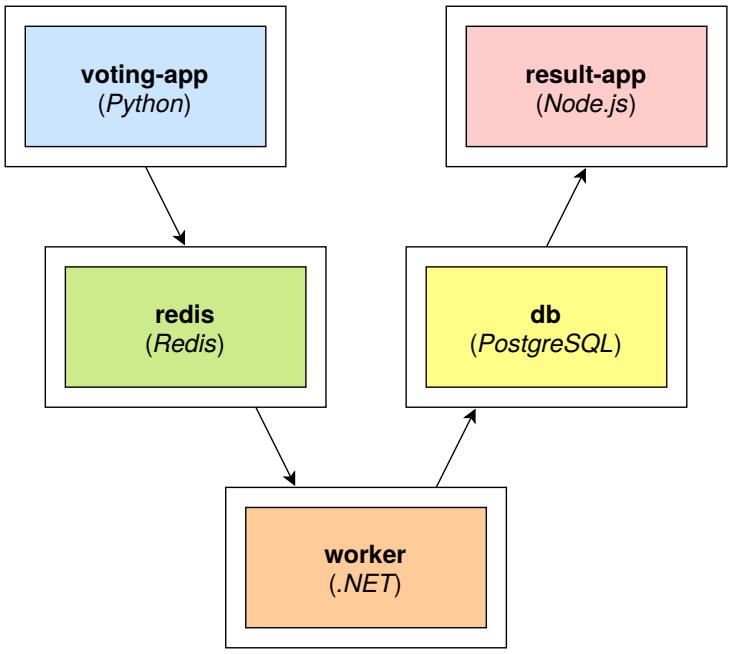


Abbildung 8.5: Komponenten des Minimalbeispiels „*Voting App*“ [Doc20g]

sowie „*deploy*“ und lässt sich durch die Verwendung von Umgebungsvariablen parametrisieren. Im ersten Schritt werden *Docker Images* basierend auf dem originalen Quellcode des *Git-Repositories* gebaut und in die AWI-eigene *Image Registry* hochgeladen. Für dieses Vorhaben wird der schon erwähnte „*Docker-in-Docker*“-Service beansprucht, welcher den *Docker Daemon* innerhalb eines *Docker*-Containers zugänglich macht. Der nächste Schritt verwendet die soeben erstellten *Images* innerhalb spezifischer *Dockerfiles*, wie der Ausschnitt der Auflistung 8.2 zeigt. In diesen *Image*-Bauplänen wird die Ausführung der einzelnen Anwendungsbestandteile ohne die Verwendung des *Root*-Nutzerkontos ermöglicht. Die Anpassung des ausführenden Benutzerkontos ist hier notwendig, da die vorherrschenden PSPs in den ausgerollten *K8s*-Clustern den privilegierten *Root*-Benutzer nicht zulassen; ggf. müssen auch die zu veröffentlichen Ports und der eigentliche Befehlsaufruf des Containers umstrukturiert werden. Letztendlich werden in dieser Phase die *Docker Images* unter der Verwendung eines sogenannten „*Robot Accounts*“ gebaut und in separate *Repositories* der *Image Registry* hochgeladen.

Auflistung 8.2: Änderung des ausführenden Nutzerkontos in einem *Dockerfile* (*Alpine Linux*)

```
...
ENV USER=nonroot
ENV UID=12345

RUN adduser -S -D -H -u "$UID" "$USER"

USER $UID
...
```

Die letzte *Stage* beinhaltet die Bereitstellung des Minimalbeispiels „*Voting App*“ in einen *Kubernetes*-Cluster und erfolgt nativ mit Hilfe des Tools *kubectl* in Verbindung mit dem aus *Rancher* stammenden „*Kubeconfig File*“. Zuerst wird der *Namespace* „*vote*“, dann ein *Registry-Secret* und abschließend die einzelnen *K8s*-Ressourcen der Container-Anwendung entfernt und schließlich wieder neu hinzugefügt. Der strukturierte Aufbau eines dabei ähnlich verwendeten *Kubernetes-*

Manifests eines *Nginx*-Webservers, kann der Auflistung 8.3 entnommen werden. Alle *Microservices* der „*Voting App*“ werden durch einen solchen *Service* für die Kommunikation innerhalb sowie außerhalb des Clusters und durch ein solches *Deployment* abgebildet. Letzteres ist eine konkrete Beschreibung darüber, wie viele und welche Container zu jedem Zeitpunkt in der verteilten Umgebung vorliegen sollen. Zu beachten ist, dass *Services* wenn möglich immer vor den jeweiligen *Pods* erstellt werden sollten, die auf diese zugreifen wollen. Die *Template*-Variablen „*<IMAGE>*“ und „*<REGISTRY>*“ können hier durch eine *String*-Substitution (s. Auflistung 7.17) ersetzt werden. Für das Minimalbeispiel wurde für jede einzelne Container-Anwendung eine separate YAML-Datei mit unterschiedlichen Ports erstellt und für die Bereitstellung durch einen Schleifen-Vorgang abgearbeitet. Das Bauen der *Docker Images* im Vorfeld erfolgt dazu äquivalent. Für Entwickler, die bereits ihre Anwendungen mit *Docker Compose*-Dateien definieren, gibt es das als Konverter agierende Tool *Kompose* [Lie19, vlg. S. 750]. Es wandelt *Docker Compose*-Definitionen in entsprechende *K8s*-Repräsentationen um; ggf. sind aber noch Anpassungen vorzunehmen.

Auflistung 8.3: Definieren von Cluster-Ressourcen durch ein *Kubernetes*-Manifest (*Nginx*-Webserver)

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: mynginx
  name: mynginx-service
  namespace: default
spec:
  type: NodePort
  ports:
  - name: "mynginx-service"
    port: 5000
    targetPort: 8080
    nodePort: 30000
  selector:
    app: mynginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: mynginx
  name: mynginx-deployment
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mynginx
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: mynginx
    spec:
      containers:
      - image: <IMAGE>
```

```
        imagePullPolicy: Always
43      name: mynginx
        ports:
45      - containerPort: 8080
          name: mynginx
47    imagePullSecrets:
      - name: <REGISTRY>
49    restartPolicy: Always
```

Mit diesem letzten Abschnitt ist die interaktive Umfrage absolviert und die Bewertung des Ergebnisses kann stattfinden. Zu aller erst muss gesagt werden, dass die durchgeführte *Google Forms*-Umfrage mit sechs Teilnehmern aufgrund der Stichprobengröße (Quantität) nicht repräsentativ ist, dennoch lassen sich einige qualitative Aussagen schlussfolgern:

Allgemein betrachtet, brauchen die Teilnehmer mehr Zeit, sich mit allen Interaktionsplattformen intensiver auseinanderzusetzen. Die Mehrheit der Beteiligten am AWI befinden sich momentan im Übergang bzw. stehen kurz davor mit ihren Anwendungen von einem einzelnen *Docker*-Host-System auf einen verteilten *Kubernetes*-Cluster zu wechseln. Weitere *Workshops* bzgl. der *K8s*-Domäne sollten hier angesiedelt werden. Daher waren einige Befragte mit den gestellten Aufgaben und Fragestellungen teilweise überfordert. Mitunter werden zu Plattform-spezifische Fragen gestellt, die zu diesem Zeitpunkt nicht vollumfänglich erfasst werden konnten, z. B. sind weitere Informationen zu *Day-2 Operations*, PSPs, CVE-*Scannern*, *Docker Content Trust* und *GitLab Runner* notwendig gewesen. Des Weiteren war die Teilnahme wegen den anhaltenden *Corona*-Beschränkungen leider nur virtuell möglich – eine Betreuung und ein Eingreifen vor Ort wäre bei den Aufgaben vermutlich besser gewesen. Eine Wiederholung der Umfrage bei einem fortgeschritteneren Kenntnisstand sollte unbedingt durchgeführt werden.

Gleichwohl gab es auch eine kritische Auseinandersetzung mit den definierten Anwendungsfällen und den getätigten Sicherheitskonfigurationen aus der Nutzersicht. Die Verwendbarkeit war durch die standardmäßig vorherrschenden PSPs und *Network Policies* in den *Kubernetes*-Clustern stark eingeschränkt. Bereits vorhandene Container-Anwendungen konnten aufgrund von nur zulässigen unprivilegierten Containern und Nutzerkonten – kein *Root* – nicht bereitgestellt werden. Allerdings gibt es hierfür auch Zuspruch, da in der Vergangenheit Sicherheitskonfigurationen für die Container-Technologie nicht thematisiert wurden. Auch die ausschließliche Verwendung von einer *Private Registry* wäre durchweg akzeptabel. Die meisten Anwendungen werden vor der Bereitstellung selber gebaut und bereits in eine *Image Registry* abgelegt, daher wäre diese noch nicht aktive Konfiguration vertretbar. Die zusätzlichen *Harbor*-Funktionen wie die eines CVE-*Scanners* und die Umsetzung von *Docker Content Trust* – sprich die digitale Signierung von *Docker Images* – wurden als unterstützend und wichtig aufgenommen. Die begrenzten RBAC-Berechtigungen im *Cluster Management Rancher* waren für die Bereitstellung von Anwendungen für die Teilnehmer ausreichend. Nur die manuelle Installation der *GitLab Runner* beanspruchte die zusätzliche Vergabe von Berechtigungen, um in einem *Namespace* erhöhte Ausführungsberechtigungen zu setzen und einen *Service Account* anzulegen – dies war bereits im Vorfeld abzusehen. Allein vom Sicherheitsgewinn und der Ersparnis des manuellen Aufwands wird die Integration von *Shared Runnern* im Kontext der *CI / CD Pipelines* der *DevOps*-Plattform *GitLab* angestrebt und auch befürwortet. Außerdem wurde angemerkt, dass redundante Handlungsmöglichkeiten im *Self-Service Portal* und im *Cluster Management* vorhanden sind. Es ist hier zu überlegen, nur die Bestellung einer neuen *Kubernetes*-Ressource sowie zusätzlicher *Worker*-Knoten und die spätere Abbestellung dieser Komponenten über das *Self-Service Portal* zuzulassen. Die darüber hinaus gehende Verwaltung eines solchen Clusters würde dann nur mit Hilfe von *Rancher* erfolgen.

9 Fazit und Ausblick

Im Wesentlichen hat sich diese Ausarbeitung mit dem vielschichtigen Themenkomplex der Container-Technologien sowie der Bereitstellung von *On-Premises*-Clustern und den dort platzierten Software-Anwendungen befasst. Von besonderer Bedeutung waren dabei die sichere und möglichst automatisierbare Gestaltung aller notwendigen Interaktionsplattformen (*Image Registry*, *Cluster Management*, *Self-Service Portal*, *CI / CD Pipelines*), um eine skalierbare Container-Infrastruktur im Rechenzentrum des AWIs und innerhalb des HIFIS-Projektes zu entwerfen.

In den ersten Kapiteln wurden in diesem Zusammenhang stehende Grundlagen vermittelt und verwendete Komponenten der Infrastruktur erläutert (s. Kapitel 2, Kapitel 3). Es hat sich gezeigt, dass in der Software-Architektur die Vorteile von *Microservices* gegenüber Monolithen in vielerlei Hinsicht überwiegen. Dieser *Microservice*-Architekturstil wird in der Software-Bereitstellung wiederum durch die Container-Virtualisierung fortgeführt. Die hierfür eingesetzten Plattformen, welche grundsätzlich auf dem *CaaS*-Modell basieren, entwickeln sich von einfachen *Single Node* Container-Systemen hin zu skalierbaren Container-Clustern. In Folge der notwendigen Automatisierungsprozesse bezogen auf die zunehmende Menge an zu verwaltenden Containern, wird sich letztendlich der Orchestrierung als Dienstkomposition bedient. Auch die Schaffung einer konsistenten Bereitstellungsplattform für *Cloud Native*-Anwendungen und die vermehrten *DevOps*-Prozesse in der Software-Entwicklung sind an dieser Stelle noch einmal zu erwähnen. Wahrhaftig realisiert werden diese aufgezeigten Aspekte durch den Einsatz von *Kubernetes* und *Docker* in einem Cluster-Verbund. Vor dem eigentlichen Hauptteil der Ausarbeitung wurden alle grundlegenden Anforderungen und Anwendungsfälle an die zu realisierende Infrastruktur im Kapitel 4 zusammengetragen. Das anschließende Kapitel 5 enthält darüber hinaus weiter gefasste Darstellungen von Ausarbeitungen, die im Zusammenhang mit dem betrachteten Themenkomplex stehen. Diese verwandten Arbeiten halfen u. a. bei der Entscheidungsfindung der konkret zu verwendenden Komponenten.

Die Analyse der Sicherheit bzw. Informationssicherheit einer skalierbaren Container-Infrastruktur erfolgte im Kapitel 6 unter der Zuhilfenahme des vom BSI jährlich herausgegebenen IT-Grundschutz-Kompendiums [BSI20a] und des abstrakten 4-Schichtenmodells „*The 4C's of Cloud Native Security*“ [Kub20g]. Es wurde festgestellt, dass sich der noch in der Entwurfsphase befindende Baustein „*SYS.1.6: Container*“ [BSI20b] des IT-Grundschutzes für die Umsetzung eines Sicherheitskonzeptes mit verschieden starken Anforderungsebenen im Kontext der Container-Technologien eignet. Weiterführende Maßnahmen für Rechenzentren, in denen *Kubernetes*-Cluster betrieben werden, konnten systematisch den einzelnen Schichten (*Cloud / Co-Lo / Corporate Datacenter*, *Cluster*, *Container*, *Code*) des Modells der *Cloud Native Security* zugeordnet werden. Des Weiteren wurden im zweiten Abschnitt der Analyse die Charakteristiken der Automatisierung einer solchen Umgebung betrachtet. Es wurde die Entwicklung der Rechenzentren hin zu SDDCs in Folge der stetig anhaltenden digitalen Transformation thematisiert und die damit einhergehende maximale Automation sowie größere Autonomie zukünftiger Software-Systeme, wie es auch bei *K8s* der Fall ist, herausgestellt.

Die praktische Umsetzung der theoretisch erarbeiteten Konzepte fand schließlich im Kapitel 7 und vollständig in der zugrunde liegenden *VMware vSphere*-Virtualisierungsumgebung statt. Die

dafür konkret gewählten Interaktionsplattformen sind die folgenden: *Harbor* als *Image Registry*, *Rancher* mit RKE als *Cluster Management*, die *VMware vRealize Suite* (*vRA*, *vRO*) als *Self-Service Portal* und *GitLab* als *CI / CD Pipeline*. *Harbor* und im Speziellen *Rancher*, das später die von den Nutzern angeforderten und automatisiert bereitgestellten RKE-*Downstream*-Cluster verwaltet, wurden komplett neu aufgebaut. Die Bestandteile der *vRealize Suite* und die *DevOps*-Plattform *GitLab* waren bereits in produktiver Verwendung und wurden entsprechend in das Szenario eingebettet. Schlussendlich wurde ein *Kubernetes*-Umfeld mit den benannten Infrastruktur-Komponenten für den Wissenschaftsbetrieb am AWI und den organisationsübergreifenden Austausch im HIFIS-Projekt erfolgreich realisiert.

Eine möglichst objektive Bewertung der Infrastruktur-Gestaltung unter den Aspekten der Sicherheit und Verwendbarkeit erfolgte anschließend im Kapitel 8. Die Sicherheitsbewertung unter der Zuhilfenahme der anerkannten CIS *Kubernetes Benchmark* [CIS19] und der darauf aufbauenden Container-Anwendung *kube-bench* [Aqu20a] wurde zufriedenstellend bestanden. Bei einigen Kontrollen (s. Abschnitt A.6) mussten allerdings Ausnahmen aufgrund der Container-basierten Architektur der *K8s*-Distribution RKE zugelassen werden; der Hersteller *Rancher Labs* gibt hierzu auch eine Einschätzung ab [Ran20a]. Es wurden u. a. *Cloud-init*-Konfigurationen (s. Auflistung A.8) für die als *Nodes* agierenden VM-Hosts und gehärtete *Templates* für die RKE-Cluster (s. Auflistung A.9) eingesetzt, um eine grundlegende Informationssicherheit zu gewähren. Eine solche Sicherheitsbetrachtung ist vor allem dann essentiell, falls Container-Anwendungen von mehreren verschiedenen Benutzern mit konkurrierenden Interessen innerhalb eines *Kubernetes*-Clusters bereitgestellt werden sollen. Momentan wird jedem Anforderer ein eigener *K8s*-Cluster automatisch zugeteilt. Hier findet also schon eine Segmentierung auf einer höheren Ebene statt, indem eigene VMs nur für diesen einen Cluster provisioniert werden. Im zweiten Teil der Evaluation erfolgte eine kritische Bewertung aus der Nutzersicht durch eine interaktive *Google Forms*-Umfrage (s. Abschnitt A.7), welche Arbeitsaufträge und Fragestellungen zu jeder einzelnen Interaktionsplattform aufweist. Durch die niedrige Stichprobengröße ist die getätigte Umfrage nicht repräsentativ, dennoch konnten einige qualitative Aussagen der Nutzerschaft ausgewertet und für den Ausblick vorgemerkt werden.

Bei einer kritischen Betrachtungsweise sind *Microservices* und Container-Cluster noch kein Allheilmittel für alle Einsatzszenarien in modernen Rechenzentren. Wegen der vielen Vorteile dieser leistungsstarken Umgebungen ist es aber durchaus der richtige Weg Container-Technologien verstärkt einzusetzen. Die Vielfalt und teilweise hohe Komplexität in diesem Bereich nehmen nach wie vor mit hoher Geschwindigkeit zu, deshalb werden sich diese Umgebungen stetig umstrukturieren, bevor die Zuverlässigkeit und eine langfristige Stabilität weiter in den Fokus rücken. Dahingegen wurde mit *Kubernetes* in dieser Ausarbeitung eine relativ stabile Container-Plattform eingesetzt, die viel Potential bietet und in der realisierten Infrastruktur im Rechenzentrum des AWIs ihre Vorzüge ausspielen kann. Im Wissenschaftsbetrieb ist eine *K8s*-Anwendungsbereitstellung mit ihren vorteilhaften Funktionen bspw. mit der am AWI entwickelten Software *webODV* [web19] zukünftig denkbar. Diese Webapplikation basiert auf dem *Open Data Viewer* (ODV) und bietet Online-Dienste an, um Ozean- und andere Umweltdaten zu analysieren und zu visualisieren.

Für eine produktive Nutzung der entworfenen *K8s*-Infrastruktur sind darüber hinaus noch weitere Arbeitsschritte notwendig: Die aktuell konfigurierten Sicherheitsmaßnahmen sind im Grunde genommen ausreichend – natürlich findet eine stetige Weiterentwicklung in diesem Bereich statt und daher sollten fortlaufende Anpassungen nach dem Stand der Technik durchgeführt werden. In diesem Kontext ist noch darüber nachzudenken standardmäßige Ressourcenbegrenzungen innerhalb vorhandener *Namespaces* oder bezogen auf bereitgestellte *Pods* einzuführen, um auch hier mögliche Angriffsvektoren zu vermeiden. Für den Betrieb eines verteilten Clusters mit *Microservice*-Anwendungen empfiehlt es sich zudem das sogenannte *Chaos Engineering* bzw. *Testing* heranzuziehen [AD19, vgl. S. 118ff]. Mit den Tools *chaoskube* und *kube-monkey* [AD19,

vgl. S. 120f.] stehen zwei Anwendungen bereit, mit denen *Pods* zufällig innerhalb eines *Kubernetes*-Clusters beendet werden. Mit Hilfe des *Chaos Testings* können dann aussagekräftigere Entscheidungen bezogen auf die Hochverfügbarkeit der eigentlichen Cluster-Komponenten, als auch der dort befindlichen Anwendungen getroffen werden. Aus der Sicht der *Usability* können noch einige zusätzliche Verbesserungen integriert werden. Die Erprobung der *K8s*-Funktion „*Rolling Update*“ in den *CI / CD Pipelines*, mit der Anwendungen ohne eine Ausfallzeit aktualisiert werden können oder die Verwendung von *Git*-SHA-*Tags* (*Secure Hash Algorithm*) bei den *Image Build*-Vorgängen [AD19, vgl. S. 249], sind hier zu nennen. Des Weiteren sollte die Speicheranbindung (*Storage*) für Container-Anwendungen in den Clustern getestet und die schon erwähnten *Shared Runner* in der *GitLab*-Plattform den jeweiligen Projekten zur Verfügung gestellt werden. Außerdem kann sich der *Load Balancer*-Problematik für die RKE-*Downstream*-Cluster noch einmal angenommen werden, damit bereitgestellte Anwendungen auch über eigene IP-Adressen und spezifizierte Ports außerhalb des Clusters aufzurufen sind [Kub20e]. Momentan können hier in den automatisierten *Workflows* nur „*NodePorts*“ angeboten werden. Es gibt bereits eine Lösung durch *MetalLB* [Met20], die allerdings noch manuell mit einem zugeteilten IP-Adressenbereich versehen werden muss. Um sonstige Einschränkungen der gewählten Interaktionsplattformen zu erkennen, steht ferner eine erneute Durchführung der interaktiven *Google Forms*-Umfrage aus.

Zukünftig können im Umfeld dieser Ausarbeitung noch weitere Themen interessant werden: So ist es denkbar, wenn genügend Erfahrungen mit *Kubernetes* gesammelt worden sind, föderierte oder geografisch verteilte *K8s*-Cluster im HIFIS-Kontext aufzubauen. Der abstrakt gehaltene Abschnitt A.8 beschäftigt sich dahingehend mit den Möglichkeiten einer *KubeFed* (*Kubernetes Cluster Federation*). In diesem organisationsübergreifenden Zusammenhang wird auch eine Mikrosegmentierung mit Hilfe von NSX [WAG+18, vgl. S. 477ff] in einem von *VMware* geprägten SDDC bzw. einer *Hybrid Cloud* immer wichtiger. Hierdurch können IKT-Netzwerke in Rechenzentren sowohl intern als auch extern durch die Bestandteile der *Network Virtualization* weitergehend abgesichert werden. Überdies gibt es bezogen auf die Informationssicherheit eines *K8s*-Clusters sogar die Option, Anwendungen in VMs anstatt üblicherweise in Containern ausführen zu lassen [Luk18, vgl. S. 602], bspw. ermöglicht der Einsatz von *Frakti* als *Container Runtime* die direkte Ausführung von Container-*Images* über einen *Hypervisor*. Jeder Container verwendet daraufhin seinen eigenen Kernel, was letztendlich zu einer besseren Isolation zwischen den Containern führt. Ebenfalls sollte die unprivilegierte Verwendung des *Docker Daemons* durch *kaniko* [Goo20b] für *Docker Build*-Vorgänge innerhalb von *Kubernetes* erprobt werden. Im HPC-Wissenschaftsbereich wird bei der Container-Virtualisierung verstärkt auf *Singularity* als *Container Runtime* gesetzt [God19]. Interessant wird hier dann auch die Integration in einem *K8s*-Cluster samt der Ausführung von Anwendungen auf entsprechend leistungsstärkerer Hardware. Eine umfassendere Betrachtung könnte zudem die Übertragbarkeit der entworfenen Umgebung auf andere Infrastrukturen beinhalten. Der wesentliche Fokus würde dabei z. B. auf der kompletten Umsetzung mit *Open Source*-Lösungen liegen. Die kommerzielle Virtualisierungsumgebung *vSphere* mit ihren *VMware*-Bestandteilen (*vRA*, *vRO*), müsste demzufolge ersetzt werden; alle anderen eingesetzten Komponenten verwirklichen bereits den *Open Source*-Gedanken. Vorstellbar wäre dies mit der Virtualisierungstechnologie KVM (*Kernel-based Virtual Machine*) [Red20b] und entsprechender Software-Lösungen für das VM-Management im Cluster-Verbund. Vermutlich würde sich in diesem Szenario der manuelle Aufwand erhöhen, da noch keine Einbettung im *Cluster Management Rancher* existiert und ggf. ein *Self-Service Portal* neu entwickelt werden müsste. Zu guter Letzt sollte auch die fortlaufende native *Kubernetes*-Integration innerhalb von *vSphere* ab der Version *7.0* [VMw20c] im Auge behalten werden. Direkt in der *VMware*-Virtualisierungsumgebung sollen dann *K8s*-Ressourcen in der Form von sogenannten *vSphere-Pods* bereitgestellt werden können. Auch an dieser richtungsweisenden Entwicklung des *Global Players* im Bereich der (Server-)Virtualisierung und des *Cloud Computings* ist zu erkennen, dass die Schaffung von Container-basierten Infrastrukturen – insbesondere *Kubernetes* – die Zukunft sein wird.

Literaturverzeichnis

[Abd18] Abdelmassih, Christian: *Container Orchestration in Security Demanding Environments at the Swedish Police Authority*, KTH, School of Electrical Engineering and Computer Science (EECS), Masterarbeit, Juli 2018. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-228531. – 59 S.

[AD19] Arundel, John ; Domingus, Justin: *Cloud Native DevOps mit Kubernetes*. dpunkt.verlag, 2019 https://learning.oreilly.com/library/view/cloud-native-devops/9781098123178/. – ISBN 9783960888284

[Aqu20a] Aqua Security: *kube-bench – Checks whether Kubernetes is deployed according to security best practices as defined in the CIS Kubernetes Benchmark*. https://github.com/aquasecurity/kube-bench. Version: Juni 2020. – abgerufen am 27.06.2020

[Aqu20b] Aqua Security: *Trivy – A Simple and Comprehensive Vulnerability Scanner for Containers and other Artifacts, Suitable for CI*. https://github.com/aquasecurity/trivy. Version: Juni 2020. – abgerufen am 03.07.2020

[Ate19] Atelsek, Jean: *Rancher Labs promises an 'easy button' for multi-cluster Kubernetes deployments*. https://info.rancher.com/hubfs/eBooks,%20reports,%20and%20whitepapers/451_Reprint_RancherLabs_09AUG2019.pdf. Version: August 2019. – abgerufen am 29.05.2020

[AWI19] AWI: *Das Alfred-Wegener-Institut (AWI)*. https://www.awi.de/ueber-uns/organisation/profil.html. Version: August 2019. – abgerufen am 11.03.2020

[BD18] Bartoletti, Dave ; Dai, Charlie: *The Forrester New WaveTM: Enterprise Container Platform Software Suites, Q4 2018*. https://info.rancher.com/hubfs/eBooks,%20reports,%20and%20whitepapers/The%20Forrester%20New%20Wave%20-%20Enterprise%20Container%20Platform%20Software%20Suites,%20Q4%202018.pdf. Version: Oktober 2018. – abgerufen am 22.03.2020

[Bjø19] Bjørgeengen, Jarle: *Lecture Notes in Computer Science; High Performance Computing: A Multitenant Container Platform with OKD, Harbor Registry and ELK*. http://dx.doi.org/10.1007/978-3-030-34356-9_7. Version: Dezember 2019. – abgerufen am 29.05.2020

[BSI20a] BSI: *IT-Grundschutz-Kompendium*. Reguvis, 2020 https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Kompendium/IT_Grundschutz_Kompendium_Edition2020.pdf?__blob=publicationFile&v=6. – ISBN 9783846209066. – abgerufen am 30.05.2020

[BSI20b] BSI: *SYS.1.6: Container*. https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Grundschutz/Drafts/Community_Draft/SYS_1_6_Container_CD.pdf?__blob=publicationFile&v=11. Version: März 2020. – abgerufen am 30.05.2020

[Can20] Canonical: *cloud-init – The standard for customising cloud instances*. https://cloud-init.io/. Version: Mai 2020. – abgerufen am 18.05.2020

[CIS19] CIS: *CIS Kubernetes Benchmark – v1.5.0 - 10-14-2019*. https://www.cisecurity.org/benchmark/kubernetes/. Version: Oktober 2019. – abgerufen am 08.06.2020

[CIS20] CIS: *About us*. https://www.cisecurity.org/about-us/. Version: August 2020. – abgerufen am 15.08.2020

[CNC20a] CNCF: *Cloud Native Computing Foundation*. https://www.cncf.io/. Version: Mai 2020. – abgreufen am 06.05.2020

[CNC20b] CNCF: *CNCF SURVEY2019 : Deployments are getting larger as cloud native adoption becomes mainstream*. https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf. Version: März 2020. – angerufen am 13.04.2020

[Deh18] Dehghani, Zhamak: *How to break a Monolith into Microservices*. https://martinfowler.com/articles/break-monolith-into-microservices.html. Version: April 2018. – abgerufen am 21.04.2020

[Doc20a] Docker: *Docker Engine overview*. https://docs.docker.com/engine/. Version: Mai 2020. – abgerufen am 11.05.2020

[Doc20b] Docker: *Docker Engine release notes*. https://docs.docker.com/release-notes/docker-engine/. Version: Mai 2020. – abgerufen am 11.05.2020

[Doc20c] Docker: *Docker Hub – Build and Ship any Application Anywhere*. https://hub.docker.com/. Version: Mai 2020. – abgerufen am 11.05.2020

[Doc20d] Docker: *Docker overview*. https://docs.docker.com/engine/docker-overview/. Version: Mai 2020. – abgerufen am 02.05.2020

[Doc20e] Docker: *Get started with Notary*. https://docs.docker.com/notary/getting_started/. Version: Mai 2020. – abgerufen am 18.05.2020

[Doc20f] Docker: *Secure Engine*. https://docs.docker.com/engine/security/. Version: Juni 2020. – abgerufen am 17.06.2020

[Doc20g] Docker Samples: *Example Voting App*. https://github.com/dockersamples/example-voting-app. Version: Februar 2020. – abgerufen am 26.07.2020

[ECTP19] Edwards, Stefan ; Czarnota, Dominik ; Tonic, Robert ; Perez, Ben: *Kubernetes : Security Whitepaper*. https://github.com/kubernetes/community/blob/master/wg-security-audit/findings/Kubernetes%20White%20Paper.pdf. Version: Juni 2019. – abgerufen am 22.03.2020

[Etc20] Etcd: *etcd – A distributed, reliable key-value store for the most critical data of a distributed system*. https://etcd.io/. Version: Mai 2020. – abgerufen am 13.05.2020

[FL19] Fowler, Martin ; Lewis, James: *Microservices Guide*. https://martinfowler.com/microservices/. Version: August 2019. – abgerufen am 21.04.2020

[Fow15a] Fowler, Martin: *Microservice Trade-Offs*. https://martinfowler.com/articles/microservice-trade-offs.html. Version: Juli 2015. – abgerufen am 07.05.2020

[Fow15b] Fowler, Martin: *MonolithFirst*. https://martinfowler.com/bliki/MonolithFirst.html. Version: Juni 2015. – abgerufen am 26.04.2020

[Fow16] Fowler, Martin: *InfrastructureAsCode*. https://martinfowler.com/bliki/InfrastructureAsCode.html. Version: März 2016. – abgerufen am 06.05.2020

[Git20a] GitLab: *GitLab Continuous Integration (CI) & Continuous Delivery (CD).* https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/. Version: Mai 2020. – abgerufen am 19.05.2020

[Git20b] GitLab: *GitLab Runner Helm Chart.* https://docs.gitlab.com/runner/install/kubernetes.html. Version: August 2020. – abgerufen am 03.08.2020

[Git20c] GitLab: *What is GitLab?* https://about.gitlab.com/what-is-gitlab/. Version: März 2020. – abgerufen am 23.03.2020

[God19] Godlove, David: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning): Singularity : Simple, secure containers for compute-driven workloads.* https://dl.acm.org/doi/10.1145/3332186.3332192. Version: August 2019. – abgerufen am 29.05.2020

[Goo20a] Google: *Google Forms: Free Online Surveys for Personal Use.* https://www.google.com/forms/about/. Version: August 2020. – abgerufen am 07.08.2020

[Goo20b] GoogleContainerTools: *kaniko – Build Container Images In Kubernetes.* https://github.com/GoogleContainerTools/kaniko. Version: August 2020. – abgerufen am 04.08.2020

[Har20] Harbor: *Harbor – Our mission is to be the trusted cloud native repository for Kubernetes.* https://goharbor.io/. Version: Mai 2020. – abgerufen am 17.05.2020

[HB19] Haar, Christoph ; Buchmann, Erik: *IT-Grundschutz für die Container-Virtualisierung mit dem neuen BSI-Baustein SYS. 1.6.* https://slub.qucosa.de/api/qucosa%3A33099/attachment/ATT-0/. Version: Februar 2019. – abgerufen am 29.05.2020

[Hel18] Helmholtz: *Helmholtz Infrastructure for Federated ICT Services (HIFIS).* https://www.helmholtz.de/fileadmin/user_upload/01_forschung/Helmholtz_Inkubator_HIFIS.pdf. Version: September 2018. – abgerufen am 11.03.2020

[HKH+19] Heiderich, M. ; Krein, N. ; Hippert, N. ; Hector, J. ; Kinugawa, M. ; Moritz, S. ; Fässler, F.: *Cure53 – Pentest-Report Rancher Server Web & API 07.2019.* https://releases.rancher.com/documents/security/pentests/2019/RAN-01-cure53-report.final.pdf. Version: Dezember 2019. – abgerufen am 23.06.2020

[HWW+19] Heiderich, M. ; Wege, M. ; Weisser, D. ; Larsson, J. ; Hector, J. ; Krein, N. ; Kinugawa, M.: *Cure53 – Pentest-Report Harbor 10.2019.* https://github.com/goharbor/harbor/blob/master/docs/security/Harbor_Security_Audit_Oct2019.pdf. Version: Oktober 2019. – abgerufen am 23.06.2020

[Kub20a] Kubernetes: *Configuration Best Practices.* https://kubernetes.io/docs/concepts/configuration/overview/. Version: Februar 2020. – abgerufen am 13.05.2020

[Kub20b] Kubernetes: *Deployments.* https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. Version: Mai 2020. – abgerufen am 13.05.2020

[Kub20c] Kubernetes: *KubeFed – Kubernetes Cluster Federation.* https://github.com/kubernetes-sigs/kubefed. Version: Mai 2020. – abgerufen am 13.06.2020

[Kub20d] KUBERNETES: *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/. Version: März 2020. – abgerufen am 13.05.2020

[Kub20e] KUBERNETES: *NGINX Ingress Controller – Bare-metal considerations*. https://kubernetes.github.io/ingress-nginx/deploy/baremetal/. Version: Juli 2020. – abgerufen am 12.07.2020

[Kub20f] KUBERNETES: *Options for Highly Available topology*. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/. Version: Mai 2020. – abgerufen am 16.06.2020

[Kub20g] KUBERNETES: *Overview of Cloud Native Security*. https://kubernetes.io/docs/concepts/security/overview/. Version: Juni 2020. – abgerufen am 16.06.2020

[Kub20h] KUBERNETES: *Pod Overview*. https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. Version: April 2020. – abgerufen am 13.05.2020

[Kub20i] KUBERNETES: *Pod Security Standards*. https://kubernetes.io/docs/concepts/security/pod-security-standards/. Version: Mai 2020. – abgerufen am 22.06.2020

[Kub20j] KUBERNETES: *Securing a Cluster*. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/. Version: Mai 2020. – abgerufen am 16.06.2020

[Kub20k] KUBERNETES: *Service*. https://kubernetes.io/docs/concepts/services-networking/service/. Version: April 2020. – abgerufen am 13.05.2020

[Kub20l] KUBERNETES: *Tools*. https://kubernetes.io/docs/reference/tools/. Version: März 2020. – abgerufen am 18.05.2020

[Kub20m] KUBERNETES: *What is Kubernetes?* https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. Version: März 2020. – abgerufen am 28.04.2020

[Lie19] LIEBEL, Oliver: *Skalierbare Container-Infrastrukturen : das Handbuch für Administratoren*. 2., aktualisierte und erweiterte Auflage. Bonn : Rheinwerk Verlag, 2019 (Rheinwerk Computing). – ISBN 3836263858 and 9783836263856 and 9783836263856

[Lin18] LINK, Vincent: *Konzeption und Realisierung einer Cloud-Plattform zur Konfiguration und dem Deployment von Services*, Universität Stuttgart, Masterarbeit, Januar 2018. http://dx.doi.org/10.18419/opus-9825. – DOI 10.18419/opus–9825

[Lin20] LINTHICUM, David S.: *Market Landscape Report – GigaOm Radar for Leveraging Federated Kubernetes*. https://info.rancher.com/hubfs/eBooks,%20reports,%20and%20whitepapers/gigaom-radar-for-leveraging-federated-kubernetes%20-%20FINAL.pdf. Version: April 2020. – abgerufen am 15.05.2020

[Luk18] LUKŠA, Marko: *Kubernetes in Action : Anwendungen in Kubernetes-Clustern bereitstellen und verwalten*. München : Hanser, 2018 https://www.hanser-elibrary.com/doi/book/10.3139/9783446456020. – ISBN 9783446456020

[Lym19] LYMAN, Jay: *Kubernetes and Beyond – Effective Implementation of Cloud-Native Software in the Enterprise*. https://info.rancher.com/hubfs/eBooks,%20reports,%20and%20whitepapers/10703_Advisory_PF_RancherLabs.pdf. Version: August 2019. – abgerufen am 29.05.2020

[Met20] MetalLB: *MetalLB – bare metal load-balancer for Kubernetes*. https://metallb.universe.tf/. Version: Juli 2020. – abgerufen am 12.07.2020

[Mou16] Mouat, Adrian: *Using Docker : Developing and Deploying Software with Containers*. 3. Sebastopol, CA : O'Reilly Media, 2016 https://learning.oreilly.com/library/view/using-docker/9781491915752/. – ISBN 9781491915752

[Nar18] Narjes, Mieke: *Konzeption und Umsetzung eines automatischen Kubernetes Deployments in die ICC der HAW in Hinblick auf spätere Weiterverwendung in der Lehre*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, Juli 2018. http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4368/

[Net20] NetApp: *Nur NetApp bietet alle Voraussetzungen, um eine maßgeschneiderte Data-Fabric-Umgebung aufzubauen*. https://www.netapp.com/de/index.aspx. Version: Mai 2020. – abgerufen am 13.05.2020

[OWA20] OWASP: *Source Code Analysis Tools*. https://owasp.org/www-community/Source_Code_Analysis_Tools. Version: Juni 2020. – abgerufen am 18.06.2020

[Pal17] Palopoli, Amedeo: *Containerization in Cloud Computing: performance analysis of virtualization architectures*, Universität Bologna, Dissertation, Dezember 2017. http://amslaurea.unibo.it/14818/

[PHP16] Peinl, Rene ; Holzschuher, Florian ; Pfitzer, Florian: Docker Cluster Management for the Cloud - Survey Results and Own Solution. In: *Journal of Grid Computing* 14 (2016), April. http://dx.doi.org/10.1007/s10723-016-9366-y. – DOI 10.1007/s10723–016–9366–y

[PR15] Pohl, Klaus ; Rupp, Chris: *Basiswissen Requirements Engineering : Aus- und Weiterbildung nach IREB-Standard zum "Certified Professional for Requirements Engineering" : Foundation Level nach IREB-Standard*. 4., überarbeitete Auflage. Heidelberg : dpunkt.verlag, 2015 http://site.ebrary.com/lib/suub/docDetail.action?docID=11050833. – ISBN 9783864916731

[Qua20] Quay: *Clair – Vulnerability Static Analysis for Containers*. https://github.com/quay/clair. Version: Mai 2020. – abgerufen am 18.05.2020

[Ran19] Rancher Labs: *A Guide to Kubernetes with Rancher*. https://info.rancher.com/hubfs/eBooks,%20reports,%20and%20whitepapers/guide-to-kubernetes-with-rancher.pdf. Version: Januar 2019. – abgerufen am 22.03.2020

[Ran20a] Rancher Labs: *CIS Benchmark Rancher Self-Assessment Guide - v2.4*. https://releases.rancher.com/documents/security/2.4/Rancher_Benchmark_Assessment.pdf. Version: Juni 2020. – abgerufen am 07.06.2020

[Ran20b] Rancher Labs: *Provisioning Kubernetes Clusters in vSphere*. https://rancher.com/docs/rancher/v2.x/en/cluster-provisioning/rke-clusters/node-pools/vsphere/provisioning-vsphere-clusters/. Version: Juli 2020. – abgerufen am 12.07.2020

[Ran20c] Rancher Labs: *Rancher – Architecture*. https://rancher.com/docs/rancher/v2.x/en/overview/architecture/. Version: Mai 2020. – abgerufen am 17.05.2020

[Ran20d] Rancher Labs: *Rancher – Best Practices Guide*. https://rancher.com/docs/rancher/v2.x/en/best-practices/. Version: März 2020. – abgerufen am 22.03.2020

[Ran20e] Rancher Labs: *Rancher – Hardening Guide v2.4*. https://releases.rancher.com/documents/security/2.4/Rancher_Hardening_Guide.pdf. Version: Juni 2020. – abgerufen am 08.06.2020

[Ran20f] Rancher Labs: *Rancher 2.4: Technical Architecture*. https://cdn2.hubspot.net/hubfs/468859/eBooks,%20reports,%20and%20whitepapers/20200305_Rancher_2_4_Architecture_WP.pdf. Version: Februar 2020. – abgerufen am 25.03.2020

[Ran20g] Rancher Labs: *Security*. https://rancher.com/docs/rancher/v2.x/en/security/. Version: März 2020. – abgerufen am 22.03.2020

[Red20a] Red Hat: *Automatisierung – IT-Automatisierung erklärt*. https://www.redhat.com/de/topics/automation/whats-it-automation. Version: Juni 2020. – abgerufen am 24.06.2020

[Red20b] Red Hat: *Virtualisierung – KVM erklärt*. https://www.redhat.com/de/topics/virtualization/what-is-KVM. Version: September 2020. – abgerufen am 07.09.2020

[SBBK15] Stucky, Karl-Uwe ; Bengel, Günther ; Baun, Christian ; Kunze, Marcel: *Masterkurs parallele und verteilte Systeme : Grundlagen und Programmierung von Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und Cloud*. 2., erw. und aktualisierte Aufl. Wiesbaden : Springer Vieweg, 2015 http://dx.doi.org/10.1007/978-3-8348-2151-5. – ISBN 9783834821515

[SF17] Syed, Madiha H. ; Fernandez, Eduardo B.: *Proceedings of the 22nd European Conference on Pattern Languages of Programs: The Container Manager Pattern*. https://dl.acm.org/doi/pdf/10.1145/3147704.3147735. Version: Juli 2017. – abgerufen am 29.05.2020

[SMS17] Souppaya, Murugiah ; Morello, John ; Scarfone, Karen: *NIST : Application Container Security Guide*. http://dx.doi.org/https://doi.org/10.6028/NIST.SP.800-190. Version: September 2017. – abgerufen am 30.05.2020

[Ubu20a] Ubuntu: *The story of Ubuntu*. https://ubuntu.com/about. Version: Mai 2020. – abgerufen am 18.05.2020

[Ubu20b] Ubuntu: *Ubuntu Cloud Images*. https://cloud-images.ubuntu.com/. Version: Mai 2020. – abgerufen am 18.05.2020

[VMw20a] VMware: *VMware Security Hardening Guides*. https://www.vmware.com/security/hardening-guides.html. Version: März 2020. – abgerufen am 22.03.2020

[VMw20b] VMware: *vRealize Suite*. https://www.vmware.com/de/products/vrealize-suite.html. Version: Mai 2020. – abgerufen am 10.05.2020

[VMw20c] VMware: *vSphere with Kubernetes Configuration and Management*. https://docs.vmware.com/en/VMware-vSphere/7.0/vsphere-esxi-vcenter-server-70-vsphere-with-kubernetes-guide.pdf. Version: Juli 2020. – abgerufen am 25.08.2020

[VMw20d] VMware: *Über uns*. https://www.vmware.com/de/company.html. Version: Mai 2020. – abgerufen am 09.05.2020

[VSTK19] Vayghan, Leila A. ; Saied, Mohamed A. ; Toeroe, Maria ; Khendek, Ferhat: *Kubernetes as an Availability Manager for Microservice Applications*. `https://arxiv.org/pdf/1901.04946.pdf`. Version: Januar 2019. – abgerufen am 29.05.2020

[WAG+18] Wöhrmann, Bertram ; Alder, Urs S. ; Grosse, Jan ; Baumgart, Günter ; Schönfeld, Thomas ; Zimmer, Dennis ; Wegner, Frank ; Söldner, Jens-Henrik: *VMware vSphere 6.7 : das umfassende Handbuch*. 5., aktualisierte und erweiterte Auflage. Bonn : Rheinwerk Verlag, 2018 (Rheinwerk Design). – ISBN 383626336X and 9783836263368 and 9783836263368

[web19] webODV: *webODV provides online Ocean Data View (ODV) services like the analysis, exploration and visualization of oceanographic and other environmental data*. `https://webodv.awi.de/`. Version: 2019. – abgerufen am 07.09.2020

[Wil15] Wilsenach, Rouan: *DevOpsCulture*. `https://martinfowler.com/bliki/DevOpsCulture.html`. Version: Juli 2015. – abgerufen am 14.03.2020

A Anhang

A.1 Installationsroutinen diverser Tools

In diesem Abschnitt sind die Installationsanweisungen einiger Tools aufgeführt, die vorwiegend in den praktisch orientierten Kapiteln 7 und 8 immer wieder verwendet werden.

Die *Docker Engine* ist eine *Open Source*-Container-Technologie zur Erstellung und Containerisierung von Anwendungen. Die Installation kann unter *Ubuntu*, wie in der Auflistung A.1 zu sehen ist, mit Hilfe des APT-Paket-Managers (*Advanced Package Tool*) erfolgen. Am Ende kann das eigene Nutzerkonto in die „*docker*“-Gruppe aufgenommen werden, damit für die Verwendung der *Docker*-Befehle keine erhöhten Berechtigungen gebraucht werden.

Auflistung A.1: Installation der *Docker Engine* unter *Linux* (*Ubuntu*)

```
#!/bin/bash
curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
↪  $(lsb_release -cs) stable"
sudo apt -y update
sudo apt -y install docker-ce docker-ce-cli containerd.io
# Add yourself to the Docker group
sudo gpasswd -a $USER docker
```

Docker Compose ist ein Werkzeug zur Definition und Ausführung von Multi-Container-*Docker*-Anwendungen. Es verwendet eine YAML-Datei zur Konfiguration der Dienste einer Anwendung. Anschließend können mit einem einzigen Befehl alle Dienste aus der erstellten Konfiguration heraus erstellt und gestartet werden. Die Installationsroutine des *Linux Binaries* der *Docker Compose* CLI, ist der Auflistung A.2 zu entnehmen.

Auflistung A.2: Installation von *Docker Compose* unter *Linux*

```
#!/bin/bash
sudo curl -L
↪  "https://github.com/docker/compose/releases/download/1.26.1/docker-compose-$(uname
↪  -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose
```

Die *Rancher Kubernetes Engine* (RKE) ist eine CNCF-zertifizierte *Kubernetes*-Distribution, die vollständig in *Docker*-Containern läuft. Sie funktioniert sowohl auf *Bare Metal*-, als auch virtualisierten Servern. RKE kann das Problem der Installationskomplexität eines *K8s*-Clusters lösen. Unabhängig vom Betriebssystem und der Plattform vereinfacht und automatisiert RKE die Installation und den Betrieb von *Kubernetes*. Die betroffene Bereitstellungsumgebung muss lediglich eine unterstützte Version von *Docker* ausführen können. Die dazugehörige *Binary*-Installation der RKE CLI ist in der Auflistung A.3 dargestellt.

Auflistung A.3: Installation der RKE CLI unter *Linux*

```
#!/bin/bash
sudo curl -L "https://github.com/rancher/rke/releases/download/v1.1.3/rke_linux-amd64" -o
↪  /usr/local/bin/rke
sudo chmod +x /usr/local/bin/rke
```

Mit dem *Kubernetes*-Kommandozeilen-Tool *kubectl* können Befehle in *K8s*-Clustern ausgeführt werden. Hiermit können u. a. Container-Anwendungen bereitgestellt, Cluster-Ressourcen inspiziert und verwaltet sowie Protokolldateien eingesehen werden. Die Auflistung A.4 zeigt die notwendigen Installationsschritte und die Integration einer *Autocomplete*-Funktion in die *Bash-Shell*.

Auflistung A.4: Installation der *kubectl* CLI unter *Linux*

```
#!/bin/bash
sudo curl -L "https://storage.googleapis.com/kubernetes-release/release/`curl -s https://sto
↪  rage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl" -o
↪  /usr/local/bin/kubectl
sudo chmod +x /usr/local/bin/kubectl
# Add autocomplete permanently to bash shell
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

Mit dem *Bash*-Skript *kubetail*, können Protokolldateien aus mehreren *Pods* zu einem *Stream* aggregiert werden; die Installationsanweisungen sind in der Auflistung A.5 zu finden. Implizit basiert *kubetail* auf der Ausführung von „*kubectl logs -f*“, nur über mehrere *Pods* hinweg. Besonders bei *Debugging*-Aufgaben ist das Tool ein nützlicher Helfer.

Auflistung A.5: Installation von *kubetail* unter *Linux*

```
#!/bin/bash
sudo curl -L "https://raw.githubusercontent.com/johanhaleby/kubetail/master/kubetail" -o
↪  /usr/local/bin/kubetail
sudo chmod +x /usr/local/bin/kubetail
```

Das Tool *Helm*, mit seiner Installationsroutine in der Auflistung A.6, unterstützt bei der Verwaltung von *Kubernetes*-Anwendungen. Die hierfür zugrunde liegenden *Helm Charts* helfen bei der Definition, Installation und Aktualisierung von sowohl einfachen als auch komplexen *K8s*-Anwendungen.

Auflistung A.6: Installation der *Helm* CLI unter *Linux*

```
#!/bin/bash
curl -LO "https://get.helm.sh/helm-v3.2.4-linux-amd64.tar.gz"
tar -zxvf helm-v3.2.4-linux-amd64.tar.gz
cd linux-amd64/ && sudo cp helm /usr/local/bin/helm
sudo chmod +x /usr/local/bin/helm
```

Für eine Entwicklungs- und Testumgebung kann der *Rancher Server* durch Ausführen eines einzelnen *Docker*-Containers, wie in der Auflistung A.7 dargestellt ist, installiert werden. Auf dem entsprechendem Host muss dafür nur die *Docker Engine* vorhanden sein.

Auflistung A.7: Installation von *Rancher* auf einem einzelnen Knoten mit *Docker* unter *Linux*

```
#!/bin/bash
docker run -d --restart=unless-stopped -p 80:80 -p 443:443 rancher/rancher:stable
```

A.2 *Cloud-init*

Mit Hilfe der in der Auflistung A.8 dargestellten *Cloud-init*-Konfiguration, können bei der VM-Bereitstellung direkt Einstellungen für eine neue Instanz mitgegeben werden. Die Maßnahmen können u. a. Benutzerkonten (*users*), Paketinstallationen (*packages*), Dateierstellungen (*write_files*) und auszuführende Befehle (*runcmd*) betreffen. Der Inhalt einer fertigen Konfigurationsdatei wird oftmals in der *Base64*-Kodierung übermittelt: „*cat cloud-config.yml | base64 -w 0*“.

In der Konfiguration werden zwei Benutzerkonten (*ubuntu*, *rke*) erstellt, mit denen nur durch SSH-*Keys* interagiert werden kann. Das zweite Benutzerkonto besitzt weniger Berechtigungen und eignet sich daher für die RKE-Installation. Die zu erstellenden Dateien enthalten u. a. Kernel-Parameter, *Docker Daemon*-Einstellungen und ein Hilfsskript für das optionale Setzen einer statischen Netzwerkkonfiguration. Dieses Hilfsskript verwendet die *String*-Umgebungsparameter „*ip-address*“, „*gateway*“ und „*dns-servers*“, welche zuvor in der *VMware vSphere*-Umgebung durch den „OVF *Environment Transport*“ deklariert werden. Außerdem werden noch Einstellungen am SSH-*Daemon*, an der *Firewall ufw* und weiteren Dingen durch Befehle unterhalb der „*runcmd*“-Kategorie abgesetzt.

Auflistung A.8: Konfiguration für die Verwendung von *Cloud-init* bei einem *Ubuntu Cloud Image* [Ran20e]

```
#cloud-config
users:
 - name: ubuntu
   home: /home/ubuntu
   shell: /bin/bash
   lock_passwd: True
   gecos: Ubuntu
   groups: [adm, sudo, docker]
   sudo: ALL=(ALL) NOPASSWD:ALL
   ssh_authorized_keys:
    - ssh-ed25519 ... ubuntu
 - name: rke
   home: /home/rke
   shell: /bin/bash
   lock_passwd: True
   gecos: RKE
   groups: [docker]
   sudo: False
   ssh_authorized_keys:
    - ssh-ed25519 ... rke

packages:
 - apt-transport-https
 - gnupg-agent
 - fail2ban
 - nfs-common

write_files:
```

```
 - content: |
    vm.overcommit_memory=1
    vm.panic_on_oom=0
    kernel.panic=10
    kernel.panic_on_oops=1
    kernel.keys.root_maxbytes=25000000
    net.bridge.bridge-nf-call-iptables=1
   path: /etc/sysctl.d/90-kubelet.conf
 - content: |
    {
     "bip": "10.3.0.1/24"
    }
   path: /etc/docker/daemon.json
 - content: |
    #!/bin/bash
    vmtoolsd --cmd 'info-get guestinfo.ovfEnv' > /tmp/ovfenv
    IPAddress=$(sed -n 's/.*Property oe:key="ip-address" oe:value="\([^"]*\).*/\1/p'
    ↪  /tmp/ovfenv)
    Gateway=$(sed -n 's/.*Property oe:key="gateway" oe:value="\([^"]*\).*/\1/p' /tmp/ovfenv)
    DNS=$(sed -n 's/.*Property oe:key="dns-servers" oe:value="\([^"]*\).*/\1/p' /tmp/ovfenv)

    if [ -n "$IPAddress" ] && [ -n "$Gateway" ] && [ -n "$DNS" ]
    then
      rm -rf /etc/netplan/*

      cat > /etc/netplan/01-netcfg.yaml <<EOF
    network:
      version: 2
      renderer: networkd
      ethernets:
        ens192:
          addresses:
            - $IPAddress
          gateway4: $Gateway
          nameservers:
            addresses : [$DNS]
    EOF
      echo 'network: {config: disabled}' >
      ↪  /etc/cloud/cloud.cfg.d/99-disable-network-config.cfg
      netplan apply
    fi
   path: /home/ubuntu/static-netcfg.sh

runcmd:
 # Purge useless Packages
 - apt -y purge snapd
 # Disabling Swap
 - sed -i '/ swap / s/^\(.*\)$/#\1/g' /etc/fstab
 # Configure Networking
 - /bin/bash /home/ubuntu/static-netcfg.sh
 # Configure Time Zone
 - timedatectl set-timezone Europe/Berlin
 # Configure SSH
 - sed -i -e '/^#*PermitRootLogin/s/^.*$/PermitRootLogin no/' /etc/ssh/sshd_config
 - sed -i -e '/^#*PubkeyAuthentication/s/^.*$/PubkeyAuthentication yes/' /etc/ssh/sshd_config
 - sed -i -e '/^#*PasswordAuthentication/s/^.*$/PasswordAuthentication no/'
 ↪  /etc/ssh/sshd_config
 - sed -i -e '/^#*ChallengeResponseAuthentication/s/^.*$/ChallengeResponseAuthentication
 ↪  no/' /etc/ssh/sshd_config
 - sed -i -e '/^#*AllowTcpForwarding/s/^.*$/AllowTcpForwarding yes/' /etc/ssh/sshd_config
 - systemctl reload ssh
```

```
  # Install Docker Engine
87 - curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | apt-key add -
   - add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
   ↪  $(lsb_release -cs) stable"
89 - apt -y update
   - apt -y install docker-ce docker-ce-cli containerd.io
91 # Configure Kernel Runtime Parameters
   - sysctl -p /etc/sysctl.d/90-kubelet.conf
93 # Create etcd User and Group
   - addgroup --gid 52034 etcd
95 - useradd --comment "etcd service account" --uid 52034 --gid 52034 etcd
   # Configure UFW
97 - ufw default deny
   - ufw default allow outgoing
99 # Rules UFW - Rancher, RKE
   - /bin/bash -c "ufw allow
   ↪  22,80,443,2376,2379,2380,6443,6783,9099,9796,10250,10254,30000:32767/tcp"
101 - /bin/bash -c "ufw allow 6783:6784,8472,30000:32767/udp"
   - ufw enable
```

A.3 RKE-*Template*

RKE verwendet eine Cluster-Konfigurationsdatei, die im YAML-Format definiert wird. In ihr wird bspw. bestimmt, durch welche Knoten der Cluster aufgebaut und wie im Speziellen *Kubernetes* bereitgestellt werden soll. Es gibt viele Konfigurationsoptionen; ein dabei in der Ausarbeitung verwendetes und zudem gehärtetes Beispiel ist in der Auflistung A.9 dargestellt. Konkret unterschieden werden globale Maßnahmen für die Cluster-Konfiguration und Einstellungen bezogen auf *Rancher* (*rancher_kubernetes_engine_config*). Vor allem wichtig ist die standardmäßige Aktivierung von PSPs (*restricted*) und der *Network Policies*, in einem neu bereitzustellenden *Kubernetes*-Cluster.

Auflistung A.9: Gehärtetes RKE-*Template* für die Provisionierung von Container-Clustern durch *Rancher* [Ran20e]

```
1  # Cluster Config
   default_pod_security_policy_template_id: restricted
3  docker_root_dir: /var/lib/docker
   enable_cluster_alerting: false
5  enable_cluster_monitoring: false
   enable_network_policy: true
7  local_cluster_auth_endpoint:
     enabled: false
9  windows_prefered_cluster: false

11 # Rancher Config
   rancher_kubernetes_engine_config:
13   kubernetes_version: v1.15.x
     authentication:
15     strategy: x509
     bastion_host:
17     ssh_agent_auth: false
     ignore_docker_version: true
19   ingress:
       provider: nginx
21   monitoring:
```

```
    provider: metrics-server
    replicas: 1
  network:
    mtu: 0
    plugin: canal
  restore:
    restore: false
  ssh_agent_auth: false
  addon_job_timeout: 30
  addons: |-
    ---
    apiVersion: v1
    kind: Namespace
    metadata:
      name: ingress-nginx
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
      name: default-psp-role
      namespace: ingress-nginx
    rules:
    - apiGroups:
      - extensions
      resourceNames:
      - default-psp
      resources:
      - podsecuritypolicies
      verbs:
      - use
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: RoleBinding
    metadata:
      name: default-psp-rolebinding
      namespace: ingress-nginx
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: Role
      name: default-psp-role
    subjects:
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:serviceaccounts
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:authenticated
    ---
    apiVersion: v1
    kind: Namespace
    metadata:
      name: cattle-system
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: Role
    metadata:
      name: default-psp-role
      namespace: cattle-system
    rules:
    - apiGroups:
      - extensions
```

```
      resourceNames:
      - default-psp
      resources:
      - podsecuritypolicies
      verbs:
      - use
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: RoleBinding
    metadata:
      name: default-psp-rolebinding
      namespace: cattle-system
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: Role
      name: default-psp-role
    subjects:
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:serviceaccounts
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:authenticated
    ---
    apiVersion: policy/v1beta1
    kind: PodSecurityPolicy
    metadata:
      name: restricted
    spec:
      requiredDropCapabilities:
      - NET_RAW
      privileged: false
      allowPrivilegeEscalation: false
      defaultAllowPrivilegeEscalation: false
      fsGroup:
        rule: RunAsAny
      runAsUser:
        rule: MustRunAsNonRoot
      seLinux:
        rule: RunAsAny
      supplementalGroups:
        rule: RunAsAny
      volumes:
      - emptyDir
      - secret
      - persistentVolumeClaim
      - downwardAPI
      - configMap
      - projected
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRole
    metadata:
      name: psp:restricted
    rules:
    - apiGroups:
      - extensions
      resourceNames:
      - restricted
      resources:
      - podsecuritypolicies
```

```
      verbs:
      - use
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRoleBinding
    metadata:
      name: psp:restricted
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: ClusterRole
      name: psp:restricted
    subjects:
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:serviceaccounts
    - apiGroup: rbac.authorization.k8s.io
      kind: Group
      name: system:authenticated
    ---
    apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: tiller
      namespace: kube-system
    ---
    apiVersion: rbac.authorization.k8s.io/v1
    kind: ClusterRoleBinding
    metadata:
      name: tiller
    roleRef:
      apiGroup: rbac.authorization.k8s.io
      kind: ClusterRole
      name: cluster-admin
    subjects:
    - kind: ServiceAccount
      name: tiller
      namespace: kube-system

  services:
    etcd:
      backup_config:
        enabled: false
        interval_hours: 12
        retention: 6
        safe_timestamp: false
      creation: 12h
      extra_args:
        election-timeout: '5000'
        heartbeat-interval: '500'
      gid: 52034
      retention: 72h
      snapshot: false
      uid: 52034
    kube_api:
      always_pull_images: true
      audit_log:
        enabled: true
      event_rate_limit:
        enabled: true
      pod_security_policy: true
      secrets_encryption_config:
```

```
        enabled: true
      service_node_port_range: 30000-32767
    kube_controller:
      extra_args:
        address: 127.0.0.1
        feature-gates: RotateKubeletServerCertificate=true
        profiling: 'false'
        terminated-pod-gc-threshold: '1000'
    kubelet:
      extra_args:
        anonymous-auth: 'false'
        event-qps: '0'
        feature-gates: RotateKubeletServerCertificate=true
        make-iptables-util-chains: 'true'
        protect-kernel-defaults: 'true'
        streaming-connection-idle-timeout: 1800s
        tls-cipher-suites: >-
          TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256,TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256,TLS_
          ↪ ECDHE_ECDSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384,TLS_E
          ↪ CDHE_RSA_WITH_CHACHA20_POLY1305,TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384,TLS_RS
          ↪ A_WITH_AES_256_GCM_SHA384,TLS_RSA_WITH_AES_128_GCM_SHA256
      fail_swap_on: false
      generate_serving_certificate: true
    scheduler:
      extra_args:
        address: 127.0.0.1
        profiling: 'false'
```

A.4 *vRealize Suite*

Mit Hilfe des *vRealize Orchestrators* (*vRO*) der *vRealize Suite* können automatisierbare *Workflows* in einer *VMware*-Virtualisierungsumgebung umgesetzt werden. Nach der Erstellung können diese dann in *vRealize Automation* (*vRA*) bzw. in der Ausprägung eines *Self-Service Portals* importiert und letztendlich einem Nutzerkreis angeboten werden. Die Auflistung A.10 zeigt weitere *vRO-Actions*, die in dem Modul „*de.awi.rancher*“ anderen *Workflows* zur Verfügung stehen; *Actions* sind hierbei äquivalent zu JS-Funktionen.

Auflistung A.10: Weitere *Actions* des Moduls „*de.awi.rancher*“ im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

function isEmptyString(str) {
  return (!str || str.length === 0 || !str.trim());
}

function checkClusterName(clusterName) {
  const CLUSTER_NAME_REGEX = '^[a-z0-9-]{1,8}-k8s-[0-9]{1,}$';

  if(!clusterName.match(CLUSTER_NAME_REGEX)) {
          throw 'Cluster name \"' + clusterName + '\" does not meet the requirements (' +
          ↪ CLUSTER_NAME_REGEX + ')!';
  }
}

function checkProjectName(projectName) {
  const PROJECT_NAME_REGEX = '^(?!System)([a-zA-Z0-9- ]{1,})$';
```

```
17
  if(!projectName.match(PROJECT_NAME_REGEX)) {
19        throw 'Project name \"' + projectName + '\" does not meet the requirements (' +
          ↪  PROJECT_NAME_REGEX + ')!';
  }
21}

23function checkNodePoolName(nodePoolName) {
  const NODE_POOL_NAME_REGEX = '^[a-z0-9-]{1,8}-k8s-[0-9]{1,}-worker-[0-9]{1,}-$';
25
  if(!nodePoolName.match(NODE_POOL_NAME_REGEX)) {
27        throw 'Node pool name \"' + nodePoolName + '\" does not meet the requirements (' +
          ↪  NODE_POOL_NAME_REGEX + ')!';
  }
29}

31function getClusterId(restHost, clusterId, clusterName, rancherCluster) {
  /* Use DynamicTypesDynamicObject */
33  if(rancherCluster) {
        if(rancherCluster.hasOwnProperty('id') && rancherCluster.hasOwnProperty('name')) {
35              clusterId = rancherCluster.id;
                clusterName = rancherCluster.name;
37      }
  }
39
  if(RANCHER.isEmptyString(clusterId) && !RANCHER.isEmptyString(clusterName)) {
41      RANCHER.checkClusterName(clusterName);
        var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters', null);
43      for(var i = 0; i < resObj.data.length; i++) {
                if(resObj.data[i].name === clusterName) {
45                      clusterId = resObj.data[i].id;
                        break;
47              }
        }
49      if(RANCHER.isEmptyString(clusterId)) {
                throw 'Cluster with the name \"' + clusterName + '\" not found!';
51      }
  } else if(RANCHER.isEmptyString(clusterId) && RANCHER.isEmptyString(clusterName)) {
53      throw 'Cluster name is missing!';
  }
55  return clusterId;
}
57
59function getProjectId(restHost, clusterId, projectId, projectName) {
  if(RANCHER.isEmptyString(projectId) && !RANCHER.isEmptyString(clusterId) &&
  ↪  !RANCHER.isEmptyString(projectName)){
        RANCHER.checkProjectName(projectName);
61      var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters/' + clusterId +
        ↪  '/projects', null);
        for(var i = 0; i < resObj.data.length; i++) {
63              if(resObj.data[i].name === projectName) {
                        projectId = resObj.data[i].id;
65                      break;
                }
67      }
        if(RANCHER.isEmptyString(projectId)) {
69              throw 'Project with the name \"' + projectName + '\" not found!';
        }
71  } else if(RANCHER.isEmptyString(projectId) && RANCHER.isEmptyString(projectName)) {
        throw 'Project name is missing!';
73  } else if(RANCHER.isEmptyString(projectId) && RANCHER.isEmptyString(clusterId)) {
```

```
            throw 'Cluster id is missing!';
  }
  return projectId;
}

function getClusterTemplateRevisionId(tpl) {
  switch(tpl) {
            case 'K8s v1.15.x Hardened':
            default:
                    return 'cattle-global-data:ctr-6p9b8';
            case 'K8s v1.15.x':
                    return 'cattle-global-data:ctr-rd6hh';
            case 'K8s v1.17.x Hardened':
                    return 'cattle-global-data:ctr-c6zsr';
            case 'K8s v1.17.x':
                    return 'cattle-global-data:ctr-vrkhk';
  }
}

function getNodeTemplateId(tpl) {
  switch(tpl) {
            case 'Low - Ubuntu 18.04, Memory 4 GiB, CPUs 2 Cores, Disk 20 GiB':
            default:
                    return 'cattle-global-nt:nt-tpzdf';
            case 'High - Ubuntu 18.04, Memory 8 GiB, CPUs 4 Cores, Disk 50 GiB':
                    return 'cattle-global-nt:nt-z48m8';
  }
}

function getRoleTemplateId(tpl) {
  switch(tpl) {
            case 'Read Only':
            default:
                    return 'read-only';
            case 'Project Member':
                    return 'project-member';
            case 'Project Owner':
                    return 'project-owner';
            case 'Cluster Member':
                    return 'cluster-member';
            case 'Cluster Owner':
                    return 'cluster-owner';
  }
}

function getUid(uid) {
  const UID_REGEX = '^[a-z0-9-]{1,8}@dmawi.de$';

  if(uid.match(UID_REGEX)) {
            return uid.split('@')[0];
  } else {
            throw 'Username (uid) \"' + uid + '\" does not meet the requirements (' +
            ↪  UID_REGEX + ')!';
  }
}

function getUserPrincipalId(dir, uid, username) {
  const USERNAME_REGEX = '^\D{1,}$';

  switch(dir) {
            case 'AD':
```

```
                if(!username.match(USERNAME_REGEX)) {
                        throw 'Username \"' + username + '\" does not meet the
                        ↪  requirements (' + USERNAME_REGEX + ')!';
                }
                return 'activedirectory_user://CN=' + username +
                ↪  ',OU=People,DC=dmawi,DC=de';
        case 'OpenLDAP':
        default:
                return 'openldap_user://uid=' + RANCHER.getUid(uid) +
                ↪  ',ou=People,dc=awi-bremerhaven,dc=de';
  }
}
```

Des Weiteren sind in den nachfolgenden Auflistungen A.11 bis A.19 alle restlichen *Workflows* abgedruckt, die im Zuge der Ausarbeitung entwickelt wurden. Hauptsächlich ist jeweils das Erstellen und Löschen einer bestimmten Ressource (*cluster*, *clusterRoleTemplateBinding*, *project*, *projectRoleTemplateBinding*, *nodePool*) innerhalb eines RKE-*Downstream*-Clusters umgesetzt. Diese *Workflows* werden schließlich als sogenannte „*Day-2 Operations*“ bezogen auf einen anzufordernden *K8s*-Cluster im *Self-Service Portal* angeboten.

Auflistung A.11: *Workflow „rancher_delete_cluster“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Delete Cluster */
var resObj = RANCHER.executeRequest(restHost, 'DELETE', '/clusters/' + clusterId, null);
result = resObj.id === clusterId;
```

Auflistung A.12: *Workflow „rancher_create_cluster_user“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Create Cluster User */
var requestContent = {
        "clusterId": clusterId,
        "roleTemplateId": RANCHER.getRoleTemplateId(roleTemplate),
        "userPrincipalId": RANCHER.getUserPrincipalId(directoryService, uid, username)
}
var resObj = RANCHER.executeRequest(restHost, 'POST', '/clusterroletemplatebindings',
↪  requestContent);
clusterRoleTemplateBindingId = resObj.id;
```

Auflistung A.13: *Workflow „rancher_delete_cluster_user“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

```

```
/* Get Cluster Role Template Binding Id */
if(RANCHER.isEmptyString(clusterRoleTemplateBindingId)) {
        var userPrincipalId = RANCHER.getUserPrincipalId(directoryService, uid, username);
        var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters/' + clusterId +
        ↪  '/clusterroletemplatebindings', null);
        for(var i = 0; i < resObj.data.length; i++) {
                if(resObj.data[i].userPrincipalId === userPrincipalId) {
                        clusterRoleTemplateBindingId = resObj.data[i].id;
                        break;
                }
        }
        if(RANCHER.isEmptyString(clusterRoleTemplateBindingId)) {
                throw 'Cluster role template binding id with user principal id \"' +
                ↪  userPrincipalId + '\" not found!';
        }
}

/* Delete Cluster User */
resObj = RANCHER.executeRequest(restHost, 'DELETE', '/clusterroletemplatebindings/' +
↪  clusterRoleTemplateBindingId, null);
result = resObj.id === clusterRoleTemplateBindingId;
```

Auflistung A.14: *Workflow „rancher_create_project“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Create Project */
RANCHER.checkProjectName(projectName);
requestContent = {
        "clusterId": clusterId,
        "name": projectName
}
var resObj = RANCHER.executeRequest(restHost, 'POST', '/projects', requestContent);
projectId = resObj.id;
```

Auflistung A.15: *Workflow „rancher_delete_project“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Get Project Id */
projectId = RANCHER.getProjectId(restHost, clusterId, projectId, projectName);

/* Delete Project */
var resObj = RANCHER.executeRequest(restHost, 'DELETE', '/projects/' + projectId, null);
result = resObj.id === projectId;
```

Auflistung A.16: *Workflow „rancher_create_project_user“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
```

```
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Get Project Id */
projectId = RANCHER.getProjectId(restHost, clusterId, projectId, projectName);

/* Create Project User */
var requestContent = {
        "projectId": projectId,
        "roleTemplateId": RANCHER.getRoleTemplateId(roleTemplate),
        "userPrincipalId": RANCHER.getUserPrincipalId(directoryService, uid, username)
}
resObj = RANCHER.executeRequest(restHost, 'POST', '/projectroletemplatebindings',
↪  requestContent);
projectRoleTemplateBindingId = resObj.id;
```

Auflistung A.17: *Workflow „rancher_delete_project_user“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Get Project Id */
projectId = RANCHER.getProjectId(restHost, clusterId, projectId, projectName);

/* Get Project Role Template Binding Id */
if(RANCHER.isEmptyString(projectRoleTemplateBindingId)) {
        var userPrincipalId = RANCHER.getUserPrincipalId(directoryService, uid, username);
        var resObj = RANCHER.executeRequest(restHost, 'GET', '/projects/' + projectId +
        ↪  '/projectroletemplatebindings', null);
        for(var i = 0; i < resObj.data.length; i++) {
                if(resObj.data[i].userPrincipalId === userPrincipalId) {
                        projectRoleTemplateBindingId = resObj.data[i].id;
                        break;
                }
        }
        if(RANCHER.isEmptyString(projectRoleTemplateBindingId)) {
                throw 'Project role template binding id with user principal id \"' +
                ↪  userPrincipalId + '\" not found!';
        }
}

/* Delete Project User */
resObj = RANCHER.executeRequest(restHost, 'DELETE', '/projectroletemplatebindings/' +
↪  projectRoleTemplateBindingId, null);
result = resObj.id === projectRoleTemplateBindingId;
```

Auflistung A.18: *Workflow „rancher_create_worker“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Get Cluster Name */
if(RANCHER.isEmptyString(clusterName)) {
        var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters/' + clusterId, null);
        clusterName = resObj.name;
```

```
        if(RANCHER.isEmptyString(clusterName)) {
                throw 'Cluster with the id \"' + clusterId + '\" not found!';
        }
} else {
        RANCHER.checkClusterName(clusterName);
}

/* Create Node Pool */
requestContent = {
    "clusterId": clusterId,
    "hostnamePrefix": clusterName + '-worker-' + Date.now() + '-',
    "nodeTemplateId": RANCHER.getNodeTemplateId(nodePoolTemplate),
    "quantity": 3,
    "worker": true
};
resObj = RANCHER.executeRequest(restHost, 'POST', '/nodepools', requestContent);
nodePoolId = resObj.id;
```

Auflistung A.19: *Workflow „rancher_delete_worker“* im *vRO*

```
const RANCHER = System.getModule('de.awi.rancher');

/* Get Cluster Id */
clusterId = RANCHER.getClusterId(restHost, clusterId, clusterName, rancherCluster);

/* Get Node Pool Id */
if(RANCHER.isEmptyString(nodePoolId) && !RANCHER.isEmptyString(nodePoolName)) {
        RANCHER.checkNodePoolName(nodePoolName);
        var resObj = RANCHER.executeRequest(restHost, 'GET', '/clusters/' + clusterId +
        ↪  '/nodepools', null);
        for(var i = 0; i < resObj.data.length; i++) {
                if(resObj.data[i].hostnamePrefix === nodePoolName) {
                        nodePoolId = resObj.data[i].id;
                }
        }
        if(RANCHER.isEmptyString(nodePoolId)) {
                throw 'Node Pool Id with the name \"' + nodePoolName + '\" not found!';
        }
} else if(RANCHER.isEmptyString(nodePoolId) && RANCHER.isEmptyString(nodePoolName)) {
        throw 'Node pool name is missing!';
}

/* Delete Node Pool */
var resObj = RANCHER.executeRequest(restHost, 'DELETE', '/nodepools/' + nodePoolId, null);
result = resObj.id === nodePoolId;
```

A.5 *GitLab*

Die Auflistung A.20 zeigt die vollständige *CI / CD Pipeline* (*.gitlab-ci.yml*) des Minimalbeispiels „*Voting App*“ [Doc20g] innerhalb der *DevOps*-Plattform *GitLab*. Die Pipeline-Verarbeitung umfasst dabei drei *Stages*, die dem *Master Branch* zugeordnet sind. Im ersten Schritt wird die originale Container-Anwendung mit Hilfe des Quellcodes gebaut und in die AWI-*Registry* hochgeladen. Im zweiten Schritt werden mit Hilfe von *Dockerfiles*, die die jeweilige Anwendung nicht als *Root*-Benutzer ausführen, Container-*Images* neu gebaut und wieder in die AWI-*Registry* hochgeladen. Im letzten Schritt wird die auf *Docker*-basierende Container-Anwendung in einem

Kubernetes-Cluster bereitgestellt. Weitere Details sind hierzu im unteren Abschnitt 8.2 zu erlangen.

Auflistung A.20: *CI / CD Pipeline* des Minimalbeispiels „*Voting App*“ [Doc20g]

```
variables:
  REGISTRY_SERVER: hub.awi.de
  REGISTRY_USER: robot$$ci-cd
  REGISTRY_PATH: fmangels-master
  APP_ORG_DIR: app-org
  APP_ROOTLESS_DIR: app-rootless
  K8S_SPECIFICATIONS_DIR: $APP_ROOTLESS_DIR/k8s-specifications
  K8S_NAMESPACE: vote
  K8S_DOCKER_REGISTRY_SECRET: awi-registry
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: ""
  DOCKER_HOST: tcp://localhost:2375

stages:
  - build_org
  - build_rootless
  - deploy

# Building the original application.
build_app-org:
  stage: build_org
  image: docker:latest
  services:
    - docker:dind
  before_script:
    - docker login -u "$REGISTRY_USER" -p "$REGISTRY_TOKEN" "$REGISTRY_SERVER"
  script:
    - for IMAGE_DIR in $(ls "$CI_PROJECT_DIR/$APP_ORG_DIR/");
        do
          if [ -d "$CI_PROJECT_DIR/$APP_ORG_DIR/$IMAGE_DIR" ] && [ -f
          ↪  "$CI_PROJECT_DIR/$APP_ORG_DIR/$IMAGE_DIR/Dockerfile" ];
            then
              echo ">> Building docker image $IMAGE_DIR <<";
              IMAGE_NAME="$REGISTRY_SERVER/$REGISTRY_PATH/$IMAGE_DIR-org:latest";
              docker build --pull -t "$IMAGE_NAME"
              ↪  "$CI_PROJECT_DIR/$APP_ORG_DIR/$IMAGE_DIR/.";
              echo ">> Pushing to registry $REGISTRY_SERVER <<";
              docker push "$IMAGE_NAME";
              echo ">> Deleting local image $IMAGE_DIR <<";
              docker image rm "$IMAGE_NAME";
            else
              echo ">> Problem with building docker image $IMAGE_DIR <<";
          fi
      done
  only:
    - master
  when: manual

# Building the rootless application.
build_app-rootless:
  stage: build_rootless
  image: docker:latest
  services:
    - docker:dind
  before_script:
    - docker login -u "$REGISTRY_USER" -p "$REGISTRY_TOKEN" "$REGISTRY_SERVER"
```

```
55    script:
      - for IMAGE_DIR in $(ls "$CI_PROJECT_DIR/$APP_ROOTLESS_DIR/");
57        do
            if [ -d "$CI_PROJECT_DIR/$APP_ROOTLESS_DIR/$IMAGE_DIR" ] && [ -f
            ↪  "$CI_PROJECT_DIR/$APP_ROOTLESS_DIR/$IMAGE_DIR/Dockerfile" ];
59            then
                echo ">> Building docker image $IMAGE_DIR <<";
61              IMAGE_NAME="$REGISTRY_SERVER/$REGISTRY_PATH/$IMAGE_DIR-rootless:latest";
                docker build --pull -t "$IMAGE_NAME"
                ↪  "$CI_PROJECT_DIR/$APP_ROOTLESS_DIR/$IMAGE_DIR/.";
63              echo ">> Pushing to registry $REGISTRY_SERVER <<";
                docker push "$IMAGE_NAME";
65              echo ">> Deleting local image $IMAGE_DIR <<";
                docker image rm "$IMAGE_NAME";
67            else
                echo ">> Problem with building docker image $IMAGE_DIR <<";
69          fi
        done
71    only:
      - master
73
  # Deploying the rootless application.
75  deploy_app-rootless:
    stage: deploy
77    image: alpine:latest
    before_script:
79      - apk add --update --no-cache curl
      - curl -L "https://storage.googleapis.com/kubernetes-release/release/`curl -s https://st
      ↪  orage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl"
      ↪  -o /usr/local/bin/kubectl
81    - chmod +x /usr/local/bin/kubectl
    script:
83      - kubectl create namespace "$K8S_NAMESPACE" --dry-run=client -o yaml | kubectl
      ↪  --kubeconfig "$K8S_CONFIG" apply -f -
      - kubectl -n "$K8S_NAMESPACE" create secret docker-registry
      ↪  "$K8S_DOCKER_REGISTRY_SECRET" --docker-server="$REGISTRY_SERVER"
      ↪  --docker-username="$REGISTRY_USER" --docker-password="$REGISTRY_TOKEN"
      ↪  --dry-run=client -o yaml | kubectl --kubeconfig "$K8S_CONFIG" apply -f -
85    - kubectl --kubeconfig "$K8S_CONFIG" -n "$K8S_NAMESPACE" delete --ignore-not-found=true
      ↪  -f "$CI_PROJECT_DIR/$K8S_SPECIFICATIONS_DIR/."
      - kubectl --kubeconfig "$K8S_CONFIG" -n "$K8S_NAMESPACE" apply -f
      ↪  "$CI_PROJECT_DIR/$K8S_SPECIFICATIONS_DIR/."
87    only:
      - master
```

A.6 CIS *Kubernetes Benchmark*

Die Tabelle A.1 listet die vollständigen Kontrollmaßnahmen der CIS *Kubernetes Benchmark* [CIS19] in der Version *1.5.0* auf. Anzumerken ist, dass nicht alle hier definierten Kontrollen für einen gehärteten RKE-Cluster, bedingt durch die zugrunde liegende Container-Architektur, anwendbar (*Not Applicable*) sind und nur die als „*Scored*“ markierten Reihen für die Bewertung betrachtet werden [Ran20a]. Weiterführende Informationen hierzu enthält der Abschnitt 8.1.

Tabelle A.1: Vollständige Kontrollübersicht der CIS *Kubernetes Benchmark* (*v1.5.0*) [CIS19, vgl. S. 267ff], [Ran20a]

ID	***Controls*** ($\sum$ 122), ***Not Applicable*** ($\sum$ 23, ★)	***Scored*** ($\sum$ 92, ✓) / ***Not Scored*** ($\sum$ 30, ✗)
1	***Control Plane Components***	
1.1	***Master Node Configuration Files***	
1.1.1	*Ensure that the API server pod specification file permissions are set to 644 or more restrictive* ★	✓
1.1.2	*Ensure that the API server pod specification file ownership is set to root:root* ★	✓
1.1.3	*Ensure that the controller manager pod specification file permissions are set to 644 or more restrictive* ★	✓
1.1.4	*Ensure that the controller manager pod specification file ownership is set to root:root* ★	✓
1.1.5	*Ensure that the scheduler pod specification file permissions are set to 644 or more restrictive* ★	✓
1.1.6	*Ensure that the scheduler pod specification file ownership is set to root:root* ★	✓
1.1.7	*Ensure that the etcd pod specification file permissions are set to 644 or more restrictive* ★	✓
1.1.8	*Ensure that the etcd pod specification file ownership is set to root:root* ★	✓
1.1.9	*Ensure that the Container Network Interface file permissions are set to 644 or more restrictive*	✗
1.1.10	*Ensure that the Container Network Interface file ownership is set to root:root*	✗
1.1.11	*Ensure that the etcd data directory permissions are set to 700 or more restrictive*	✓
1.1.12	*Ensure that the etcd data directory ownership is set to etcd:etcd*	✓
1.1.13	*Ensure that the admin.conf file permissions are set to 644 or more restrictive* ★	✓
1.1.14	*Ensure that the admin.conf file ownership is set to root:root* ★	✓
1.1.15	*Ensure that the scheduler.conf file permissions are set to 644 or more restrictive* ★	✓
1.1.16	*Ensure that the scheduler.conf file ownership is set to root:root* ★	✓
1.1.17	*Ensure that the controller-manager.conf file permissions are set to 644 or more restrictive* ★	✓
1.1.18	*Ensure that the controller-manager.conf file ownership is set to root:root* ★	✓
1.1.19	*Ensure that the Kubernetes PKI directory and file ownership is set to root:root*	✓
1.1.20	*Ensure that the Kubernetes PKI certificate file permissions are set to 644 or more restrictive*	✓
1.1.21	*Ensure that the Kubernetes PKI key file permissions are set to 600*	✓
1.2	***API Server***	
1.2.1	*Ensure that the –anonymous-auth argument is set to false*	✗

1.2.2	*Ensure that the –basic-auth-file argument is not set*	✓
1.2.3	*Ensure that the –token-auth-file parameter is not set*	✓
1.2.4	*Ensure that the –kubelet-https argument is set to true*	✓
1.2.5	*Ensure that the –kubelet-client-certificate and –kubelet-client-key arguments are set as appropriate*	✓
1.2.6	*Ensure that the –kubelet-certificate-authority argument is set as appropriate*	✓
1.2.7	*Ensure that the –authorization-mode argument is not set to AlwaysAllow*	✓
1.2.8	*Ensure that the –authorization-mode argument includes Node*	✓
1.2.9	*Ensure that the –authorization-mode argument includes RBAC*	✓
1.2.10	*Ensure that the admission control plugin EventRateLimit is set*	✗
1.2.11	*Ensure that the admission control plugin AlwaysAdmit is not set*	✓
1.2.12	*Ensure that the admission control plugin AlwaysPullImages is set*	✗
1.2.13	*Ensure that the admission control plugin SecurityContextDeny is set if PodSecurityPolicy is not used*	✗
1.2.14	*Ensure that the admission control plugin ServiceAccount is set*	✓
1.2.15	*Ensure that the admission control plugin NamespaceLifecycle is set*	✓
1.2.16	*Ensure that the admission control plugin PodSecurityPolicy is set*	✓
1.2.17	*Ensure that the admission control plugin NodeRestriction is set*	✓
1.2.18	*Ensure that the –insecure-bind-address argument is not set*	✓
1.2.19	*Ensure that the –insecure-port argument is set to 0*	✓
1.2.20	*Ensure that the –secure-port argument is not set to 0*	✓
1.2.21	*Ensure that the –profiling argument is set to false*	✓
1.2.22	*Ensure that the –audit-log-path argument is set*	✓
1.2.23	*Ensure that the –audit-log-maxage argument is set to 30 or as appropriate*	✓
1.2.24	*Ensure that the –audit-log-maxbackup argument is set to 10 or as appropriate*	✓
1.2.25	*Ensure that the –audit-log-maxsize argument is set to 100 or as appropriate*	✓
1.2.26	*Ensure that the –request-timeout argument is set as appropriate*	✓
1.2.27	*Ensure that the –service-account-lookup argument is set to true*	✓
1.2.28	*Ensure that the –service-account-key-file argument is set as appropriate*	✓
1.2.29	*Ensure that the –etcd-certfile and –etcd-keyfile arguments are set as appropriate*	✓
1.2.30	*Ensure that the –tls-cert-file and –tls-private-key-file arguments are set as appropriate*	✓
1.2.31	*Ensure that the –client-ca-file argument is set as appropriate*	✓
1.2.32	*Ensure that the –etcd-cafile argument is set as appropriate*	✓
1.2.33	*Ensure that the –encryption-provider-config argument is set as appropriate*	✓
1.2.34	*Ensure that encryption providers are appropriately configured*	✓
1.2.35	*Ensure that the API Server only makes use of Strong Cryptographic Ciphers*	✗
1.3	***Controller Manager***	

1.3.1	*Ensure that the –terminated-pod-gc-threshold argument is set as appropriate*	✓
1.3.2	*Ensure that the –profiling argument is set to false*	✓
1.3.3	*Ensure that the –use-service-account-credentials argument is set to true*	✓
1.3.4	*Ensure that the –service-account-private-key-file argument is set as appropriate*	✓
1.3.5	*Ensure that the –root-ca-file argument is set as appropriate*	✓
1.3.6	*Ensure that the RotateKubeletServerCertificate argument is set to true*	✓
1.3.7	*Ensure that the –bind-address argument is set to 127.0.0.1*	✓
1.4	***Scheduler***	
1.4.1	*Ensure that the –profiling argument is set to false*	✓
1.4.2	*Ensure that the –bind-address argument is set to 127.0.0.1*	✓
2	***etcd***	
2.1	*Ensure that the –cert-file and –key-file arguments are set as appropriate*	✓
2.2	*Ensure that the –client-cert-auth argument is set to true*	✓
2.3	*Ensure that the –auto-tls argument is not set to true*	✓
2.4	*Ensure that the –peer-cert-file and –peer-key-file arguments are set as appropriate*	✓
2.5	*Ensure that the –peer-client-cert-auth argument is set to true*	✓
2.6	*Ensure that the –peer-auto-tls argument is not set to true*	✓
2.7	*Ensure that a unique Certificate Authority is used for etcd*	✗
3	***Control Plane Configuration***	
3.1	***Authentication and Authorization***	
3.1.1	*Client certificate authentication should not be used for users*	✗
3.2	***Logging***	
3.2.1	*Ensure that a minimal audit policy is created*	✓
3.2.2	*Ensure that the audit policy covers key security concerns*	✗
4	***Worker Nodes***	
4.1	***Worker Node Configuration Files***	
4.1.1	*Ensure that the kubelet service file permissions are set to 644 or more restrictive* ★	✓
4.1.2	*Ensure that the kubelet service file ownership is set to root:root* ★	✓
4.1.3	*Ensure that the proxy kubeconfig file permissions are set to 644 or more restrictive* ★	✓
4.1.4	*Ensure that the proxy kubeconfig file ownership is set to root:root* ★	✓
4.1.5	*Ensure that the kubelet.conf file permissions are set to 644 or more restrictive* ★	✓
4.1.6	*Ensure that the kubelet.conf file ownership is set to root:root* ★	✓
4.1.7	*Ensure that the certificate authorities file permissions are set to 644 or more restrictive*	✓
4.1.8	*Ensure that the client certificate authorities file ownership is set to root:root*	✓
4.1.9	*Ensure that the kubelet configuration file has permissions set to 644 or more restrictive* ★	✓

4.1.10	*Ensure that the kubelet configuration file ownership is set to root:root* ★	✓
4.2	***Kubelet***	
4.2.1	*Ensure that the –anonymous-auth argument is set to false*	✓
4.2.2	*Ensure that the –authorization-mode argument is not set to AlwaysAllow*	✓
4.2.3	*Ensure that the –client-ca-file argument is set as appropriate*	✓
4.2.4	*Ensure that the –read-only-port argument is set to 0*	✓
4.2.5	*Ensure that the –streaming-connection-idle-timeout argument is not set to 0*	✓
4.2.6	*Ensure that the –protect-kernel-defaults argument is set to true*	✓
4.2.7	*Ensure that the –make-iptables-util-chains argument is set to true*	✓
4.2.8	*Ensure that the –hostname-override argument is not set*	✗
4.2.9	*Ensure that the –event-qps argument is set to 0 or a level which ensures appropriate event capture*	✗
4.2.10	*Ensure that the –tls-cert-file and –tls-private-key-file arguments are set as appropriate* ★	✓
4.2.11	*Ensure that the –rotate-certificates argument is not set to false*	✓
4.2.12	*Ensure that the RotateKubeletServerCertificate argument is set to true*	✓
4.2.13	*Ensure that the Kubelet only makes use of Strong Cryptographic Ciphers*	✗
5	***Policies***	
5.1	***RBAC and Service Accounts***	
5.1.1	*Ensure that the cluster-admin role is only used where required*	✗
5.1.2	*Minimize access to secrets*	✗
5.1.3	*Minimize wildcard use in Roles and ClusterRoles*	✗
5.1.4	*Minimize access to create pods*	✗
5.1.5	*Ensure that default service accounts are not actively used.*	✓
5.1.6	*Ensure that Service Account Tokens are only mounted where necessary*	✗
5.2	***Pod Security Policies***	
5.2.1	*Minimize the admission of privileged containers*	✗
5.2.2	*Minimize the admission of containers wishing to share the host process ID namespace*	✓
5.2.3	*Minimize the admission of containers wishing to share the host IPC namespace*	✓
5.2.4	*Minimize the admission of containers wishing to share the host network namespace*	✓
5.2.5	*Minimize the admission of containers with allowPrivilegeEscalation*	✓
5.2.6	*Minimize the admission of root containers*	✗
5.2.7	*Minimize the admission of containers with the NET_RAW capability*	✗
5.2.8	*Minimize the admission of containers with added capabilities*	✗
5.2.9	*Minimize the admission of containers with capabilities assigned*	✗
5.3	***Network Policies and CNI***	
5.3.1	*Ensure that the CNI in use supports Network Policies*	✗
5.3.2	*Ensure that all Namespaces have Network Policies defined*	✓

5.4	***Secrets Management***	
5.4.1	*Prefer using secrets as files over secrets as environment variables*	✗
5.4.2	*Consider external secret storage*	✗
5.5	***Extensible Admission Control***	
5.5.1	*Configure Image Provenance using ImagePolicyWebhook admission controller*	✗
5.6	***General Policies***	
5.6.1	*Create administrative boundaries between resources using namespaces*	✗
5.6.2	*Ensure that the seccomp profile is set to docker/default in your pod definitions*	✗
5.6.3	*Apply Security Context to Your Pods and Containers*	✗
5.6.4	*The default namespace should not be used*	✓

A.7 *Google Forms*-Umfrage

Für eine möglichst repräsentative Bewertung der realisierten *Kubernetes*-Infrastruktur bezogen auf die Verwendbarkeit und den möglichen Einschränkungen, wurde eine *Google Forms*-Umfrage [Goo20a] erstellt. Eine Kopie der Umfrage ist in diesem Abschnitt nachfolgend abgebildet und gliedert sich in wesentlich sechs Abschnitten auf. Die ersten beiden Abschnitte besitzen einen erklärenden Charakter und beinhalten allgemeine Fragestellungen zum Kenntnisstand des Teilnehmers. Anschließend werden nacheinander die einzelnen involvierten Interaktionsplattformen abgearbeitet: *Harbor* als *Image Registry*, *Rancher* mit RKE als *Cluster Management*, die *VMware vRealize Suite* (*vRA*, *vRO*) als *Self-Service Portal* und *GitLab* als *CI / CD Pipeline*. Zu jeder Plattform sind im Vorfeld Arbeitsaufträge platziert, damit Erfahrungen mit den einzelnen Bestandteilen der Infrastruktur gesammelt werden können, bevor es zu den eigentlichen Fragestellungen geht. Die erstellte Umfrage wird ausführlicher im Abschnitt 8.2 erläutert und abschließend mit einer Bewertung versehen.

Evaluation der Kubernetes-Infrastruktur

Sehr geehrte Damen und Herren,

während meiner Masterarbeit im Studiengang Informatik (KSS) an der Hochschule Bremen und in Kooperation mit dem Alfred-Wegener-Institut, Helmholtz-Zentrum für Polar- und Meeresforschung, habe ich mich mit der Sicherheit und der automatisierten Bereitstellung eines Kubernetes-Clusters beschäftigt. An der Realisierung dieser Infrastruktur sind wesentlich vier Plattformen beteiligt: Ein Self-Service Portal, ein Cluster Management, eine Image Registry und eine CI / CD Pipeline. In Folge einer Evaluation möchte ich von Ihnen gerne die Verwendbarkeit und die möglichen Einschränkungen dieser Interaktionsplattformen erfahren. Ich freue mich, wenn Sie sich ein wenig Zeit nehmen, um die nachstehenden Aufgaben zu bearbeiten. Anschließend bitte ich Sie darum die jeweils dazu gestellten Fragen zu beantworten.

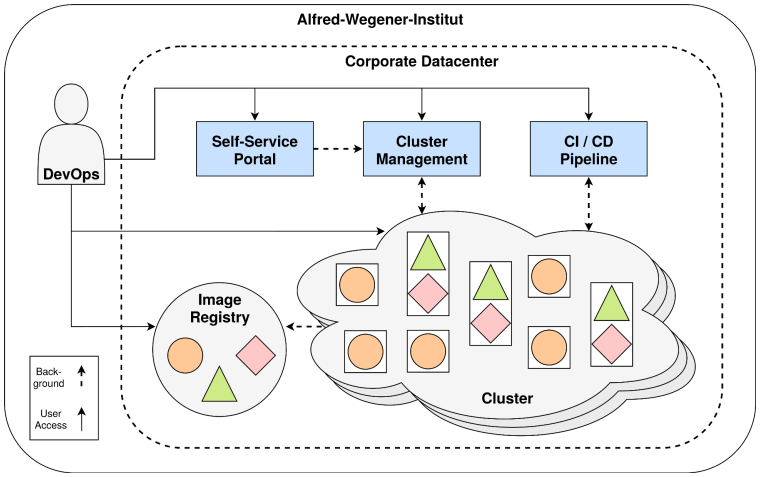
Ich danke Ihnen vielmals.

Titel der Masterarbeit: Analyse der Sicherheit und der automatisierten Bereitstellung eines On-Premises-Clusters auf der Grundlage der Container-basierten Virtualisierung: Kubernetes im Wissenschaftsbetrieb.

Fabian Mangels <fabian@mangels.it, fmangels@awi.de>

* Erforderlich

Kontextdiagramm des Szenarios



1. Mit welchen Container-Technologien konnten Sie bereits Erfahrungen sammeln? *

   Wählen Sie alle zutreffenden Antworten aus.

   - [ ] Docker
   - [ ] Kubernetes
   - [ ] Keine Angabe
   - [ ] Sonstiges: _____

2. Wie schätzen Sie Ihre Kompetenz im Umgang mit Container-Technologien ein? *

   Markieren Sie nur ein Oval.

	1	2	3	4	5	
Sehr schlecht	◯	◯	◯	◯	◯	Sehr gut

3. Stellen Sie bereits Software-Anwendungen basierend auf Docker-Containern bereit? *

   Markieren Sie nur ein Oval.

   - ◯ Ja
   - ◯ Nein

4. Stellen Sie bereits Software-Anwendungen in einem Kubernetes-Cluster bereit? *

   Markieren Sie nur ein Oval.

   - ◯ Ja
   - ◯ Nein

5. Setzen Sie bereits CI / CD Pipelines für Ihre Software-Projekte ein? *

Markieren Sie nur ein Oval.

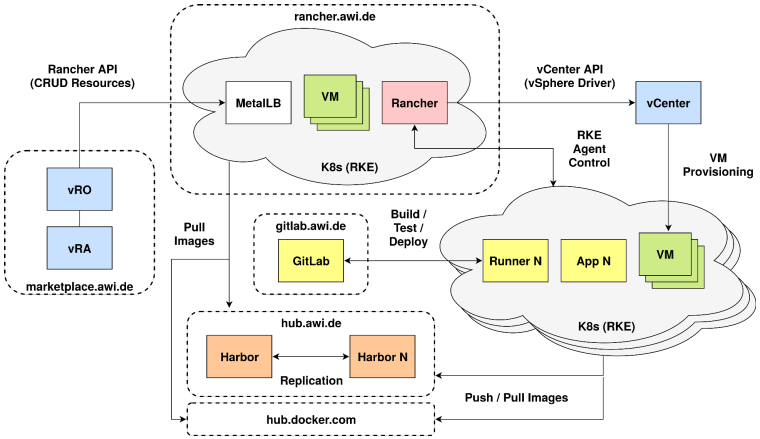
- ◯ Ja
- ◯ Nein

Realisierung der Infrastruktur

Wie bereits im Einleitungstext erwähnt, setzt sich die realisierte Infrastruktur aus mehreren Interaktionsplattformen zusammen, die unter einander Abhängigkeiten aufweisen. In den nachfolgenden Abschnitten werden die vier beteiligten Plattformen betrachtet und die Möglichkeit gegeben, hierzu jeweils Aufgaben zu absolvieren. Abgeschlossen werden die Abschnitte wiederum durch die Beantwortung einiger Fragen.

Self-Service Portal --- VMware vRealize Suite mit vRA & vRO (https://marketplace.awi.de)
Cluster Management --- Rancher mit RKE (https://rancher.awi.de)
Image Registry --- Harbor (https://hub.awi.de)
CI / CD Pipeline --- GitLab (https://gitlab.awi.de)

Abhängigkeiten der Interaktionsplattformen innerhalb des Szenarios



6. Mit welchen konkreten Plattformen konnten Sie bereits Erfahrungen sammeln? *

Wählen Sie alle zutreffenden Antworten aus.

- ☐ VMware vRealize Suite (AWI Marketplace)
- ☐ Rancher
- ☐ Harbor
- ☐ GitLab
- ☐ Keine Angabe

Self-Service Portal

URL: https://marketplace.awi.de

7. Aufgaben --- Self-Service Portal:

Wählen Sie alle zutreffenden Antworten aus.

- ☐ 1. Melden Sie sich mit Ihren AWI-Credentials am Self-Service Portal unter der obigen URL an.
- ☐ 2. Rufen Sie den Ressourcen-Katalog unter der Rubrik "Catalog" auf.
- ☐ 3. Suchen und fordern Sie über den Katalog einen "K8s Cluster" durch Bestätigen des "REQUEST"-Links an.
- ☐ 4. Wählen Sie passende Templates ("Cluster", "Node Pool") aus und schicken Sie Ihre Anforderung durch Drücken des "SUBMIT"-Buttons los.
- ☐ 5. Warten Sie bis der Vorgang abgeschlossen ist.
- ☐ 6. Machen Sie sich mit den weiteren Aktionen ("ACTIONS", Day-2 Operations) der angeforderten Cluster-Ressource unter der Rubrik "Deployments" vertraut. Die angeforderte Cluster-Ressource sollte aber, um die nächsten Aufgaben abschließen zu können, nicht gelöscht werden.

Fragen --- Self-Service Portal:

8. Die gestellten Aufgaben mit dem Self-Service Portal konnten ohne Probleme von Ihnen erledigt werden. *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Stimme ich gar nicht zu	◯	◯	◯	◯	◯	Stimme ich voll und ganz zu

9. Wo traten bei Ihnen Probleme mit dem Self-Service Portal auf? *

Markieren Sie nur ein Oval.

- ◯ Keine Probleme
- ◯ Sonstiges: _____

10. Welche bestimmten Day-2 Operations fehlen Ihnen, die Sie auf Ihre angeforderte K8s-Cluster-Ressource anwenden würden? *

Markieren Sie nur ein Oval.

- ◯ Keine Angabe
- ◯ Sonstiges: _____

11. Gibt es von Ihrer Seite aus Verbesserungsvorschläge oder Wünsche bezogen auf das Self-Service Portal und der Anforderung bzw. Verwaltung eines K8s-Clusters hierüber?

_____

_____

_____

_____

_____

Cluster Management

URL: https://rancher.awi.de

12. Aufgaben --- Cluster Management:

Wählen Sie alle zutreffenden Antworten aus.

- ☐ 1. Melden Sie sich mit Ihren AWI-Credentials am Cluster Management unter der obigen URL an.
- ☐ 2. Machen Sie sich mit Ihrem zuvor angeforderten K8s-Cluster vertraut. Wählen Sie dafür unter "Global" Ihren Cluster aus und navigieren Sie durch die verschiedenen Rubriken des Menüs.
- ☐ 3. Rufen Sie in Ihrem Cluster das Projekt "Default" unter der Rubrik "Projects/Namespaces" auf.
- ☐ 4. Wählen Sie "Workloads" unter der Rubrik "Ressources" aus.
- ☐ 5. Stellen Sie einen neuen Workload im Cluster bereit, indem Sie den Button "Deploy" betätigen.
- ☐ 6. Füllen Sie die Pflichtfelder ("Name", "Docker Image", "Namespace") aus und stellen Sie die Container-Anwendung durch Drücken des Buttons "Launch" bereit. Als Beispiel kann das Docker Image "hello-world" und "Job" als "Workload Type" verwendet werden.
- ☐ 7. Rufen Sie die Logging-Informationen durch die Aktion "View Logs" an der entsprechenden Pod- bzw. Container-Ressource auf und betrachten Sie das Ergebnis.
- ☐ 8. Sie können sich von der bereitgestellten Anwendung auch das zugrunde liegende K8s-Manifest durch die Aktion "View YAML" anzeigen lassen.
- ☐ 9. Legen Sie für eine spätere Aufgabe ein neues Projekt mit dem Namen "GitLab Runner" und dem Namespace "gitlab-runner" an. Das Anlegen des Projektes könnten Sie zum Beispiel auch über das Self-Service Portal veranlassen. Wenn Sie wollen können Sie noch weitere Aktionen ausprobieren.

Optional --- Verwenden von "kubectl":

Der K8s-Cluster kann auch nativ mit Hilfe des Tools "kubectl" verwaltet und u. a. Anwendungsbereitstellungen vorgenommen werden. Unter der Rubrik "Cluster" kann das dazu benötigte "Kubeconfig File" heruntergeladen werden, oder es wird hier direkt eine kubectl-Session im Browser mit "Launch kubectl" gestartet.

Optional, Installation von "kubectl" auf Ihrem Rechner:
>>>
sudo curl -L "https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl" -o /usr/local/bin/kubectl
sudo chmod +x /usr/local/bin/kubectl
echo "source <(kubectl completion bash)" >> ~/.bashrc
<<<
(Anleitungen für andere Plattformen: https://kubernetes.io/docs/tasks/tools/install-kubectl/)

Fragen --- Cluster Management:

13. Die gestellten Aufgaben mit dem Cluster Management konnten von Ihnen ohne Probleme erledigt werden. *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Stimme ich gar nicht zu	◯	◯	◯	◯	◯	Stimme ich voll und ganz zu

14. Wo traten bei Ihnen Probleme mit dem Cluster Management auf? *

Markieren Sie nur ein Oval.

- ◯ Keine Probleme
- ◯ Sonstiges: _____

15. Stellt es für Sie ein Problem dar, Docker-Images unprivilegiert und ohne einen Root-Nutzer auszuführen? (Pod Security Policies, PSP) *

Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein
- ◯ Keine Angabe

16. Sind die möglichen Aktionen innerhalb des Cluster Managements zu stark eingeschränkt? *

Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein
- ◯ Keine Angabe

17. Wären Sie damit einverstanden, Container-Images nur von einer Private Registry in einem K8s-Cluster zuzulassen? Die direkte Verwendung von "öffentlichen" Images (Bspw. von Docker Hub) wäre dann ohne das vorherige Hochladen und Anpassen in eine Private Registry nicht möglich. *

Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein
- ◯ Keine Angabe

18. Gibt es von Ihrer Seite aus Verbesserungsvorschläge oder Wünsche bezogen auf das Cluster Management?

_____

_____

_____

_____

_____

Image Registry

URL: https://hub.awi.de

Voraussetzung --- Image Registry

Für die nächsten Aufgaben wird die Docker Runtime benötigt. Falls sie nicht auf Ihren System vorhanden ist, kann sie unter Linux (Ubuntu) wie folgt installiert werden:
>>>
curl -fsSL "https://download.docker.com/linux/ubuntu/gpg" | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
sudo apt -y update
sudo apt -y install docker-ce docker-ce-cli containerd.io
<<<
(Anleitungen für andere Plattformen: https://docs.docker.com/get-docker/)

19. Aufgaben --- Image Registry:

Wählen Sie alle zutreffenden Antworten aus.

- [ ] 1. Melden Sie sich mit Ihren AWI-Credentials an der Image Registry unter der obigen URL an.
- [ ] 2. Erstellen Sie ein neues Projekt für Ihre zukünftigen Repositories unter der Rubrik "Projects" durch Drücken des Buttons "+ NEW PROJECT".
- [ ] 3. Wählen Sie Ihr Projekt aus und navigieren Sie auf die Rubrik "Robot Accounts".
- [ ] 4. Erstellen Sie einen neuen Robot Account durch Drücken des Buttons "+ NEW ROBOT ACCOUNT". Wählen Sie einen Namen und verifizieren Sie, dass beide Haken unter "Permissions" und dort unter "Artifact" (Push, Pull) gesetzt sind.
- [ ] 5. Wenn Sie wollen können Sie noch weitere Einstellungen für Ihr Projekt setzen, z. B. kann unter der Rubrik "Configuration" ein "Vulnerability scanning" mit den integrierten CVE-Scannern (Trivy, Claire) automatisch nach einem Image-Push aktiviert werden.
- [ ] 6. Melden Sie sich an der Image Registry mit Ihren oder den Robot Account-Credentials über die Kommandozeile an.
- [ ] 7. Laden Sie ein vorhandenes Docker-Image in Ihr eben erstelltes Projekt hoch. Als Beispiel können Sie das Docker-Image "hello-world" aus der öffentlichen Docker Hub Registry herunterladen und mit einem neuen Tag versehen.
- [ ] 8. Sehen Sie sich ihr Projekt und das neu angelegte Repository an. Es sollte ein neuer Eintrag vorhanden sein.
- [ ] 9. Lassen Sie jetzt das Image beim Hochladen automatisch durch Notary signieren. Hierfür müssen die Umgebungsvariablen "DOCKER_CONTENT_TRUST" und "DOCKER_CONTENT_TRUST_SERVER" gesetzt werden. Beim erstmaligen Verwenden von Docker Content Trust, müssen zudem ein "root key" und entsprechend ein "repository key" erstellt werden.
- [ ] 10. Sehen Sie sich ihr Projekt erneut an. Es wurde im eben erstellten Repository ein neuer Image-Tag angelegt, welches durch Docker Notary signiert wurde.
- [ ] 11. Die Signierung kann bei gesetzten Umgebungsvariablen durch ein "docker trust revoke" wieder entfernt werden; das zugehörige Passwort und die erstellten Zertifikate müssen hierfür vorhanden sein.
- [ ] 12. Um spätere Probleme mit anderen Docker-Registries zu vermeiden, sollten die Umgebungsvariablen wieder gelöscht werden.

Code-Listings --- Image Registry:

Zu 6.
>>>
docker login hub.awi.de
<<<

Zu 7.
>>>
docker pull hello-world
docker tag hello-world:latest hub.awi.de/PROJECT/REPOSITORY:latest
docker push hub.awi.de/PROJECT/REPOSITORY:latest
<<<

Zu 9.
>>>
export DOCKER_CONTENT_TRUST=1
export DOCKER_CONTENT_TRUST_SERVER=https://hub.awi.de:4443

docker tag hello-world:latest hub.awi.de/PROJECT/REPOSITORY:signed
docker push hub.awi.de/PROJECT/REPOSITORY:signed
<<<

Zu 11.
>>>
docker trust revoke hub.awi.de/PROJECT/REPOSITORY:signed
<<<

Zu 12.
>>>
unset DOCKER_CONTENT_TRUST
unset DOCKER_CONTENT_TRUST_SERVER
<<<

Fragen --- Image Registry:

20. Die gestellten Aufgaben mit der Image Registry konnten von Ihnen ohne Probleme erledigt werden. *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Stimme ich gar nicht zu	◯	◯	◯	◯	◯	Stimme ich voll und ganz zu

21. Wo traten bei Ihnen Probleme mit der Image Registry auf? *

Markieren Sie nur ein Oval.

- ◯ Keine Probleme
- ◯ Sonstiges: _____

22. Wie wichtig ist Ihnen die Nutzung eines CVE-Scanners, um Schwachstellen in ihrem Container-Image zu analysieren? *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Unwichtig	◯	◯	◯	◯	◯	Sehr wichtig

23. Wie wichtig ist Ihnen die Möglichkeit Container-Images zu signieren? (Docker Content Trust) *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Unwichtig	◯	◯	◯	◯	◯	Sehr wichtig

24. Gibt es von Ihrer Seite aus Verbesserungsvorschläge oder Wünsche bezogen auf die Image Registry?

_____

_____

_____

_____

_____

CI / CD Pipeline

URL: https://gitlab.awi.de
URL des Beispiel-Projekts: https://gitlab.awi.de/fmangels/k8s-example-ci-cd

25. Aufgaben --- CI / CD Pipeline:

Wählen Sie alle zutreffenden Antworten aus.

- ☐ 1. Melden Sie sich mit Ihren AWI-Credentials an der DevOps-Plattform GitLab unter der obigen URL an.
- ☐ 2. Legen Sie ein neues Projekt durch Drücken des Buttons "New project", Ausfüllen des anschließenden Formulars und Bestätigen des Buttons "Create project" an.
- ☐ 3. Wählen Sie Ihr neu erstelltes Projekt aus und navigieren Sie zu den Runner-Einstellungen unter der Rubrik "Settings", dann "CI /CD" und dort unter "Runners".
- ☐ 4. Notieren Sie sich unter "Set up a specific Runner manually" den "Registration Token" für einen solchen Runner.
- ☐ 5. Installieren Sie sich das Tool "helm" auf Ihrem Rechner.
- ☐ 6. Laden Sie sich das "Kubeconfig File" aus dem Cluster Management herunter.
- ☐ 7. Installieren Sie einen neuen GitLab Runner in Ihrem K8s-Cluster und im Namespace "gitlab-runner" mit Hilfe des Helm Chart Repositories "https://charts.gitlab.io", den notierten "Registration Token" sowie dem "Kubeconfig File". (Vorher muss ich noch temporär erweiterte Berechtigungen in ihrem Cluster setzen!)
- ☐ 8. Es sollte nun ein Eintrag unter "Runners activated for this project" in den Einstellungen des GitLab-Projekts auftauchen. Auch im Cluster Management sollte ein neuer Pod im Projekt "GitLab Runner" vorhanden sein.
- ☐ 9. Fügen Sie nun Quellcode einer Container-Anwendung Ihrem GitLab-Projekt hinzu. Hierzu gibt es ein Beispiel-Projekt (s. obige URL), welches Quellcode für einen einfachen Nginx-Webserver und eine CI / CD Pipeline enthält.
- ☐ 10. Passen Sie die CI /CD Pipeline ".gitlab-ci.yml" durch das Setzen der Umgebungsvariablen ("REGISTRY_USER", "REGISTRY_PATH") an und platzieren Sie die Datei in das Wurzelverzeichnis Ihres GitLab-Projektes. Die folgenden Variablen sind bzgl. der Sicherheit unter "Settings", dann "CI /CD" und dort unter "Variables" anzulegen: "REGISTRY_TOKEN" und "K8S_CONFIG".
- ☐ 11. Starten Sie die einzelnen Stages der neuen CI / CD Pipeline unter der Rubrik "CI / CD" und dort unter "Pipelines". Die "Build Stage" sollte zumindest einmal vor der "Deploy Stage" erfolgreich ausgeführt werden.
- ☐ 12. Schauen Sie sich im Cluster Management im Projekt "Default" um, hier sollte jetzt der Nginx-Webserver als Deployment mit einem Pod und einem Container auftauchen.

Code-Listings --- CI / CD Pipeline:

Zu 5.
>>>
curl -LO "https://get.helm.sh/helm-v3.2.4-linux-amd64.tar.gz"
tar -zxvf helm-v3.2.4-linux-amd64.tar.gz
cd linux-amd64/ && sudo cp helm /usr/local/bin/helm
sudo chmod +x /usr/local/bin/helm
<<<
(Anleitungen für andere Plattformen: https://helm.sh/docs/intro/install/)

Zu 7.
>>>
helm repo add gitlab https://charts.gitlab.io
helm --kubeconfig ... install --namespace gitlab-runner gitlab-runner gitlab/gitlab-runner --set gitlabUrl="https://gitlab.awi.de/" --set runnerRegistrationToken="..." --set rbac.create=true --set runners.privileged=true
<<<

Fragen --- CI / CD Pipeline:

26. Die gestellten Aufgaben mit einer CI / CD Pipeline konnten von Ihnen ohne Probleme erledigt werden. *

Markieren Sie nur ein Oval.

	1	2	3	4	5	
Stimme ich gar nicht zu	◯	◯	◯	◯	◯	Stimme ich voll und ganz zu

27. Wo traten bei Ihnen Probleme mit einer CI / CD Pipeline auf? *

Markieren Sie nur ein Oval.

◯ Keine Probleme
◯ Sonstiges: _____

28. Würden Sie es befürworten, bereits vorhandene GitLab Runner (Shared Runners) für Ihr GitLab-Projekt ohne eine manuelle Installation zu verwenden? *

Markieren Sie nur ein Oval.

◯ Ja
◯ Nein
◯ Keine Angabe

29. Gibt es von Ihrer Seite aus Verbesserungsvorschläge oder Wünsche bezogen auf eine CI / CD Pipeline?

_____
_____
_____
_____
_____

Vielen Dank!

30. Haben Sie noch abschließende Bemerkungen?

_____
_____
_____
_____
_____

A.8 *KubeFed*

Mit *KubeFed* (*Kubernetes Cluster Federation*) [Kub20c] wird es möglich *K8s*-Cluster föderiert und geografisch verteilt zu betreiben. Neben dem gewöhnlichen Betrieb, Cluster über verschiedene Brandschutzzonen hinweg in einem Rechenzentrum (RZ) zu verteilen, kann die Aufteilung mit Hilfe der *KubeFed*-Architektur auch geografisch über mehrere RZ-Standorte oder sogar über noch größere Distanzen geschehen.

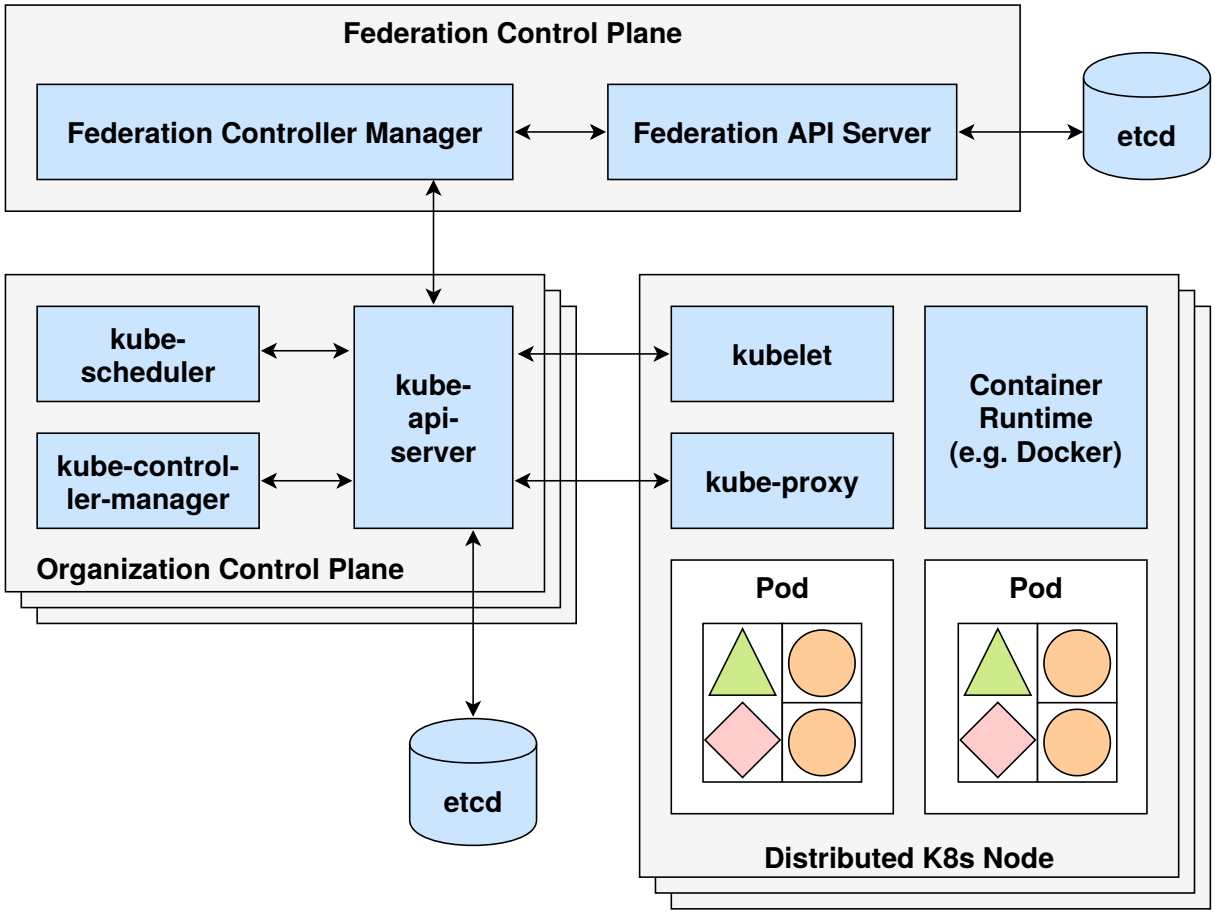


Abbildung A.1: *K8s*-Cluster in einer Föderation [Luk18, vgl. S. 604], [Lie19, vgl. S. 1164]

Der schematische Aufbau dieser Architektur kann der Abbildung A.1 entnommen werden. Im Grunde genommen beinhaltet der föderierte Aufbau einen Cluster aus wiederum mehreren Clustern und unterscheidet sich nicht wesentlich von der regulären *K8s*-Architektur wie sie im Abschnitt 3.5 ersichtlich ist. Anstelle der einzelnen Knoten sind hingegen Cluster vorzufinden. Eine *KubeFed* besteht aus einer übergeordneten Steuerungsebene (*Federation Control Plane*, FCP) und mehreren untergeordneten *Kubernetes*-Clustern, die ebenso eine Steuerungsebene (*Organization Control Plane*) enthalten. Die Delegation von föderierten Arbeitsaufträgen erfolgt dementsprechend hierarchisch von oben nach unten. Die FCP verwaltet Aufträge eines untergeordneten *Kubernetes*-Clusters und diese Cluster verwalten entsprechend die Anwendungen auf ihren Arbeitsknoten. Als Schnittstelle für die Orchestrierung dient ein *Federation API Server*, der die gleiche Funktionalität wie der *kube-api-server* besitzt, nur eine Ebene höher betrachtet. Des Weiteren kommt ein *Federation Controller Manager* zum Einsatz, welcher analog zum *kube-controller-manager* existiert. Der letzte Bestandteil der FCP ist ein Schlüsselwertspeicher in Ausprägung von *etcd*, um die Verbund-API-Objekte konsistent abzulegen. Für den übergreifenden *K8s*-Betrieb in einer Föderation nimmt in diesem Zusammenhang außerdem ein DNS eine zentrale

Rolle ein; dies wird in der Abbildung allerdings nicht betrachtet. [Luk18, vgl. S. 603f.], [Lie19, vgl. S. 1163ff]

Zu den Vorteilen des Einsatzes von einer Multi-*Cloud* bzw. eines *Federated Kubernetes* gehört die hohe Verfügbarkeit (HA) und die erweiterte Ausfallsicherheit (FT). Hiermit wird es möglich Cluster über mehrere RZ-Standorte oder öffentliche *Clouds* hinweg zu verschalten und eine gemeinsame Schnittstelle für die Verwaltung zu verwenden. Darüber hinaus wird der *Lock-in* eines *Cloud*-Anbieters vermieden und *Kubernetes* fungiert als Abstraktionsebene, indem die Komplexität sowie die einzelnen Details der zugrunde liegenden Anbieter bzw. RZs verborgen bleiben. Außerdem wird es möglich einen Cluster am eigenen Standort mit einem Cluster in der Infrastruktur eines *Cloud*-Anbieters zu kombinieren. So können sensible Komponenten lokal ausgeführt und weniger schützenswerte Bestandteile in öffentlichere Umgebungen verlagert werden. Es ist auch durchaus vorstellbar in Anbetracht der Skalierbarkeit, eine Anwendung anfänglich in einem kleinen lokalen Cluster bereitzustellen und sobald die Kapazitätsanforderungen überschritten werden, das Konstrukt auf öffentliche *Cloud*-Anbieter flexibel auszudehnen. Der Euphorie zum Trotz muss gesagt werden, dass *KubeFed* für das *On-Premise*-Umfeld noch nicht wirklich ausgereift ist und dementsprechend keine stabile Version existiert [Lie19, vgl. S. 1166f.]. Übergangsweise kann das Betreiben mehrerer *Kubernetes*-Instanzen bei den großen *Cloud*-Anbietern, die eigene *K8s*-Services bereitstellen, hierfür die bessere Wahl sein.

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet. Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.

Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Bremen, den 15.09.2020

Fabian Mangels <fabian@mangels.it>
Matrikel-Nr.: 5074215