Dem Fachbereich IV der Universität Trier
zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
eingereichte Dissertation

Modellbildung und Umsetzung von Methoden zur energieeffizienten Nutzung von Containertechnologien

vorgelegt von

Sandro Kreten, M. Ed.

Matthias-Jacoby-Straße 6
54523 Hetzerath

Datum des Antrages auf Eröffnung des Promotionsverfahrens:
01.04.2021

Betreuer am Fachbereich: Prof. Dr. Peter Sturm
Externer Betreuer an der Hochschule Trier (Standort Birkenfeld): Prof. Dr. Stefan Naumann

Abstract

The use of cloud software and scaled web apps as well as web services has increased extremely in recent years, leading to an increase in high performance cloud data centers. In addition to the improvement of services, this is also reflected in the global power consumption of data centers, which currently amounts to slightly more than 1% (equivalent to about 200 TWh). Forecasts predict a massive increase in the power consumption of cloud data centers in the coming years. The basis of this movement is the acceleration of administration and development, which is partly the result of the use of containers. As the basis for millions of web apps and services, they speed up the scaling, provisioning and updating of cloud services.

In this paper it is shown that containers, in addition to their many technical advantages, offer opportunities for reducing the energy consumption of cloud data centers, which result from an inefficient configuration of containers as well as container runtimes.

Based on a survey and an evaluation of suitable literature, probable problems in the use of containers are uncovered in an initial step. Furthermore, the sensitivity of administrators and developers regarding the energy consumption of container software is determined. Proceeding from the results of the survey and the evaluation, standard scenarios in the container environment are used to investigate the components of the de facto Standards Docker.

Then a model consisting of measurement methodology, recommendations for an efficient configuration of containers and tools is described. The measurement methodology should be easy to apply and support common technology in data centers. Also, the recommendations for action give both developers and administrators the opportunity to decide which components of docker should be used and which could be omitted, in the sense of energy-efficient use and depending on the usage scenario of the containers. The resulting containers can be used in terms of energy efficiency on servers and equally on PCs and embedded systems (as part of IoT and Edge Cloud) and thus not only counteract the previously described problem in the cloud.

The thesis also deals with the behaviour of scaled web applications. Common orchestration tools define static scaling points for applications, which in most cases are based on CPU usage. It can be seen that neither the actual accessibility nor the power consumption of the applications is taken into account. The autoscaler of the open source container orchestration tool Kubernetes is considered, which is extended by a newly developed tool. It shows that a dynamic adjustment of the scaling points can be achieved by means of a pre-evaluation of common usage scenarios as well as information about their power consumption and the accessibility per increasing load.

Finally, an empirical investigation of the generated model in the form of three simulations follows, which should show the effects on the energy consumption of cloud data centers.

Abstract

Die Nutzung von Cloud-Software und skalierten Web-Apps sowie Web-Services hat in den letzten Jahren extrem zugenommen, was zu einem Anstieg der Hochleistungs-Cloud-Rechenzentren führt. Neben der Verbesserung der Dienste spiegelt sich dies auch im weltweiten Stromverbrauch von Rechenzentren wider, der derzeit etwas mehr als 1% (entspricht etwa 200 TWh) beträgt. Prognosen sagen für die kommenden Jahre einen massiven Anstieg des Stromverbrauchs von Cloud-Rechenzentren voraus. Grundlage dieser Bewegung ist die Beschleunigung von Administration und Entwicklung, die unter anderem durch den Einsatz von Containern entsteht. Als Basis für Millionen von Web-Apps und -Services beschleunigen sie die Skalierung, Bereitstellung und Aktualisierung von Cloud-Diensten.
In dieser Arbeit wird aufgezeigt, dass Container zusätzlich zu ihren vielen technischen Vorteilen Möglichkeiten zur Reduzierung des Energieverbrauchs von Cloud-Rechenzentren bieten, die aus einer ineffizienten Konfiguration von Containern sowie Container-Laufzeitumgebungen resultieren.
Basierend auf einer Umfrage und einer Auswertung geeigneter Literatur werden in einem ersten Schritt wahrscheinliche Probleme beim Einsatz von Containern aufgedeckt. Weiterhin wird die Sensibilität von Administratoren und Entwicklern bezüglich des Energieverbrauchs von Container-Software ermittelt. Aufbauend auf den Ergebnissen der Umfrage und der Auswertung werden anhand von Standardszenarien im Containerumfeld die Komponenten des de facto Standards Docker untersucht. Anschließend wird ein Modell, bestehend aus Messmethodik, Empfehlungen für eine effiziente Konfiguration von Containern und Tools, beschrieben. Die Messmethodik sollte einfach anwendbar sein und gängige Technologien in Rechenzentren unterstützen. Darüber hinaus geben die Handlungsempfehlungen sowohl Entwicklern als auch Administratoren die Möglichkeit zu entscheiden, welche Komponenten von Docker im Sinne eines energieeffizienten Einsatzes und in Abhängigkeit vom Einsatzszenario der Container genutzt werden sollten und welche weggelassen werden könnten. Die resultierenden Container können im Sinne der Energieeffizienz auf Servern und gleichermaßen auf PCs und Embedded Systems (als Teil von IoT und Edge Cloud) eingesetzt werden und somit nicht nur dem zuvor beschriebenen Problem in der Cloud entgegenwirken.
Die Arbeit beschäftigt sich zudem mit dem Verhalten von skalierten Webanwendungen. Gängige Orchestrierungswerkzeuge definieren statische Skalierungspunkte für Anwendungen, die in den meisten Fällen auf der CPU-Auslastung basieren. Es wird dargestellt, dass dabei weder die tatsächliche Erreichbarkeit noch der Stromverbrauch der Anwendungen berücksichtigt werden. Es wird der Autoscaler des Open-Source-Container-Orchestrierungswerkzeugs Kubernetes betrachtet, der um ein neu entwickeltes Werkzeug erweitert wird. Es wird deutlich, dass eine dynamische Anpassung der Skalierungspunkte durch eine Vorabauswertung gängiger Nutzungsszenarien sowie Informationen über deren Stromverbrauch und die Erreichbarkeit bei steigender Last erreicht werden kann.
Schließlich folgt eine empirische Untersuchung des generierten Modells in Form von drei Simulationen, die die Auswirkungen auf den Energieverbrauch von Cloud-Rechenzentren darlegen sollen.

Wissenschaftlicher Werdegang des Autors

10/2010 – 09/2013
Bachelor Lehramt Informatik/Mathematik
Universität Trier, 54296 Trier
Bachelorarbeit Thema: Der Körper der algebraischen Zahlen
Abschluss: Bachelor of Education

10/2013 – 08/2015
Master Lehramt Informatik/Mathematik
Universität Trier, 54296 Trier
Masterarbeit Thema: Webbasierte Graphvisualisierung
Abschluss: Master of Education

ab 06/2018
Promotionsvorhaben
Universität Trier, 54296 Trier
Thema: Modellbildung und Umsetzung von Methoden zur energieeffizienten Nutzung von Containertechnologien

Wissenschaftliche Veröffentlichungen des Autors

Guldner, A., Kern, E., Kreten, S., and Naumann, S. (2021). Criteria for Sustainable Software Products: Analysing Software, Informing Users and Politics. Springer.

Mancebo, J., Guldner, A., Kern, E., Kessler, P., Kreten, S., Garcia, F., Calero, C., and Naumann, S. (2019). Assessing the Sustainability of Software - A Method Comparison, chapter 1, pages 1 –16. Springer.

Guldner, A., Kreten, S., Müller, A. and Roth, T. (2019). Lernen in der digitalen Welt - Innovative Unterrichtsprojekte aus der MINT-Forschung. MNU Journal - Ausgabe 03.2019 - ISSN 0025-5866. Verlag Klaus Seeberger.

Gollmer, K., Kreten, S., Stolz, F., Dartmann, G. and Burger, G. (2019). IoT-Workshop: Blueprint for pupils' education in IoT in Big Data Analytics for Cyber-Physical Systems, chapter 15. Elsevier.

Kreten, S., Guldner, A., and Naumann, S. (2018). An Analysis of the Energy Consumption Be-

havior of Scaled, Containerized Web Apps. Sustainability - MDPI, 10 - 2710.

Kreten, S. and Guldner, A. (2017). Resource Consumption Behavior in Modern Concurrency Models. EnviroInfo.

Vorträge

Kreten, S. (Deutsche OpenStack Tage 2019). Energy Efficient Docker Usage. https://youtu.be/kr5mqmMqXpo. 25.11.2019.

Lehraufträge

Sommersemester 2016
Lehrbeauftragter Hochschule Trier (Standort Birkenfeld) „Algorithmen und Datenstrukturen“

Wintersemester 2018 und 2019
Lehrbeauftragter Hochschule Trier (Standort Birkenfeld) „Programmierung II“

Sommersemester 2018, 2019 und 2020
Lehrbeauftragter Hochschule Trier (Standort Birkenfeld) „Theoretische Informatik“

Inhaltsverzeichnis

1 Motivation und Einleitung **2**

1.1 Container und ihre Auswirkungen auf Rechenzentren . . . 3

1.2 Chancen und Risiken . . . 5

1.3 Energiemessungen von Software . . . 8

2 Ziele und Aufbau der Arbeit **10**

2.1 Ziele der Arbeit . . . 10

2.2 Aufbau der Arbeit . . . 12

3 Container und Cloud - Historie und Technik **13**

3.1 Historische Entwicklung von Containern . . . 13

3.2 Entwicklung des de facto Standards Docker . . . 15

3.3 Cloud Native Stack . . . 17

3.3.1 Technischer Aufbau von Containern . . . 18

3.3.1.1 Vergleich zwischen Containern und klassischer Virtualisierung . . 18

3.3.1.2 Funktionaler Aufbau . . . 18

3.3.1.3 Innerer Schichtenaufbau von Containern: Das Layer-System . . . 21

3.3.1.4 Netzwerke zwischen Containern . . . 23

3.3.1.5 Container-Cluster und Orchestrierungstools . . . 24

3.3.1.6 Fazit: Vor- und Nachteile von Containern sowie Abgrenzung zu anderen Virtualisierungstechniken . . . 25

3.3.2 Aufbau und Funktionsweise von Docker . . . 27

3.3.2.1 Information zum Docker Prozess Stack . . . 27

3.3.2.2 Aufbau von Images, der Build-Prozess und das Starten und Stoppen von Containern . . . 28

3.3.2.3 Orchestrierung mit Docker-Compose . . . 30

3.3.2.4 Volumes und das Layer-System . . . 32

3.3.2.5 Networking . . . 34

3.3.2.6 Monitoring und Logging . . . 37

3.3.2.7 Docker Engine API . . . 39

3.4 Docker als Basis der modernen Cloud . . . 39

3.4.1 Die Cloud im täglichen Leben . . . 40

3.4.2 Dienste der Cloud . . . 40

3.4.3 Docker Lifecycle . . . 42

3.4.4 Open Container Initiative . . . 42

3.4.5 Cloud Native Computing Foundation . . . 43
3.4.6 Container-Cluster-Software Kubernetes . . . 44
3.4.6.1 Funktionsweise von Kubernetes . . . 44
3.4.6.2 Kubernetes im Kontext des Cloud Native Stack . . . 46
3.4.7 Chancen und Risiken von Docker als Basis der modernen Cloud . . . 47
3.5 Energieverbrauch von Docker und des Cloud Native Stacks . . . 47
3.5.1 Energieverbrauch von Docker . . . 47
3.5.2 Auswirkungen von Docker auf Orchestrierer . . . 48
3.6 Einsatzszenarien und Praxisbeispiele . . . 48
3.6.1 Typische Einsatzszenarien für Container . . . 48
3.6.2 Praxisbeispiel I: Eine Webapp in Azure . . . 49
3.6.3 Praxisbeispiel II: Containereinsatz in der Software-Entwicklung . . . 52
3.6.3.1 Continuous Integration . . . 53
3.6.3.2 Continuous Deployment . . . 54
3.6.3.3 Folgen für den Energieverbrauch von Containersystemen . . . 54
3.7 Zwischenfazit . . . 54

4 Vorhandene Forschungsergebnisse 56
4.1 Verwandte Arbeiten im Bereich Green ICT und Energiemessung von Software . . . 56
4.2 Verwandte Arbeiten im Bereich der Effizienz von Rechenzentren . . . 56
4.3 Verwandte Arbeiten im Bereich Energiemessung von Docker und Containern . . . 57
4.4 Verwandte Arbeiten im Bereich der Container-Orchestrierung und Skalierung . . . 58
4.5 Fazit und Abgrenzung . . . 59

5 Nutzungsanalyse von Containersystemen und Docker 60
5.1 Fragenkatalog der Umfrage . . . 60
5.2 Auswertung der Ergebnisse . . . 61
5.3 Interpretation der Ergebnisse . . . 66

6 Modell und Messmethodik 69
6.1 Aufbau des Modells zur energieeffizienten Nutzung von Containern . . . 69
6.2 Messmethodik und Messumgebung . . . 71
6.2.1 Iterationen des physischen Messaufbaus . . . 71
6.2.2 Iterationen des Datenanalysevorgangs . . . 74
6.2.3 Aufbau des Messskripts . . . 75
6.2.4 Ablauf einer Messung . . . 76

7 Darstellung der Messungen **78**
7.1 Energieeffizienz von Docker . . . 78
7.1.1 Messmethodik und Messszenario . . . 79
7.1.2 Darstellung der Messergebnisse . . . 80
7.1.3 Interpretation der Ergebnisse . . . 81
7.2 Untersuchung der Kernkomponenten von Docker . . . 81
7.2.1 Docker im IDLE . . . 82
7.2.1.1 Messszenario und Messmethodik . . . 82
7.2.1.2 Darstellung der Messergebnisse . . . 82
7.2.1.3 Interpretation der Ergebnisse . . . 82
7.2.2 Container im IDLE . . . 83
7.2.2.1 Messszenarien und Messmethodik . . . 83
7.2.2.2 Darstellung der Messergebnisse . . . 83
7.2.2.3 Interpretation der Ergebnisse . . . 83
7.2.3 Docker Build-Prozesse für Images . . . 85
7.2.3.1 Messszenario und Messmethodik . . . 85
7.2.3.2 Darstellung der Messergebnisse . . . 87
7.2.3.3 Interpretation der Ergebnisse . . . 91
7.2.4 Start-, Stopp- und Löschvorgang von Containern . . . 91
7.2.4.1 Messszenario und Messmethodik . . . 91
7.2.4.2 Darstellung der Messergebnisse . . . 94
7.2.4.3 Interpretation der Ergebnisse . . . 96
7.2.5 Auswirkungen der Containergröße auf die Effizienz während der Laufzeit . . . 97
7.2.5.1 Messszenario und Messmethodik . . . 97
7.2.5.2 Darstellung der Messergebnisse . . . 99
7.2.5.3 Interpretation der Ergebnisse . . . 100
7.2.6 Auswirkungen der Imagegröße auf die Effizienz der Erzeugung eines Containers 100
7.2.6.1 Messszenario und Messmethodik . . . 100
7.2.6.2 Darstellung der Messergebnisse . . . 100
7.2.6.3 Interpretation der Ergebnisse . . . 100
7.2.7 Auswirkungen des Layersystems auf die Energieeffizienz während der Laufzeit von Containern . . . 101
7.2.7.1 Messszenario und Messmethodik . . . 101
7.2.7.2 Darstellung der Messergebnisse . . . 102
7.2.7.3 Interpretation der Ergebnisse . . . 104
7.2.8 Storage-Treiber . . . 104
7.2.8.1 Messszenario und Messmethodik . . . 104

7.2.8.2 Darstellung der Messergebnisse 105
7.2.8.3 Interpretation der Ergebnisse 105
7.2.9 Effizienz von Containern während der Laufzeit mit und ohne Volumes 106
7.2.9.1 Messszenario und Messmethodik 106
7.2.9.2 Darstellung der Messergebnisse 107
7.2.9.3 Interpretation der Ergebnisse 108
7.2.10 Netzwerk-Treiber 109
7.2.10.1 Messszenario und Messmethodik 109
7.2.10.2 Darstellung der Messergebnisse 112
7.2.10.3 Interpretation der Ergebnisse 113
7.2.11 Logging-Treiber 113
7.2.11.1 Messszenario und Messmethodik 113
7.2.11.2 Darstellung der Messergebnisse 114
7.2.11.3 Interpretation der Ergebnisse 116
7.2.12 Container-Runtimes *runC*, *kata* und *runq* 116
7.2.12.1 Messszenario und Messmethodik 116
7.2.12.2 Darstellung der Messergebnisse 117
7.2.12.3 Interpretation der Ergebnisse 117
7.3 Zusammenfassung und Fazit aller Messungen 118

8 Docker Handlungsempfehlungen **120**
8.1 Empirische Überprüfung der Handlungsempfehlungen 121
8.1.1 Erste Ersparnis-Simulation 121
8.1.1.1 Aufbau des ersten Testszenarios 121
8.1.1.2 Messaufbau zur ersten Simulation 123
8.1.1.3 Messergebnisse bezüglich Build, Container-Start, mittlerer und voller Last des ersten Testszenarios 123
8.1.1.4 Interpretation der ersten Simulation 125
8.1.2 Zweite Ersparnis-Simulation 126
8.1.2.1 Aufbau des zweiten Testszenarios 126
8.1.2.2 Messaufbau zur zweiten Simulation 127
8.1.2.3 Messergebnisse bezüglich Build, Container-Start, mittlerer und voller Last des zweiten Testszenarios 128
8.1.2.4 Interpretation der zweiten Simulation 129
8.1.3 Nutzung der Handlungsempfehlungen mit anderen Container-Systemen . . . 129
8.1.4 Transfer des Modells auf Orchestrierungstools (Modell-Kernbestandteil III) . 130
8.1.4.1 Transfer auf den Kubernetes Horizontal Pod Auto Scalers 130

8.1.4.2 Funktionaler Aufbau des Skalierungstools 131
8.1.4.3 Simulation 135
8.1.4.4 Fazit und Ausblick 137
8.1.5 Hochrechnung der Ergebnisse auf Rechenzentren 137
8.1.6 Zusammenfassung und Fazit 138

9 Fazit und Ausblick **139**

Anhang: Messreihen, Code und Weiteres **141**

Literatur **154**

Abbildungsverzeichnis

1 Verschiedene Cloud-Ansätze der Rechenzentrumsbetreiber [Gutzeit, 2020] 4
2 Google Trends bzgl. der Suchanfragen zu Docker [Google, 2019a] 5
3 Stack Overflow Insights 6
4 Vorhersagen des Energieverbrauch von Rechenzentren 8
5 GreenIT 9
6 Struktureller Aufbau eines Hypervisors Typ 1 15
7 Struktureller Aufbau eines Hypervisors Typ 2 15
8 Azure verwendet ebenfalls Docker, z.B. beim Start einer Webapp 16
9 Struktureller Aufbau von klassischer Virtualisierung 19
10 Struktureller Aufbau von Containertechnologien 20
11 PID Namespaces für Host und Container 20
12 Beispiel des Layer-Systems anhand eines Docker-Containers auf Basis eines Ubuntu-Images 22
13 Beispiel des Layer-Sharings anhand mehrerer Docker-Container auf Basis eines Ubuntu-Images 23
14 Beispiel für einen Pod als logische Einheit im Kubernetes Kontext 25
15 Beispiel für mehrere Pods im Kontext eines Kubernetes Nodes 26
16 Funktionaler Docker Prozess Stack Pre-/Post 1.11 28
17 Verbindung zwischen Containern, Docker Bridge und dem Host-System 35
18 AWS Cloud Services und Beispiele (eigene Abbildung gemäß [Amazon, 2018]) . . . 41
19 Docker Lifecycle 43
20 Orchestrierer im Cloud Native Stack 46
21 Acht Use Cases für Docker 49
22 Azure stellt eine Vielzahl an Services aus unterschiedlichen Kategorien (eigene Abbildung) 50
23 Azure erlaubt die direkte Auswahl von Docker-Container (eigene Abbildung) 51
24 Es können Container aus unterschiedlichen Registries gewählt werden (eigene Abbildung) 51
25 Auch Docker Compose Dateien können angegeben werden (eigene Abbildung) . . . 52
26 Die erstelle Webapp kann automatisch und manuell Skaliert werden (eigene Abbildung) 52
27 Gitlab CI Code Pipelines 53
28 Gitlab CI Code Pipelines Konfiguration 53
29 Verteilung der Umfrageteilnehmer auf verschiedene Berufsgruppen 62
30 Nutzung verschiedener Containersysteme 62
31 Verwendungszweck von Containern 63

32 Native oder orchestrierte Verwendung von Containern 63
33 Verwendung von Containern in der Cloud oder lokal 64
34 Typische Einsatzszenarien von Containern 64
35 Welche Logging-Treiber werden verwendet? 65
36 Welche Netzwerk-Treiber werden verwendet? 65
37 Welche Logging-Treiber werden verwendet? 66
38 Bleiben Container ein Trend? 66
39 Sinnhaftigkeit von Energiemessungen im Containerkontext 67
40 Sensibilität der Umfrageteilnehmer gegenüber einer Verschlechterung der Erreichbarkeit von Diensten mit gleichzeitigen Vorteilen beim Stromverbrauch 67
41 Messumgebung zur Bestimmung des Energieverbrauchs von Software 72
42 UML Activity Diagramm zum groben Ablauf des Messskripts (eigene Abbildung) . . 77
43 Messaufbau zur Überprüfung der Effizienz des Bridge-Treibers (eigene Abbildung) . 110
44 Lastmaxima 114
45 Leistungsaufnahme für einen skalierten bzw. unskalierten Service (normalisiert) . . . 132
46 Simultane Zugriffe auf einen skalierten bzw. unskalierten Service (normalisiert) . . . 132
47 Ermittlung des Skalierungspunktes aus den Deltas der Leistungsaufnahme und der simultanen Zugriffe (normalisiert) 133
48 Ermitteltes Skalierungsintervall in CPU-Prozent in Relation zu Leistungsaufnahme und simultanen Zugriffen (normalisiert) 133
49 Bestimmung des Skalierungspunktes 136
50 Bestimmung des Skalierungsintervalls 136

Tabellenverzeichnis

1 Wachstum der Anzahl von Berechnungseinheiten in Rechenzentren 7
2 Überblick über die Docker Storage Treiber [Liebel, 2017, Seiten 307-341] 33
3 Übersicht über die Docker Logging Treiber [Docker Inc., 2021d] 38
4 Ergebnisübersicht der Messung bezüglich des Energieverbrauchs der Sprachen C#, Go und Clojure (vgl. Kreten and Guldner [2017]) 80
5 Energieverbrauch der Sprachen C#, Go und Clojure im Testintervall (vgl. Kreten and Guldner [2017]) 81
6 Kollisionsauflösungen pro Ws der Sprachen C#, Go und Clojure im Testintervall (vgl. Kreten and Guldner [2017]) 81
7 Darstellung der durchschnittlichen Leistungsaufnahme von Testdurchläufen je 1 Minute 84
8 Leistungsaufnahme und Dauer eines default Docker Build-Prozesses ohne Cache . . 87
9 Leistungsaufnahme und Dauer eines Docker Build-Prozesses mit der Option –cache-from 88
10 Leistungsaufnahme und Dauer eines Docker Build-Prozesses mit explizitem Cache-busting 89
11 Unterschiede zwischen einem großen und einem kleineren Basis-Image ohne Cache gemäß Listing 9 und Listing 10 89
12 Unterschiede zwischen Build gemäß Listing 9 und einem Multistage-Build 90
13 Unterschiede zwischen Build gemäß Listing 9 und einem Build mit Buildkit 91
14 Erzeugung und Start eines Containers mit *docker run* 94
15 Löschen eines Containers mit *docker container rm* 94
16 Start eines zuvor erzeugten beziehungsweise gestoppten Containers mit *docker container start* 95
17 Stoppen eines laufenden Containers mit *docker container stop* 95
18 Erzeugung und Start eines Containers mit *docker-compose up -d* 95
19 Erzeugung und Start eines Containers mit *docker-compose up -d* ohne Erzeugung eines Netzwerks 96
20 Erzeugung und Start eines Go-API-Servers und einer MySQL-Datenbank mit *docker-compose up -d* 96
21 Erzeugung und Start eines Go-API-Servers und einer MySQL-Datenbank mit zwei hintereinander ausgeführten *docker run* Befehlen 96
22 Messreihe zur Bestimmung der Belastungsobergrenze des Go-Servers (Basis-Image golang:1.11-alpine3.10 (325 MB)) 98
23 Messreihe zur Bestimmung der Belastungsobergrenze des Go-Servers (Basis-Image golang (788 MB)) 98

24 Messergebnisse zum Go-Server; Belastung mit 28 Usern (Basis-Image golang:1.11-alpine3.10 (325 MB)) . . . 99
25 Messergebnisse zum Go-Server; Belastung mit 28 Usern (Basis-Image golang (788 MB)) . . . 100
26 Erzeugung und Start eines Containers mit *docker run* und Basis golang . . . 101
27 Erzeugung und Start eines Containers mit *docker run* und Basis golang:1.11-alpine3.10 . . . 101
28 Vergleich der Leistungsaufnahme (ohne Baseline) bei linear ansteigender Last und Containeranzahl . . . 102
29 Vergleich der Leistungsaufnahme (ohne Baseline) eines skalierten Webservers mit gleichmäßiger Lastverteilung . . . 103
30 Messergebnisse zum skalierten Go-Server; Belastung mit 28 Usern für 15 Sekunden (Basis-Image golang (788 MB)) . . . 103
31 Messergebnisse zum skalierten Go-Server; Belastung mit 28 Usern für 15 Sekunden (Basis-Image golang:1.11-alpine3.10 (325 MB)) . . . 103
32 Messergebnisse von zwei Go-Servern; Belastung mit 28 Usern für 15 Sekunden (Basis-Image Container 1: golang:1.11-alpine3.10 (325 MB), Basis-Image Container 2: golang (788 MB)) . . . 104
33 Zugriffe auf einen Nginx-Container mit overlay2 Storage-Treiber . . . 105
34 Zugriffe auf einen Nginx-Container mit ZFS Storage-Treiber . . . 105
35 Zugriffe auf einen Go-Server-Container mit overlay2 Storage-Treiber und dabei erzeugten Layern . . . 106
36 Zugriffe auf einen Go-Server-Container mit ZFS Storage-Treiber und dabei erzeugten Layern . . . 106
37 Messung eines Nginx-Servers mit persistent im Container abgelegter dynamischer Webseite . . . 107
38 Messung eines skalierten Nginx-Servers mit persistent im Container abgelegter dynamischer Webseite . . . 107
39 Messung eines Nginx-Servers mit im Volume abgelegter dynamischer Webseite . . . 108
40 Messung eines skalierten Nginx-Servers mit im Volume abgelegter dynamischer Webseite . . . 108
41 Messung eines Go-API-Servers mit im Container abgelegter Datenbank und ausschließlich lesenden Zugriffen . . . 108
42 Messung eines Go-API-Servers mit im Volume abgelegter Datenbank und ausschließlich lesenden Zugriffen . . . 108
43 Messung eines Go-API-Servers mit im Container abgelegter Datenbank und ausschließlich schreibenden Zugriffen . . . 109

44 Messung eines Go-API-Servers mit im Volume abgelegter Datenbank und ausschließlich schreibenden Zugriffen . . . 109
45 Ergebnisübersicht der Messung bezüglich des Energieverbrauchs der Netzwerk-Treiber Bridge und OVS Bridge . . . 112
46 Runden pro Ws der Netzwerk-Treiber Bridge und OVS Bridge . . . 112
47 Übersicht über die Effizienz der Logging-Treiber *none*, *fluentd*, *local*, *json-file*, *journald* und *syslog* . . . 115
48 Übersicht über die Effizienz des Logging-Treiber *none* in Kombination mit Logging im Volume . . . 115
49 Zugriffe auf einen Nginx-Container mit Container-Runtime *runC* . . . 117
50 Zugriffe auf einen Nginx-Container mit Container-Runtime *kata* . . . 117
51 Zugriffe auf einen Nginx-Container mit Container-Runtime *runq* . . . 117
52 Konfiguration der Container in Simulation 1 mit und ohne Handlungsempfehlungen . 122
53 Build-Prozess des ersten Testszenarios mit und ohne Handlungsempfehlungen 124
54 Start der Container des ersten Testszenarios mit und ohne Handlungsempfehlungen . 124
55 Ressourcenverbrauch der Container im ersten Testszenario bei niedriger Auslastung (fünfzigtausend Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen 124
56 Ressourcenverbrauch der Container im ersten Testszenario bei mittlerer Auslastung (dreihunderttausend Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen . . 125
57 Ressourcenverbrauch der Container im ersten Testszenario bei hoher Auslastung (zwei Millionen Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen 125
58 Energieverbrauch des ersten Testszenarios mit und ohne Handlungsempfehlungen . . 125
59 Konfiguration der Container in Simulation 2 mit und ohne Handlungsempfehlungen . 127
60 Build-Prozess des zweiten Testszenarios mit und ohne Handlungsempfehlungen . . . 128
61 Start der Container des zweiten Testszenarios mit und ohne Handlungsempfehlungen 128
62 Ressourcenverbrauch der Container im zweiten Testszenario bei mittlerer Auslastung (25.000 Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen 128
63 Ressourcenverbrauch der Container im zweiten Testszenario bei hoher Auslastung (40.000 Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen 129
64 Energieverbrauch des zweiten Testszenarios mit und ohne Handlungsempfehlungen . 129

Listings

1 Dockerfile zur Erzeugung eines Images auf Basis von scratch 29
2 Dockerfile zur Erzeugung eines Images auf Basis von scratch 29
3 Erzeugung eines Images mit Hilfe einer Dockerfile 30
4 Starten eines Containers mit Basis-Image myimage 30
5 Beispiel für eine einfache docker-compose.yaml 31
6 docker-compose.yaml für eine Container-Umgebung mit Frontend und Datenbank . . 31
7 Starten der Container-Umgebung mit Docker Compose 32
8 Auslesen der Spannung vom Messgerät über SNMP 75
9 Messzenario beschreibende Dockerfile 85
10 Messzenario beschreibende Dockerfile 86
11 Messzenario beschreibende Dockerfile 86
12 Messzenario beschreibende Dockerfile 89
13 Erzeugen und Starten eines Containers 92
14 Löschen eines gestoppten oder laufenden Containers 92
15 Starten eines Containers. 92
16 Starten eines Containers mit dem Befehl docker-compose 92
17 Aufbau der Compose-Datei 92
18 Aufbau der Compose-Datei mit externem Netzwerk 93
19 Aufbau der Compose-Datei mit Server und Datenbank 93
20 Starten eines Servers und einer Datenbank in zwei Containern 94
21 docker-compose.yaml zum Start der Netzwerk-Testumgebung 110

Abkürzungsverzeichnis

NFV - Network Function Virtualization
SDN - Software Defined Networks
LXC - Linux Containers
IAAS - Infrastructure as a Service
PAAS - Platform as a Service
FAAS - Function as a Service
CNCF - Cloud Native Computing Foundation
CI - Continuous Integration
CD - Continuous Delivery
OS - Operating System
VMM - Virtual Machine Monitor
VM - Virtuelle Maschine
KVM - Kernel Based Virtual Machine
IPC - Inter Process Communication
NAT - Network Address Translation
K8 - Kubernetes
ICT - Information and Communications Technology
OCI - Open Container Initiative
CLI - Command Line Interface
CoW - Copy-On-Write
NAT - Network Address Translation
DNS - Domain Name Service
RPC - Remote Procedure Call
SUT - System under Test
DAE - Data Aggregator and Evaluator
WG - Workload Generator
PM - Power Meter
RAPL - Running Average Power Level
JVM - Java Virtual Machine
CRI - Container Runtime Interface
OVS - Open vSwitch

1 Motivation und Einleitung

In der Mitte des ersten Jahrzehnts des 21. Jahrhunderts hat eine Veränderung der globalen Rechenzentrumsinfrastruktur begonnen, die bis zum aktuellen Zeitpunkt anhält. Dies ist neuen Herausforderungen geschuldet, denen sich Betreiber und Administratoren gleichermaßen stellen müssen. Eine dieser Herausforderungen ist der jährliche Internetverkehr, der beispielsweise alleine zwischen den Jahren 2016 und 2017 um circa 4 Zetabytes gestiegen ist [Cisco, 2018]. Grund hierfür ist die enorme Anzahl an internetfähigen Geräten , die besonders im Consumermarkt seit Beginn des Jahrtausends und speziell seit Vorstellung des ersten iPhones im Jahr 2007 mehr und mehr zunehmen. So stieg die Summe internetfähiger Geräte weltweit zwischen 2003 und 2010 von rund 500 Millionen Geräten auf über 12 Milliarden [Statista, 2011]. Da mobile Endgeräte aufgrund ihrer Größe mit Einschränkungen umgehen müssen, begann ein Wandel der Rechenzentren hin zu Serviceanbietern, auf welche eben genannte Endgeräte zugreifen können. Dabei finden Berechnungen häufig ausschließlich auf den Servern statt, während die Endgeräte nur zur Darstellung gesammelter Informationen dienen. Diese Entwicklung ist heute auch auf stationären Endgeräten in Form von Cloud-Services, Programmen mit always-online-Pflicht und Update-Abos, erkennbar. Bestes Beispiel ist die Veränderung der Microsoft-Office-Suite, von einer Software, die einmal erworben mehrere Jahre unterstützt wurde, hin zu einem Paket im Abo-Modell mit Anbindung an die Cloud zur Übertragung der verfassten Dokumente an alle Endgeräte eines Nutzers beziehungsweise einer Nutzergruppe [Microsoft, 2019]. Es ist zu beachten, dass der Begriff Cloud im folgenden allgemein für Cloud-Services und die darunterliegende Cloud-Architektur verwendet wird und nicht für einen Remote-Datenspeicher.

Diese Veränderungen an die Anforderungen von Software haben verständlicherweise einen großen Einfluss auf den Aufbau von Rechenzentren, deren neue Anforderungsgebiete sich auf die Erreichbarkeit von Services und der Verarbeitung vieler Anfragen beziehen. Um der wachsenden Anzahl an Anfragen an Rechenzentren gerecht zu werden, hat sich zum einen eine Entwicklung im Bereich der Netzwerktechnik eingestellt und zum anderen in der Auslastung von Servern. In beiden Fällen stellte sich eine verbreitete Nutzung von Virtualisierungstechniken ein, die in einer besseren Ausnutzung, Administration und Strukturierung der zur Verfügung stehenden Hardware resultierte. In diesem Kontext trifft man oft auf Begriffe wie *Software Defined Network*, *Network Function Virtualization*, *Hypervisor* und *virtuelle Maschine*, die alle einen Einfluss auf die Konsolidierung von Bare-Metal Servern und die Netzwerk-Peripherie haben. Mithilfe von Virtualisierungstechniken ließen sich somit Ressourcen bündeln und besser ausnutzen. Den weiter ansteigenden Anforderungen konnten aber auch sie nicht ganz gerecht werden. Gerade im strukturellen Bereich sowie in der Auslastung der Hardware wurde hier zunächst einiges an Potenzial verschenkt, ehe man auf neue Techniken im Bereich der Softwarebereitstellung und Entwicklung zurückgriff. So wurde im Jahr 2013 die Software Docker vorgestellt, die damals auf der im Jahr 2008 präsentierten Virtualisierungstechnik Linux Containers (LXC) basierte, welche wiederum auf Eigenschaften des Linux-Kernels zurückgriff, die

bereits seit 2002 beziehungsweise 2007 im Linux-Kernel enthalten sind. Grob versucht Docker, und im Speziellen LXC, kleinstmögliche und voneinander abgeschottete virtuelle Berechnungsinstanzen zu erzeugen, die statt „monolithischer Anwendungen aus dem klassischen Lehrbuch-Serverbereich [...] Dienste entkoppelt, und einzeln (miniaturisiert) als Microservice in je einem Container betrieben und miteinander vernetzt“[Liebel, 2017] bereitstellt. Die Vorteile, wie beispielsweise Portabilität und Sicherheit, der bis dahin typischen Virtualisierung mit Hypervisoren wie Xen, Hyper-V oder KVM, bleiben größtenteils enthalten, wobei die Ressourceneffizienz erneut durch weniger Overhead gesteigert wird. Ein Container beinhaltet nur die Bestandteile eines Betriebssystems, die zum Ausführen der Software gebraucht werden. Alle anderen Komponenten entfallen. Somit ist es möglich, mehrere tausende Berechnungseinheiten in Form von Containern auf einem Server parallel auszuführen und dadurch für eine bessere Bereitstellung verschiedenster Services zu sorgen.

1.1 Container und ihre Auswirkungen auf Rechenzentren

Mit der im letzten Abschnitt erwähnten Entwicklung in der Virtualisierung hin zu Containern geht eine zusätzliche Veränderung der Rechenzentrumsarchitektur hin zu Cloud-Rechenzentren einher. Das Angebot von Cloud-Rechenzentren weicht dabei deutlich von dem klassischer Rechenzentren ab und unterliegt meist dem prominenten Begriff der Cloud Native Architecture. Der Begriff, der 2014 von Matt Stine in seinem Buch „Migrating to Cloud-native Application Architectures“ [Stine, 2015] eingeführt wurde, beschreibt eine Architektur von modernen Anwendungen nach folgenden vier Anforderungen:

i) Modularisierung

ii) Autonomie

iii) klein

iv) API-getrieben

Modularisierung und Autonomie, sowie die Größe der einzelnen Services, werden durch den Einsatz von Containern begünstigt. Bestandteile können auf diese Weise schnell und oftmals verlustfrei ausgetauscht werden. Weiter fördert die Modularisierung eine bessere Skalierung. Die Modularisierung setzt wiederum eine Kommunikation zwischen den Modulen voraus, die oftmals über APIs gewährleistet wird.

Die Angebote von Cloud-Rechenzentren fördern und fordern diese Architektur sowie deren Anforderungen, indem sie möglichst kleine Services (zum Beispiel Datenbank, Webserver und Storage von einander getrennt) zur Verfügung stellen, deren Konfigurationen bis zu einem gewissen Grad vorgeschrieben sind. Die Services selber kommunizieren über vorgefertigte APIs. Als Grundlage der meisten Services werden dabei Container verwendet, die oftmals mit Docker betrieben werden. Insgesamt folgt daraus ein geringerer administrativer Aufwand für den Kunden, der zwar wie bereits

zuvor VMs anmietet, diese jedoch mit festgeschriebenen Services bevölkert (siehe hierzu im Detail Abschnitt 3.3.2).
Trotz dieser Veränderung, die vor allem bei großen Anbietern wie Microsoft oder Amazon sehr weit fortgeschritten ist, hat der Kunde trotzdem die Wahl zwischen fünf verschiedenen Cloud-Ansätzen. Cloud-Rechenzentren stellen beispielsweise Webspaces in Form von Infrastructure as a Service (IAAS), Platform as a Service (PAAS) oder Function as a Service (FAAS) zur Verfügung (siehe Abbildung 1). Obwohl einige große Firmen noch den Ansatz der IAAS verwenden, wenden sich andere Unternehmen aus Kostengründen PaaS und FaaS schneller zu. Da die eigene Verwaltung von VMs, Clustering und Storage entfällt, kann hier neben Geld auch Zeit eingespart werden. So dauert das Aufsetzen eines skalierenden Webservices mit SSL-Zertifikat in der Cloud keine Minute, während man sich auf dem eigenen Server um die Installation und Absicherung des Webservers selbst kümmern muss. Im Kontext der Verwendung von Cloud-Services wird auch häufig von DevOps gesprochen. Mitarbeiter in dieser Sparte kümmern sich um die Verbindung zwischen Entwicklern und Cloud-Rechenzentren und übernehmen die letzten administrativen Aufgaben, die nicht von Cloud-Rechenzentren übernommen werden. Beispielsweise zählen hierzu im Äußersten die Verwaltung virtueller Netze, Domänen und Storage sowie die Einrichtung von Code-Pipelines. In den meisten Fällen kann hier auf bereits vorkonfigurierte Elemente zurückgegriffen werden, die erst bei umfangreicherer Nutzung einer erweiterten Konfiguration bedürfen. Container selbst, als Basis einer jeden Cloud-nativen Anwendung, fördern darüber hinaus die Entwicklungs-, Bereitstellungs-, und Testgeschwindigkeit neuer Applikationen, indem sie verstärkt auf Pipelines, Continuous Integration (CI) und Continuous Delivery (CD) sowie erhöhte Skalierbarkeit setzen. So kann beispielsweise jeder Commit in der Softwareversionisierung zum Neustart und dem damit verbundenen, automatisierten Ausrollen der Software führen. Aus diesem Grund spricht man auch von sogenannten Rolling-Releases.

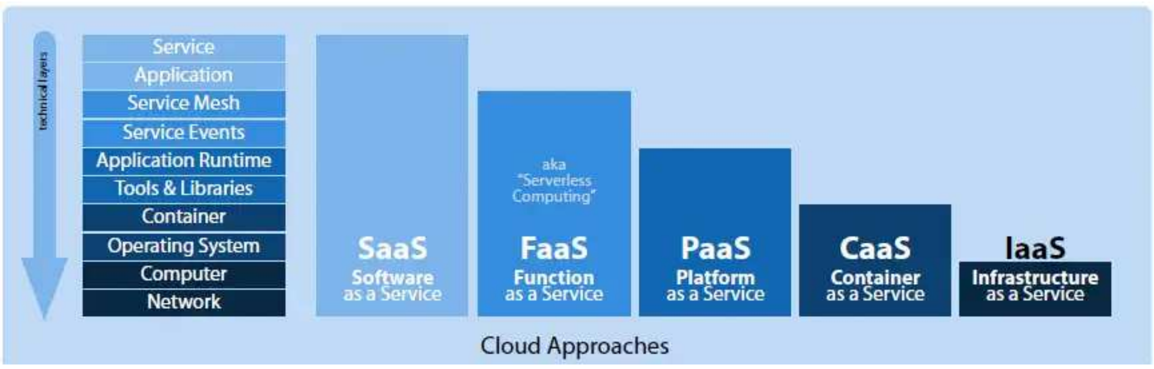

Abbildung 1: Verschiedene Cloud-Ansätze der Rechenzentrumsbetreiber [Gutzeit, 2020]

Die hier beschriebenen Cloud-Rechenzentren beziehen sich in erster Linie auf die sogenannte Public-Cloud. Also eine Cloud, die von jedem zahlenden Kunden verwendet werden kann. Große Firmen betreiben zumeist eigene Cloud-Rechenzentren, die nur ihnen und ihren Partnern zur Verfügung stehen. Einer der größte Anbieter privater Cloud-Ressourcen in Deutschland ist BMW, die mit Hilfe der

Software OpenStack 40000 virtuelle CPUs und über 400.000 VMs verwalten [Pöschl, 2019]. Auch eine Verbindung von privaten und public Cloud-Ressourcen ist möglich. In diesem Fall spricht man von der Hybrid-Cloud.
Resultierend aus der Verlagerung der Server in Cloud-Rechenzentren, und der steigenden Anzahl an Services, wuchs die Anzahl an Cloud-Rechenzentren zwischen 2016 und 2018 um 11% von 338 auf 448 an. Zusätzlich dazu muss erwähnt werden, dass Cloud-Rechenzentren aktuell 95% des gesamten Traffics auf Rechenzentren verarbeiten [Cisco, 2018]. Ein weiterer Fakt, der das Wachstum der Cloud und die einhergehende Notwendigkeit von Cloud-Rechenzentren unterstreicht, ist die Gründung der Cloud Native Computing Foundation (CNCF) im Jahr 2015. Als Unterprojekt der Linux Foundation vereinigt sie zum jetzigen Zeitpunkt mehr als 904 Unternehmen und Softwarelösungen im Cloudbereich mit einer Marktkapitalisierung von 15 Billionen US-Dollar und einer Finanzierungssumme von über 15 Milliarden US-Dollar [CNCF, 2019] (Stand 03/2021).

1.2 Chancen und Risiken

Wie bereits zuvor erwähnt, sorgen Cloud-Rechenzentren für einen geringeren administrativen Aufwand in Unternehmen und stellen dem Nutzer durch permanente Verbindung zu einer Vielzahl an Services eine bessere Nutzererfahrung zur Verfügung. Darüber hinaus steigt durch die verstärkte Nutzung von Containern, die auch in den Suchtrends von Google und Stack Overflow deutlich erkennbar ist (siehe Abbildung 2 und Abbildung 3), und der Verwendung der Cloud Native Architecture, die Entwicklungsgeschwindigkeit von Applikationen, respektive ihrer Updatezyklen, stark an. Ressourcen können durch verstärkte Modularisierung und der daraus resultierenden besseren Skalierung sowie Lastbalancierung effektiver ausgeschöpft werden und erlauben so in Kombination mit horizontaler und vertikaler Skalierung eine effektive, nachfragengesteuerte Bereitstellung von Online-Diensten. Doch auch die einfache Bereitstellung von Services „out of the box“erleichtert nicht nur Administratoren, sondern ebenfalls ungeschultem Personal die tägliche Arbeit.

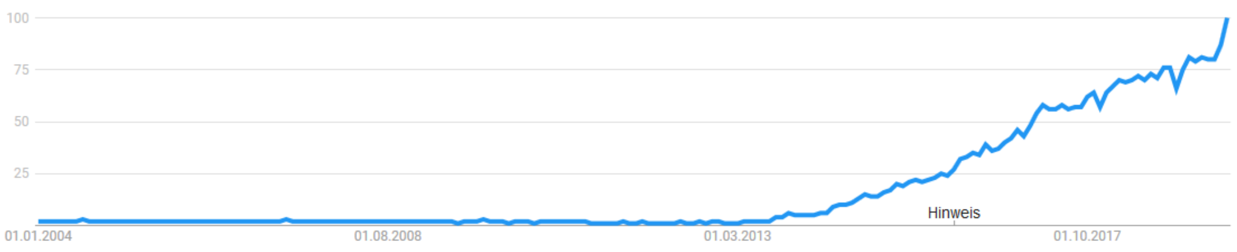

Abbildung 2: Google Trends bzgl. der Suchanfragen zu Docker [Google, 2019a]

Obwohl durch den Einsatz von Cloud Techniken und im Besonderen Containern Ressourcen gespart

und Server konsolidiert werden können, steigt die Anzahl an Berechnungseinheiten in den weltweiten Rechenzentren aufgrund der stärkeren Modularisierung kontinuierlich an (siehe Tabelle 1).

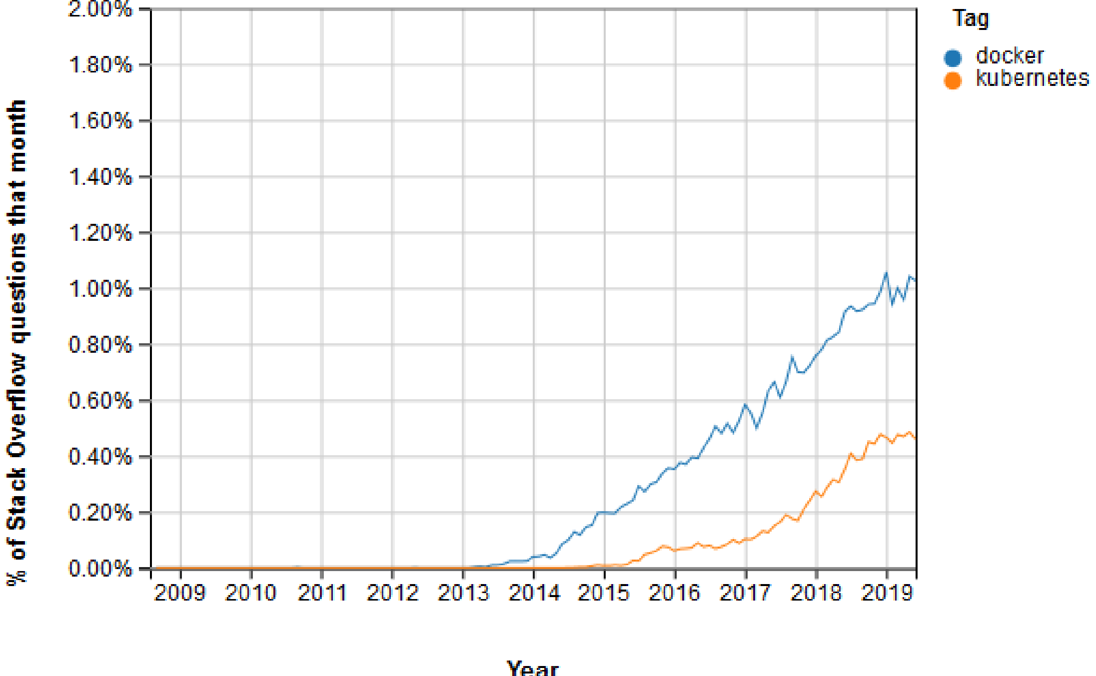

Abbildung 3: Stack Overflow Insight bzgl. der Anfragen pro Monat zum Thema Docker und Kubernetes [Stack Overflow, 2019]

Hierzu ein belegendes Zitat aus dem Cisco Cloud Global Index:

> „In December 2015, [Netflix] launched a few thousand containers per week across a handful of workloads and compute instances. By April 2017, it had launched more than one million containers. These containers represent hundreds of workloads and compute instances. This thousandfold increase in container usage happened over a year time frame, and growth doesn't look to be slowing down" [Cisco, 2018].

Netflix ist hierbei auch ein gutes Beispiel, da der jährlich ansteigende Datenverkehr hauptsächlich durch den Konsum von Online-Videos und Streaming induziert wird. Circa 80% des globalen Datenflusses gingen 2018 auf die Übertragung von Videos zurück [The Shift Project, 2019]. Um der steigenden Nachfrage gerecht zu werden, setzt Netflix, wie im Zitat gezeigt, verstärkt auf den Einsatz von Containern.

Entsprechend der steigenden Anzahl an Berechnungseinheiten in Form von Containern, bedingt durch die immer weiter ansteigende Nachfrage nach Cloud-Services und deren Erreichbarkeit, steigt der jährliche Stromverbrauch von Rechenzentren also trotz der vermeintlich energieeffizienten Cloud

	2016	2017	2018	2019	2020	2021
Traditional data center workloads and compute instances	42.1	41.4	40.8	39.1	36.2	32.9
Cloud data center workloads and compute instances	199.4	262.4	331.0	393.3	459.2	553.7
Total data center workloads and compute instances	241.5	303.8	371.8	432.4	495.4	566.7
Cloud workloads and compute instances as a percentage of total data center workloads and compute instances	83%	86%	89%	91%	93%	94%
Traditional workloads and compute instances as a percentage of total data center workloads and compute instances	17%	14%	11%	9%	7%	6%

Tabelle 1: Wachstum der Anzahl von Berechnungseinheiten in Rechenzentren (gemäß [Cisco, 2018])

immer weiter an. Voneinander unabhängige Vorhersagemodelle zeigen dabei ein zukünftiges Problem, das aus der Umstellung auf Cloud-Rechenzentren sowie der Nutzung von Container resultiert (siehe Abbildung 4), obwohl Cloud-Rechenzentrumstechnologien im Allgemeinen energieeffizienter als ältere Rechenzentrumstechnologien sind. So schrieb Jeff Barr, einer der Hauptverantwortlichen bei AWS, dass durch einen Wechsel in die Cloud eine Strombedarfssenkung von 84% möglich sei [Amazon, 2021]. Es ist jedoch hier, wie vorhin erwähnt, zu beachten, dass es immer mehr Cloud-Rechenzentren gibt und somit insgesamt der Stromverbrauch wächst. Container sind hierbei aber nicht die Ursache des Problems, da sie ebenfalls zu einer besseren Auslastung und zur Konsolidierung von Servern beitragen. Die steigende Anzahl an Berechnungseinheiten ist einzig der hohen Nachfrage geschuldet. Auch wenn die Modelle aus Abbildung 4 alle unterschiedliche Szenarien prognostizieren, bleibt zu erkennen, dass in allen Modellen mit einem deutlichen Anstieg des durch Rechenzentren induzierten Stromverbrauchs gerechnet wird. Im Modell von Andrae von 2019 (Abbildung 4, schwarz) sieht man, dass im Jahr 2010 183 TWh an Energie in den Rechenzentren der Welt verbraucht wurden, während es im Jahr 2018 bereits 211 TWh an Energie waren. Dies stimmt mit circa 1% des globalen Energieverbrauchs überein. Gemäß dieser Daten, wird sich der Energieverbrauchswert im schlechtesten Fall bis zum Jahr 2030 auf fast 2000 TWh entwickeln, was zu diesem Zeitpunkt bereits mehr als 6% des globalen Energieverbrauchs entsprechen würde [Andrae, 2017] [Andrae, 2019]. Doch auch ohne Prognosen lässt sich klar sagen, dass der Energieverbrauch von Rechenzentren wachsen wird, wenn wir auf kommende Techniken wie autonomes Fahren, zusätzliche Streaming-Angebote (ebenso im Gaming-Bereich (siehe Google Stadia)) sowie die wachsende Anzahl an IoT-Geräten schauen.
Da nicht davon auszugehen ist, dass wir als Konsumenten auf die absolute Verfügbarkeit wichtiger Services und damit beispielsweise auch unsere Smartphones verzichten wollen und sich Entwickler

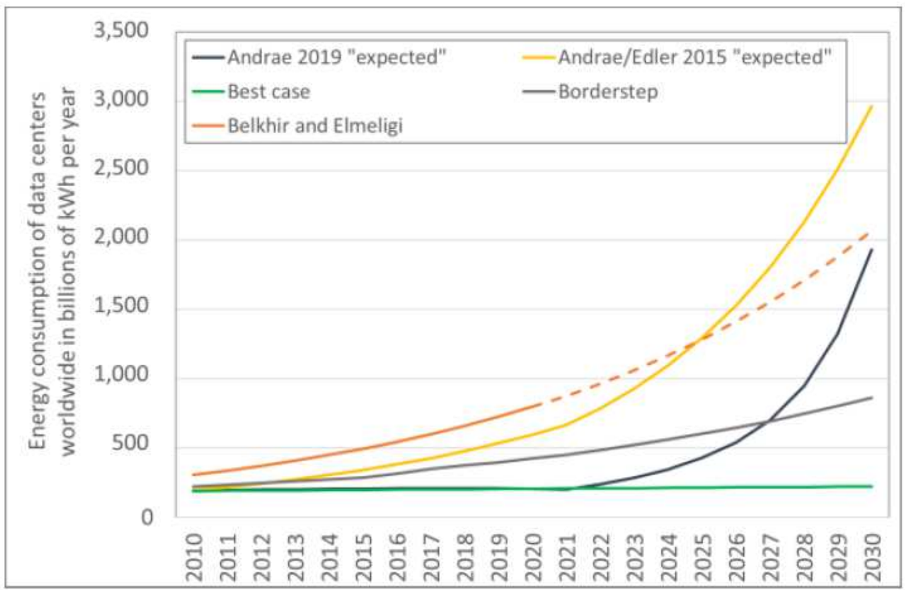

Abbildung 4: Vorhersagen des Energieverbrauch von Rechenzentren[Hintemann and Hinterholzer, 2019]

nicht der neu erlernten Techniken und der vereinfachten Administration entledigen wollen, tut sich hiermit also eine Diskussion bezüglich der Energieeffizienz von Cloud Rechenzentren auf, die sich damit auch in die aktuelle Klimadiskussion eingliedert. Bricht man die gesammelten Informationen auf die Basis des gesteigerten Energieverbrauchs herunter, steht hier definitiv der durch Software und Hardware induzierte Ressourcenverbrauch im Mittelpunkt. Betrachtet man dabei nur die Softwareseite, muss man, wie zuvor angedeutet, auf die hohe Anzahl an Berechnungseinheiten, respektive Container, schauen, in denen, wie im Beispiel von Netflix, die Software ausgeliefert und zur Verfügung gestellt wird. Aus diesem Grund beschäftigt sich diese Arbeit mit dem Ressourcen- und ganz speziell mit dem Energieverbrauch von Containersystemen.

1.3 Energiemessungen von Software

Passend zum zuvor beschriebenen Sachverhalt konnte der Bereich Green IT in den letzten Jahren zunehmend an Bedeutung gewinnen. Zu erkennen ist dies beispielsweise bei der Betrachtung der Google Trends zwischen 2004 und heute bei Angabe von Green IT (vgl Abbildung 5). Zu einem der Unterbereiche von Green IT zählen eben auch Energiemessungen im Bereich Software, die ebenfalls in den letzten Jahren an Popularität gewinnen konnten und immer wieder Ziel der Forschung sind. Ein Beispiel hierfür sind die zuletzt erschienenen Arbeiten [Mancebo et al., 2019] und [Kern et al., 2018], die sich mit neuartigen Messmethoden und Kriterien für energieeffiziente Software auseinandersetzen. Die resultierten Kriterien sind mitunter Grund für die Einführung der freiwilligen Zertifizierung des Blauen Engels für ressourcen- und energieeffiziente Softwareprodukte Ende 2019 (vergleiche [Bundesministerium für Umwelt, 2019]). Neben der Energieeffizienz von Software, befasst sich Green IT mit der Lebensdauer von Hardware sowie der notwendigen Wartung von Software.

Abbildung 5: Google Trends zwischen 2004 und heute für den Suchbegriff Green IT [Google, 2019b]

Mittlerweile gibt es bereits Ergebnisse zu häufig benutzter Software, die sich meist auf Desktop- und Serveranwendungen beziehen. So zeigen beispielsweise Suárez-Albela et al. in [Suárez-Albela et al., 2017] die unterschiedliche Leistungsaufnahme von Webservern, während Guldner et al. in [Guldner et al., 2018] auch die Unterschiede zwischen bekannter Desktopsoftware wie exemplarisch unterschiedlichen Office Suiten zeigen. Ein auf Containersysteme angepasstes Modell ist aktuell nicht zu finden, obwohl Container als Basis einer millionenfachen Anzahl von Berechnungseinheiten große Potentiale aufweisen könnten. Denn bereits eine kleine energetische Veränderung könnte große Auswirkungen auf den Energieverbrauch von Rechenzentren haben.

2 Ziele und Aufbau der Arbeit

Im Folgenden werden die Ziele und der Aufbau der Arbeit dargestellt.

2.1 Ziele der Arbeit

Um die zuvor erwähnte Anpassung von Containersystemen mit dem Zweck einer Reduktion des durch Container induzierten Stromverbrauchs zu erreichen, werden drei Kernziele definiert:

i) Ermittlung von Nutzungsweisen und Szenarien von Containern über eine Umfrage im Expertenumfeld und eine Evaluation geeigneter Literatur

ii) Konzeption eines Modells für eine energieeffiziente Nutzung von Containersystemen am Beispiel vom de facto Standard Docker bestehend aus drei Modell-Kernbestandteilen

iii) Bereitstellung von Tools zur weiteren Verbesserung der Energieeffizienz von Containersystemen

Mithilfe einer Umfrage und der Evaluation geeigneter Literatur sollen mögliche Probleme bei der Nutzung von Containern aufgedeckt werden. Dabei wird vor allem darauf geachtet, ob Nutzer ihre Container selbst verwalten oder mit den standardmäßigen Einstellungen verwenden. Weiter soll die Sensibilität von Administratoren und Entwicklern bezüglich des Energieverbrauchs von Containersoftware bestimmt werden. Aus den Ergebnissen der Umfrage und der Evaluation resultierend sollen Standardszenarien im Containerumfeld genutzt werden, um die Bausteine der Software Docker (siehe Abschnitt 3.3.2) zu untersuchen. Als Ziel sollen Handlungsempfehlungen aus diesen Untersuchungen abgeleitet werden, die sowohl Entwicklern als auch Administratoren die Möglichkeit geben, im Sinne einer energieeffizienten Nutzung von Docker und in Abhängigkeit vom Nutzungsszenario der Container zu entscheiden, welche Bestandteile des Docker-Systems angewandt werden sollten und welche entfallen könnten. Die daraus resultierenden Container sollen bezüglich ihrer Energieeffizienz sowohl auf Servern und gleichermaßen auf PCs und Embedded Systems (als Teil von IoT und Edge Cloud) anwendbar sein und damit nicht nur dem zuvor geschilderten Problem in der Cloud entgegenwirken. Es ist dabei zu beachten, dass nicht die Software überprüft wird, die in einem Container läuft. Es soll daher auch gezeigt werden, dass diese gesondert untersucht werden muss.
Das dritte Hauptziel befasst sich vorwiegend mit dem Verhalten von skalierten Web-Applikationen. Gängige Orchestrierungstools legen statische Skalierungspunkte für Applikationen fest, die in den meisten Fällen auf der CPU-Auslastung aufbauen. Dadurch wird weder auf die tatsächliche Erreichbarkeit noch auf den Stromverbrauch der Anwendungen geachtet. Nachfolgend wird der Autoscaler des Open Source Container Orchestrierungstools Kubernetes durch ein neu entwickeltes Tool erweitert. Mit ihm soll gezeigt werden, dass mit Hilfe einer Vorevaluation gängiger Nutzungsszenarien sowie Informationen zu deren Stromverbrauch und der Erreichbarkeit pro wachsender Auslastung,

eine dynamische Anpassung der Skalierungspunkte erreicht werden kann. Resultierend aus dieser dynamischen Anpassung soll wiederum eine deutlich erkennbare Energieersparnis folgen. Dabei soll die Bestimmung der Skalierungspunkte derart gestaltet sein, dass sie wiederum möglichst energiesparend vollzogen werden kann.

2.2 Aufbau der Arbeit

Nachfolgenden wird der Aufbau dieser Ausarbeitung beschrieben, wobei Kernbestandteile der Arbeit kurz angemerkt werden. Kapitel 3 wird sich mit einer Einführung in die Containerwelt befassen. Hierbei wird ein historischer Überblick über die Entwicklung von Containertechnologien gegeben und deren Zusammenhang mit Virtualisierung beschrieben. Darüber hinaus wird dargestellt, welchen Stellenwert Container in der Cloud-Landschaft und im sogenannten Cloud Native Stack haben und wie sie diese auch jetzt noch beeinflussen. Zudem wird der strukturelle und der funktionale Aufbau von Containern erläutert, damit diese bezüglich Vor- und Nachteilen mit klassischer Virtualisierung verglichen werden können. In Abschnitt 3.3.2 wird Docker als de facto Standard für Containersoftware bezüglich des Aufbaus und der Funktionsweise vorgestellt. Im Nachgang wird Docker in Unterkapitel 3.4 als Basis der modernen Cloud eingeordnet. Betrachtet wird dabei das Angebot der Cloud sowie unsere Verbindung mit ihr im täglichen Leben. Zusätzlich dazu wird ein Blick auf die Bemühungen um Standardisierung im Container-Kontext geworfen. Da Container-Cluster im späteren Verlauf der Ausarbeitung von zentralem Interesse sein werden, bezieht sich Abschnitt 3.4.6 auf Funktionsweisen der Cluster-Software Kubernetes. Anschließend wird ein Überblick über aktuelle Aussagen bezüglich des Energieverbrauchs von Docker und des Cloud Native Stacks gegeben, bevor Kapitel 3 mit der Darstellung typischer Einsatzszenarien und Praxisbeispiele schließt.

In Kapitel 4 werden nachfolgend thematisch verwandte Arbeiten vorgestellt. Hierbei wird unterschieden zwischen Veröffentlichungen aus dem Bereich Green ICT sowie Energiemessungen von Software und Artikeln zu den Themen Effizienz von Rechenzentren, Docker und Containern sowie Container-Orchestrierung und Skalierung.

Um die praktische Arbeitsweise mit Docker zu evaluieren, widmet sich Kapitel 5 ausschließlich einer Umfrage zum Nutzungsverhalten von Docker. Dabei stehen zuerst der Fragenkatalog und das Ziel der Erhebung im Fokus, bevor die Ergebnisse dargestellt und interpretiert werden.

Unterkapitel 6.1 stellt die Bestandteile eines Modells zur energieeffizienten Nutzung von Containern vor, bevor in Unterkapitel 6.2 die Messmethodik und in Kapitel 7 die dazugehörigen Energiemessungen präsentiert werden. Die Ergebnisse werden in Kapitel 8 zum Aufbau von Handlungsempfehlungen für einen energieeffizienten Umgang genutzt, die anschließend in einer Simulation Anwendung finden und einer Erfolgsüberprüfung unterzogen werden. Abschnitt 8.1.4 schließt dann die Bildung des Modells mit der Vorstellung eines Tools zum energieeffizienten Autoscaling in Kubernetes ab, wobei hier erneut eine Simulation zur Erfolgsüberprüfung eingesetzt wird.

Abschließend mündet die Ausarbeitung in einem Fazit und einem Ausblick auf die Zukunft der Containertechnologie sowie der Weiterentwicklung des Modells, bevor im Anhang der zur Messung verwendete, eigens verfasste Code und ein Auszug der Messreihen ebenso wie die verwendete Literatur dargestellt werden.

3 Container und Cloud - Historie und Technik

Um die Auswirkungen von Containertechnologien auf die Cloud deutlich darstellen zu können, bedarf es der Betrachtung der Historie von Virtualisierungstechniken und Containern. Durch die Darstellung des klar erkennbaren roten Fadens in der Weiterentwicklung der Virtualisierung wird deutlich, warum Containertechnologien der logische nächste Schritt waren und welche Vor- und Nachteile Container bieten. Aus diesem Grund wird im folgenden Kapitel zunächst die Historie der Container sowie die Entwicklung von Docker betrachtet, bevor ein genauer Blick auf die heutige Cloud-Landschaft und den Einfluss von Containern auf dieses Ökosystem geworfen wird. Weiter sollen sowohl der strukturelle als auch der funktionale Aufbau von Containern und im Speziellen Docker betrachtet werden.

3.1 Historische Entwicklung von Containern

Die Grundlagen für die Betriebssystem-Virtualisierung, wie sie im Allgemeinen heute in Rechenzentren Verwendung findet, wurden bereits im Jahr 1974 durch Gerald J. Popek und Robert P. Goldberg in ihrem ACM-Artikel „Formal Requirements for Virtualizable Third Generation Architectures“ gelegt. Im Artikel gehen sie im Besonderen auf architektonische Vorgaben zur Unterstützung eines Hypervisors oder, wie im Text genannt, Virtual Machine Monitors (VMM), ein [Popek and Goldberg, 1974]. Dabei wird hier unter anderem auch auf die Virtualisierung und die Emulation der dem Betriebssystem zugrunde liegenden Hardware sowie der Verteilung der Hardware-Ressource eingegangen. Kernaussage bleiben die folgenden drei Anforderungen an die Host-Systeme:

i) Äquivalenz - Programme zeigen in einer virtualisierten Umgebung ein identisches Verhalten zur äquivalenten realen Maschine

ii) Ressourcenkontrolle - Der Hypervisor muss vollständige Kontrolle über die zur Verfügung stehenden Ressourcen besitzen

iii) Effizienz - Der Hypervisor hat keinen Zugriff auf eine statische Menge von relevanten Prozessorinstruktionen

Bereits etwas früher, ab dem Jahr 1967, gab es erste Versuche durch IBM, Virtualisierung möglich zu machen. Dabei wurden jedoch erst Teile der Hardware, wie beispielsweise die Speicherverwaltung, virtualisiert. Mit der heutigen Betriebssystem-Virtualisierung können diese Versuche daher nur schwer verglichen werden.
Es muss hierbei auch beachtet werden, dass ein Ansatz zur Virtualisierung nicht nur Software-seitige, sondern in erster Linie Hardware-seitige Unterstützung erfordert. So haben heutzutage alle großen Hersteller ihre eigenen Techniken, um Virtualisierung auf ihrer Hardware möglich zu machen. Im Falle von IBM führte man bereits in den 1980er Jahren die Technologie des *Logic Partioning* ein, die

Virtualisierung gewährleisten sollte. Bei HP hieß eine ähnliche Technik *Integrity Virtual Machines*. Diese sollen an dieser Stelle nicht weiter beschrieben werden, jedoch sollte bemerkt werden, dass alle großen Hersteller an eigenen Lösungen zur Virtualisierung arbeiteten. Nach Einführung der ersten Virtualisierungslösungen in den 80er Jahren wurde die Entwicklung der Virtualisierungstechnik häufig bei bedeutenden Unix und Serverherstellern wie IBM, Sun Microsystems oder HP vorangetrieben, die bereits Ende der 90er Jahre Serverlösungen auslieferten, welche Virtualisierung unterstützten. Ab circa 2005 entfachte das Interesse an Virtualisierung erneut, da leistungsfähigere Hardware immer schwieriger werdende Serveradministration erforderte (komplexere Systeme, die zuverlässig und sicher sein mussten) und Multiprozessor-Systeme, ebenso wie Servercluster, nach neuen Lösungsmöglichkeiten verlangten. Eine der damals entwickelten und heute noch Anwendung findenden Techniken ist die Prozessorerweiterung im Rahmen der x86-Hypervisoren, die im Zuge einer kollaborativen Forschung verschiedener CPU-Hersteller entwickelt wurde. Bekannte Virtualisierungssoftware wie Xen, die Software von VMWare, KVM oder Hyper-V von Microsoft nutzen diese Technik aus. Dabei werden x86 Prozessoren, wie exemplarisch die Intel VT-x Prozessoren, mit einem zusätzlichen Befehlssatz ausgestattet, auf den die Virtualisierungssoftware, respektive ein Hypervisor, zurückgreifen kann.
Die nächste Veränderung im Virtualisierungsmarkt war die Einführung der Container-Technolgie, die ihren Ursprung, wie schon die Hypervisoren, zu einem deutlich früheren Zeitpunkt fanden. Erste Versuche mit der Isolation einzelner Software wurden bereits 1979 von Unix-Entwicklern durchgeführt. Diese Entwicklung zog sich allerdings bis in die späten 90er Jahre durch, ohne dabei größerer Popularität zu erlangen. Erst Mitte der 2000er und spätestens mit der Einführung von Containerfeatures in den Linuxkernel im Jahr 2008 gewannen Containertechnologien ein deutlich höheres Ansehen, was in der Vorstellung der Containersoftware Docker im Jahr 2013 seinen Höhepunkt fand (siehe hierzu Unterkapitel 3.2).
Abschließend muss im zeitlichen Verlauf noch die Abgrenzung zwischen zwei Typen von Hypervisoren gemacht werden. Bereits ein Jahr vor Veröffentlichung des oben genannten Artikels von Popek und Goldberg stellte letzterer in seiner Promotion zwei unterschiedliche Formen von Hypervisoren dar [Goldberg, 1973]. Der sogenannte Typ 1 Hypervisor soll dabei direkt auf der Hardware aufsitzen und keine Installation eines Host-Systems benötigen (Abbildung 6). Voraussetzung hierfür ist allerdings die Bereitstellung entsprechender Treiber durch die Hardwarehersteller. Beispiele heutiger Typ 1 Hypervisoren sind Xen oder Hyper-V.

Beim Typ 2 Hypervisor muss im Vorhinein ein zugrunde liegendes Host-Betriebssystem installiert werden (Abbildung 7). Dabei werden keine expliziten Gerätetreiber für den Hypervisor benötigt, denn die Treiber des Hosts können hierbei ausgenutzt werden. Lediglich das Host-System muss den Hypervisor unterstützen. Beispiele für einen solchen Hypervisor sind Oracle VirtualBox und VMware Workstation.

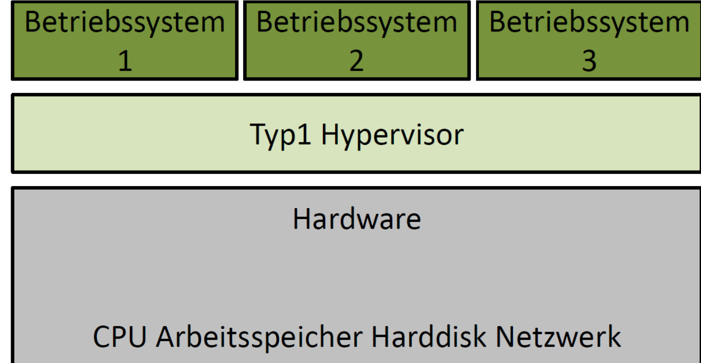

Abbildung 6: Struktureller Aufbau eine Hypervisors Typ 1 (eigene Abbildung)

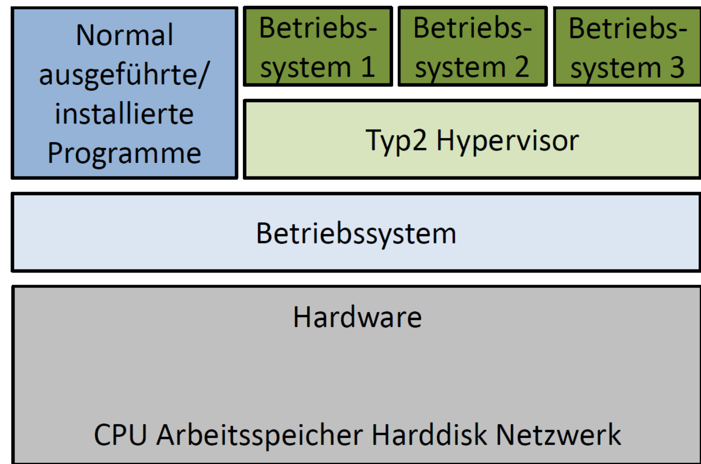

Abbildung 7: Struktureller Aufbau eine Hypervisors Typ 2 (eigene Abbildung)

3.2 Entwicklung des de facto Standards Docker

Wie bereits in Kapitel 1 und Unterkapitel 3.1 erwähnt, wurde Docker im Jahr 2013 auf der Python Konferenz PyCon in Santa Clara, Kalifornien vorgestellt. Seitdem hat sich Docker zum Marktführer in Sachen Containerisierung entwickelt, der bis Ende 2019 über 750 Docker Enterprise Customer bediente. Docker stellt mehr als 8 Millionen Container-Repositories auf seiner Plattform Docker Hub zur Verfügung, die circa 300 Milliarden Downloads zählen und von mehr als 7 Millionen registrierten Nutzern verwaltet werden. Dazu kommen mehr als 200 offizielle Veranstaltungen im Docker Kontext sowie über 100.000 3rd Party Projekte, die Docker nutzen [Docker Inc., 2021a]. In den alljährlich erhobenen Stackoverflow Insights von 2020 zeigt sich Docker zudem als Nummer 2 der „Most Loved, Dreaded, and Wanted Platforms“ hinter Linux sowie als Nummer 3 der Top Plattformen hinter Windows und Linux [Stack Overflow, 2020]. Grund für die Beliebtheit von Docker ist die Simplizität, mit der Container durch den Nutzer erzeugt werden können. Neben einer einfach verständlichen CLI bietet Docker zudem eine Art Config-File, die sogenannte Dockerfile, in welcher Container spezifiziert werden können. Darüber hinaus brachte Docker zudem eingebaute Überwachung und Steuerung sowie ein Storage-Management mit sich und prägte schnell die Begriffe „Everthing as Code“ sowie

„Compile once, run everywhere“. Dabei kann jede Art von Software in einem Container laufen, wobei dies für Desktop-Anwendungen umständlich und eher untypisch ist.
Zu Beginn seiner Entwicklung hieß die Firma, die heute als Docker Inc. bekannt ist, noch dotCloud. Wie am ehemaligen Namen bereits erkennbar ist, liegt der Fokus bei Docker klar auf der Cloud. Noch im selben Jahr benannte sich die Firma allerdings um. Bereits im Jahr 2014 wurden große Unternehmen wie Red Hat und Microsoft auf Docker aufmerksam, was letztlich in einem Zusammenschluss im von Google initiierten Kubernetes-Projekt zur Orchestrierung von Containern führte (auf Kubernetes wird gesondert in Abschnitt 8.1.4 eingegangen). Durch diese steigende Beliebtheit sammelte Docker bis zum Ende des Jahres 2015 insgesamt 180 Millionen US-Dollar an Finanzierungshilfe. Bis zum heutigen Zeitpunkt sind es bereits 272 Millionen US-Dollar in 8 Finanzierungsrunden [Crunchbase, 2021]. Als weiteres Resultat aus der Kooperation mit großen Unternehmen bildet Docker heute sichtbar die Basis der Cloud-Stacks großer Anbieter wie Microsoft und Google (vergleiche Abbildung 8).

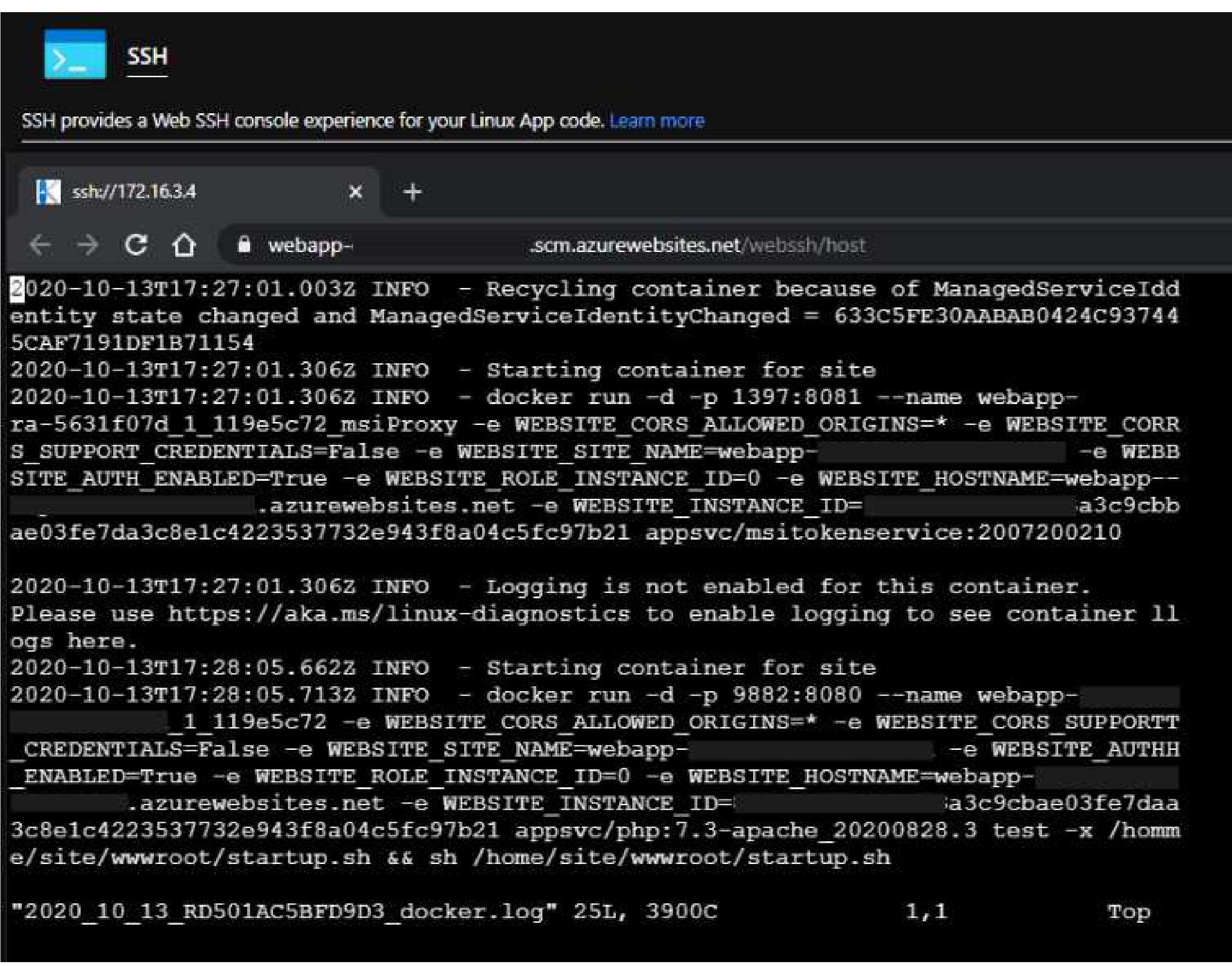

Abbildung 8: Azure verwendet ebenfalls Docker, z.B. beim Start einer Webapp (eigene Abbildung)

Docker selbst gründete im Jahr 2015 die OCI (Open Container Initiative), die sich mit dem Aufbau von Industrie-Standards im Bereich der Container-Runtimes sowie der Image-Spezifikation beschäftigt. So

nutzte Docker zu Beginn LXC (Linux Containers) als Grundlage, welches später durch die Eigenentwicklungen *runC* [Open Container Initiative, 2016] und *containerd* [The Linux Foundation, 2019a] ersetzt wurden. runC wird heute von der OCI verwaltet und entspricht der von der Initiative aufgesetzten Spezifikation von Container-Runtimes. containerd ist ebenfalls eine Container Laufzeitumgebung, die sich jedoch nicht wie runC um die Bereitstellung der Container nach OCI-Spezifikation kümmert, sondern um das Management der Container bezüglich Image, Storage und Networking sowie der Überwachung und Kommunikation. Der Prozess abstrahiert die durch runC gebauten Container-Pakete und erlaubt die tatsächliche Interaktion mit Containern. containerd gilt als Herzstück von Docker und wurde daher von der Docker Inc. im Jahr 2017 an die Cloud Native Computing Foundation (CNCF) gespendet. Auf CNCF und OCI wird in Unterkapitel 3.4 genauer eingegangen. Seit 2017 gab es keine größeren Änderungen im Docker-Kosmos. Zwar erscheinen in regelmäßigen Abständen neue Community Versionen (aktuell 20.10) sowie Enterprise Editions (aktuell Version 3), große Meilensteine lassen jedoch auf sich warten. Tatsächlich hat Docker im November 2019 seine Enterprise-Sparte an das Unternehmen Mirantis verkauft und Konzentriert sich fortan verstärkt auf eine Verbesserung des Container Build Prozesses. Insgesamt kommen aus dem Docker Umfeld, vor allem aus dem Bereich der Container Orchestrierung und dabei speziell Kubernetes, aktuell größere Entwicklungssprünge. containerd und runC werden gesondert als Open Source Projekte im Rahmen der Vorgaben der OCI weiterentwickelt.

3.3 Cloud Native Stack

Der Cloud Native Stack ist eine Erweiterung des in Abbildung 7 dargestellten Hypervisor Typ 2 Modells. Aufgesetzt auf den durch Cloud-Anbieter administrierten VMs befindet sich ein weiterer Stack an Container-Tools, bestehend aus Container-Runtime, -Orchestrierung, -Monitoring, -Networking und -Storage. Wie dieser Aufbau im Detail aussieht, wird erst klar, wenn man die Unterschiede zwischen klassischer Virtualisierung und Containern betrachtet. Die zuvor erwähnten Tools, wie beispielsweise die Container-Runtime, können dabei frei aus einer Vielzahl von unterschiedlichen Angeboten gewählt werden, funktionieren im Kern jedoch zumeist gleich.
Um zu verstehen, wie der Cloud Native Stack mit seinen Komponenten oberhalb der virtuellen Maschinen funktioniert und wie, beziehungsweise wo, sich Docker innerhalb dieses Ökosystems einordnet, ist eine Betrachtung des allgemeinen technischen Aufbaus von Containern und im Speziellen Docker unumgänglich. Weiter ist ein Überblick über die Standardisierung von Containern und typische Einsatzszenarien für Administratoren und Entwickler förderlich, um den Cloud Native Stack und Wichtigkeit von Containern in diesem Konstrukt zu verstehen.

3.3.1 Technischer Aufbau von Containern

In den nachfolgenden Abschnitten wird die Funktionsweise von Containern erläutert und mit klassischer Virtualisierung verglichen. Darüber hinaus wird der Aufbau eines Containers sowie die Kommunikation zwischen Containern erläutert. Abschließend werden Container-Cluster sowie deren Orchestrierung betrachtet, die im Cloud Native Stack verstärkt Einsatz findet.

3.3.1.1 Vergleich zwischen Containern und klassischer Virtualisierung

Obwohl in mancher Literatur wie beispielsweise bei [Congfeng Jiang et al., 2016] eine Gleichsetzung zwischen klassischen Hypervisoren und Containersoftware stattfindet, gibt es leichte strukturelle Unterschiede zwischen diesen. Vergleicht man die äußere Struktur von Typ 2 Hypervisoren und einer Container-Runtime wie Docker (vergleiche Abbildung 9 und Abbildung 10), so lassen sich auch hier wenige Abweichungen erkennen. Deutlich erkennbar ist jedoch, dass in einem Container kein Gast-Betriebssystem installiert ist. Stattdessen nutzen Container nur diejenigen Bestandteile des Host-System-Kernels, die zur Ausführung der Applikation notwendig sind. So ist es möglich, mehrere Apps auf einem Server oder einer VM laufen zu lassen, die sonst auf den gleichen Netzwerk-Stack oder Namespace zugreifen und sich gegenseitig behindern würden. Voraussetzung dafür ist allerdings, dass das Host-System und der Container die gleiche Kernelbasis besitzen. Daraus resultiert im Vergleich zu Typ 2 Hypervisoren eine komplett unterschiedliche innere Struktur, denn im Gegensatz zu virtuellen Maschinen wird im Falle von Containersoftware keinerlei Hardware virtualisiert.
So ist es zudem möglich, auf eine hybride Container-Architektur aufzubauen, bei der das Host-Betriebssystem bereits einer virtualisierten VM entspricht. Dies wird häufig in Cloud Rechenzentren angewendet, um eine zusätzliche Abschottung zwischen mehreren Nutzern zu gewährleisten. Ein solches Vorgehen wird auch in vielen Fällen als *hybrid containerization* bezeichnet.
Aus den oben genannten Gründen ist aktuell eine „Containerisierung" von Linux Apps auf dem Betriebssystem Windows nur erschwert möglich. Die häufigste Lösung besteht darin, mithilfe eines Typ 2 Hypervisors eine Linux VM zu erzeugen, auf welcher wiederum Docker ausgeführt wird. Die Installation des „Docker Desktop for Windows" setzt daher Hyper-V und VirtualBox voraus [Docker Inc., 2019f].

3.3.1.2 Funktionaler Aufbau

Die Funktionsweise von Containern basiert auf drei Eigenschaften des Linux Kernel: *namespaces*, *chroot* und *cgroups*. Das wichtigste Feature zur Erstellung eines Containers sind dabei die namespaces. Sie wurden mit Kernel-Version 2.4.19 im Jahr 2002 eingeführt und umfassen fünf wesentliche und unterschiedliche Arten von namespaces, die nachfolgend in absteigender Reihenfolge bezüglich der Wichtigkeit im Kontext von Containern dargestellt sind [Liebel, 2017, Seite 69] [man7.org, 2021b]:

i) **IPC namespaces**

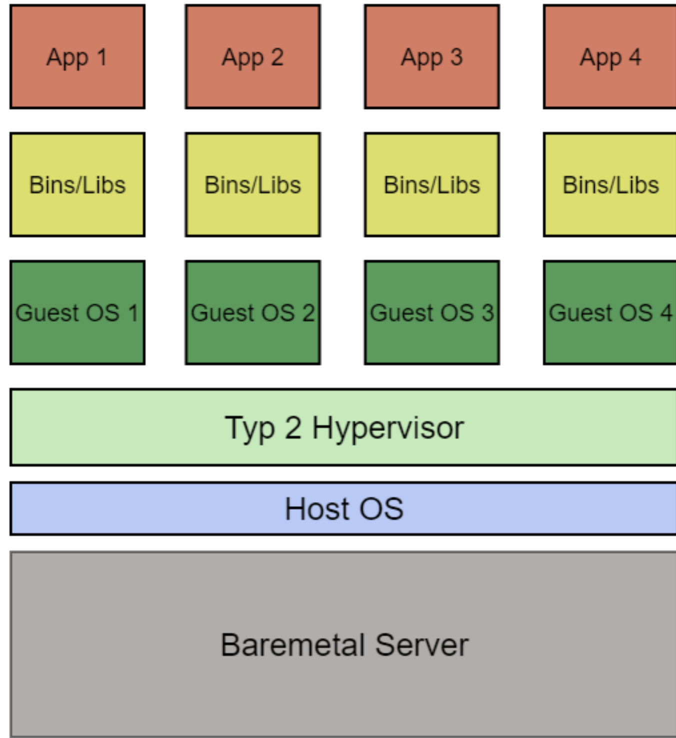

Abbildung 9: Struktureller Aufbau von klassischer Virtualisierung im Kontext eines Hypervisors Typ 2 (eigenen Abbildung)

Dienen der Isolation von Inter Prozess Kommunikationsressourcen. Dazu zählen zum Beispiel POSIX message queues. Wird unter anderem für Shared Memory Mechanismen verwendet, bei dem zwei Prozesse einen Teil des Hauptspeichers gemeinsam nutzen können.

ii) **mount namespaces**
Dienen zur Erstellung eines eigenen Dateisystems innerhalb des Host-Dateisystems. Dabei wird ein neuer Mountpoint erzeugt, der als Einstieg dient.

iii) **network namespaces**
Dienen zur Virtualisierung eines Netzwerkstacks mit eigenem loopback interface. Jeder network namespace besitzt eine eigene private IP-Adresse, Routing Tabellen, Socket und Ports.

iv) **pid namespaces**
Erlauben die Erzeugung eines neuen unabhängigen Hauptprozesses mit PID 1 (beispielsweise eine Bash). Ein neuer Prozess, der als Kind eines solchen Prozesses gestartet wird, erhält eine PID, die abhängig vom neuen Hauptprozess ist. Trotzdem haben alle Prozesse des namespaces auch eine PID in Abhängigkeit vom Hauptprozess des Host-Systems (vergleiche dazu Abbildung 11).

v) **User namespaces**

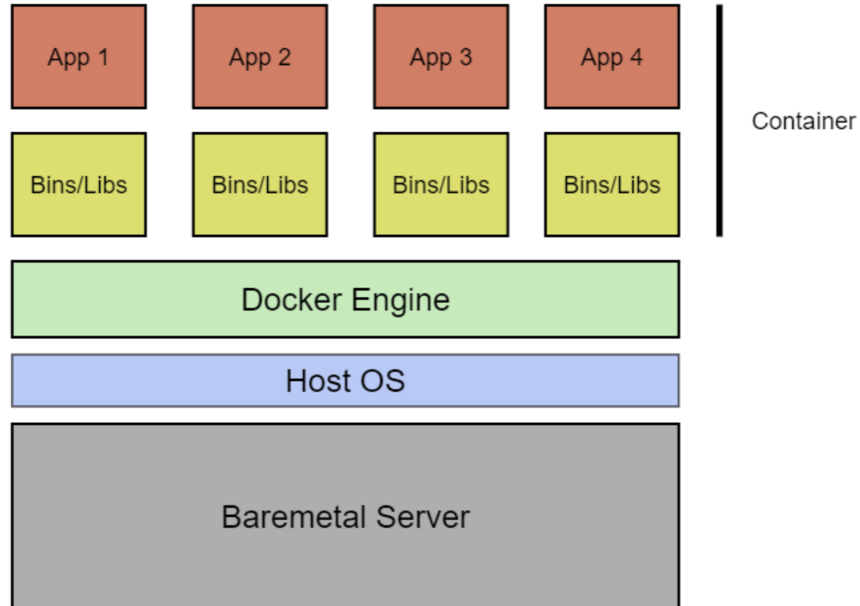

Abbildung 10: Struktureller Aufbau von Containertechnologien (eigenen Abbildung)

Dienen der Abschottung von sicherheitsrelevanten Informationen wie User IDs und Gruppen IDs sowie root Directories und Berechtigungen. So ist es möglich, dass ein Prozess innerhalb des namespaces volle Privilegien genießt, während er außerhalb eingeschränkt arbeitet:

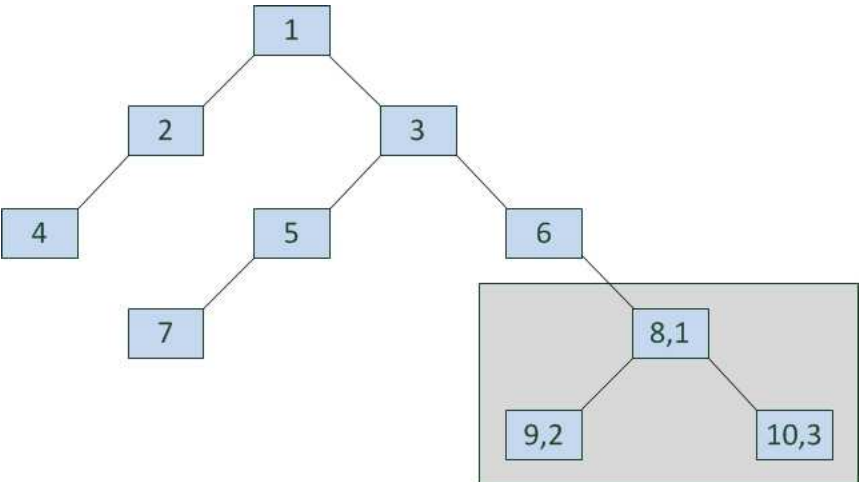

Abbildung 11: PID Namespaces für Host und Container (grauer Kasten) ([Liebel, 2017, Seite 70])

Das namespace-System greift dabei auf ein weiteres Kernel-Feature zurück, das bereits vor der Linux-Zeit Bestandteil im Unix Kernel war. Mit dem Befehl chroot ist es möglich, das root Verzeichnis eines Prozesses anzupassen. Dies wirkt sich dann wieder auf die Kindprozesse aus. Erst durch dieses Feature ist die Isolation des Users und der Usergruppen möglich [man7.org, 2021a].
Mithilfe der zuletzt beschriebenen Funktionen ist es also möglich, isolierte Bereiche zu erzeugen, in

denen abgeschottete Prozesse und Nutzer auf einem ebenfalls abgeschotteten Dateisystem arbeiten. Mit Hilfe der Kernel Funktion cgroups, die seit Kernel-Version 2.6.24 (2007) zur Verfügung steht, ist es nun machbar, die Ressourcen einer Sammlung von Prozessen innerhalb eines namespace zu limitieren. Zu diesen Ressourcen zählen unter anderem CPU, Arbeitsspeicher, Netzwerk und Disc I/O [man7.org, 2021].

3.3.1.3 Innerer Schichtenaufbau von Containern: Das Layer-System

Wie bereits erwähnt, besteht ein Container nur aus denjenigen Teilen des Host-System-Kernels, die zum Betrieb einer Applikation notwendig sind. Dazu gehören korrespondierende Pakete und Userland-Teile des Host-OS [Liebel, 2017, Seite 97]. Um beim Neuaufbau eines Containers nicht von Null starten zu müssen, gibt es sogenannte Images, die bestimmte Teile des Kernels mit sich bringen, um Software ausführbar zu machen. So gehört beispielsweise zum offiziellen golang-Image [Docker Inc., 2019d] nur der Go-Compiler, der die Übersetzung von Go-Programmen in Maschinensprache ermöglicht. Im Gegenzug dazu gibt es Images, die oftmals mehr Pakete beinhalten als zum Ausführen einer Applikation notwendig sind. Hierzu zählt beispielsweise das offizielle Ubuntu-Image [Docker Inc., 2019e], welches auch CLI-Pakete wie *nano*, *ping* oder *apt* enthält, die oftmals nicht benötigt werden. Auf alle Bestandteile eines Images, die zum Starten eines Containers gebraucht werden, sind dabei nur lesende Rechte vergeben. Verändert man nun innerhalb des Containers, durch eine Operation wie beispielsweise das Hinzufügen eines Pakets mit Hilfe von apt oder das Schreiben einer Datei, dessen Dateisystem, so wird eine neue Schreib-/Leseschicht (im Folgenden R/W-Layer) auf das unveränderbare Image gelegt (vergleiche Abbildung 12). Bei jeder Lese-Operation müssen im Gegenzug alle erzeugten Schichten durchlaufen werden, um denjenigen Layer zu finden, welcher die gesuchte Datei enthält. Exportiert man einen angepassten Container als neues Basis-Image, so ist dieses Image um die Anzahl an veränderten Schichten größer als das ursprüngliche Image. Auf diese Weise ergibt sich ein Schichtensystem, welches in der Containerwelt sinnvoll genutzt werden kann. Die Anzahl an Schichten ist dabei aber in vielen Container-Systemen beschränkt.
Auch zu erwähnen ist, dass die Schreib-/Leseschicht nur beim Export eines Containers in ein neues Image übernommen wird. Andernfalls wird diese Schicht gelöscht.
Um die Ablage der Layer eines Containers im Host-Dateisystem und deren Strukturierung kümmern sich sogenannte Storage Driver. Da diese in jedem Container-System variieren, wird erst zu einem späteren Zeitpunkt genauer auf die Docker Storage Driver und deren Funktion sowie Vor- und Nachteile eingegangen.
Die Vorteile des Schichtensystems im Allgemeinen ergeben sich im Caching und der Skalierung von Containern. Container mit gleicher zu Grunde liegender Layer-Basis unterscheiden sich wie oben gezeigt nur durch ihre R/W-Layer. Daraus ergibt sich die Möglichkeit des sogenannten Layer-Sharings. Die Layer eines Images werden dabei nur einmal vom System indexiert und liegen allen Containern mit gleicher Image-Basis zu Grunde, was zu noch schlankeren Containern führt (vergleiche Abbil-

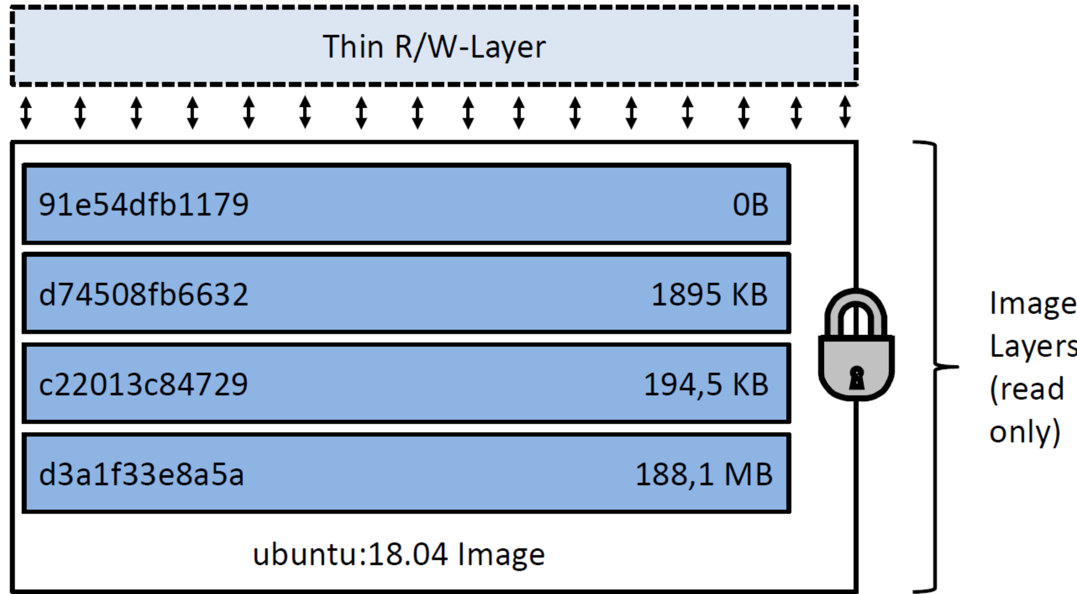

Abbildung 12: Beispiel des Layer-Systems anhand eines Docker-Containers auf Basis eines Ubuntu-Images (eigene Abbildung gemäß [Docker Inc., 2019a])

dung 13). Als Nachteil ergibt sich dadurch, dass das Container-System stets auf Kollisionen geprüft werden muss, um im Zweifelsfall einzugreifen. Werden Container nun skaliert, um beispielsweise eine balancierte Lastverteilung zu erreichen, so durchlaufen alle Container des gleichen Typs die genau identischen Layer, wenn sie exemplarisch auf ein zuvor installiertes Paket oder eine Datei zugreifen möchten.

Auf eine ähnliche Mechanik kann beim Erzeugen neuer Images zurückgegriffen werden. Bereits erzeugte Images können als Caching-Grundlage verwendet werden. Die zuvor gebildeten Layer werden dann nur kopiert. Dies verringert die Dauer des Build-Prozesses und sorgt im Gegenzug wieder für eine in Teilen identische Layer-Basis, was wiederum das Layer-Sharing begünstigt.
Daten, die in Layern abgespeichert werden, sind persistent im Container vorhanden. Wird der Container versehentlich gelöscht, sind somit alle erzeugten Daten im Container verloren. Dies kann besonders für Datenbanken oder Log-Dateien schwerwiegende Folgen haben. Um diesem Problem entgegenzuwirken, gibt es in den meisten Container-Softwares ein Volume-System. Dabei werden Teile des Containers von der Indexierung durch das Layer-System ausgeschlossen. Zusätzlich werden die ausgeschlossenen Ordner dem Namespace der Container-Software zugeordnet, auf welchen sowohl das Host-System als auch der Container Zugriff haben. Wird der Container nun gelöscht, bleiben die Daten des Volumes auf dem Host erhalten und können zukünftig zu anderen Containern hinzugefügt werden. Im späteren Verlauf der Arbeit wird gesondert auf das Docker Volume-System eingegangen,

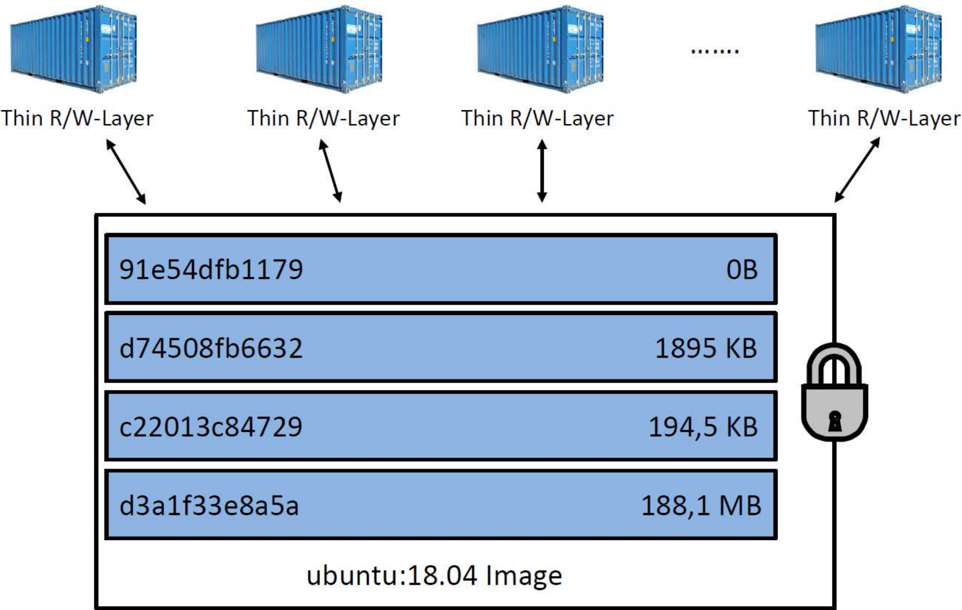

Abbildung 13: Beispiel des Layer-Sharings anhand mehrerer Docker-Container auf Basis eines Ubuntu-Images (eigene Abbildung gemäß [Docker Inc., 2019a])

da es den Energieverbrauch von Docker sowohl positiv als auch negativ beeinflussen kann.

3.3.1.4 Netzwerke zwischen Containern

Neben dem Layer-System bildet die Kommunikation unter Containern ein weiteres großes Themengebiet ab. Nachfolgend werden die Funktionsweise der Kommunikation und ihre Besonderheiten nur kurz erläutert. Im Verlauf der Arbeit wird dann noch einmal genauer auf das Networking im Kontext von Docker eingegangen.

Da es Containern möglich sein muss, miteinander zu kommunizieren, werden diese mit virtuellen Netzwerken verbunden. Im Allgemeinen erhalten Container dabei eine IP-Adresse aus dem privaten IP-Netz des Host-Systems (172er Netz). Dies erlaubt zum einen die Nutzung von Linux iptables (zur automatischen Paketweiterleitung und Verbindung mit der Außenwelt) und zum anderen macht dies Service-Discovery (Container finden sich anhand ihres Namens) möglich. Durch Letzteres wird die Erreichbarkeit von Services erhöht, da diese auch mit anderer IP-Adresse, rein durch ihren Namen, stets auffindbar sind. Die statische Vergabe von IP-Adressen wird jedoch nicht ausgeschlossen und steht als zusätzliche Option zur Verfügung. Es gibt verschiedene Aufbauarten von virtuellen Netzen zur Verbindung von Containern. Eine wichtige Frage hierbei: Handelt es sich um ein Single-Host-System oder um einen verteilten Cluster? Im Falle eines Single-Host Systems können virtuelle *Bridge*-Netzwerke zur Verbindung von Containern verwendet werden. Zur Kommunikation mit außerhalb des Hosts befindlichen Systemen wird dann eine Portweiterleitung sowie NAT eingesetzt.

Sollen Container miteinander verbunden werden, die sich auf zwei unterschiedlichen Host-Systemen befinden, so werden hierfür Container-Cluster-Tools verwendet. Diese werden genauer in Ab-

satz 3.3.1.5 beschrieben. Überwiegend werden hier zur Realisierung der Netzwerkverbindung sogenannte *Overlay*-Netze verwendet. Diese gebrauchen oftmals Tunnel oder Datenverkapselung, um Pakete sicher zwischen mehreren Host-Systemen zu transportieren. Eine tiefgreifendere Erläuterung der Funktionsweise von Overlay-Netzen wird an dieser Stelle nicht ausgeführt, da sie für die restliche Arbeit nicht von Belang ist (für mehr Informationen vergleiche [Kurose and Ross, 2012, Seite 184]). Es sollte jedoch erwähnt werden, dass es mehrere unterschiedliche Ansätze für die Umsetzung von Netzwerken zwischen Containern gibt, insbesondere Bridge-Netzwerke, die besonders im Docker-Kontext eine Rolle für diese Arbeit spielen.

3.3.1.5 Container-Cluster und Orchestrierungstools

Wie bereits in den letzten Kapiteln deutlich gemacht, wird durch Container die Ausführung vieler parallel laufender Berechnungseinheiten auf einem Server möglich, was zu einem erhöhten administrativen Aufwand führt. Durch die in Absatz 3.3.1.4 beschriebene Overlay-Technik ist es zudem möglich, die Container-Landschaft über mehrere Server hinweg, also auf einem Server-Cluster, zu betreiben. Um hierbei nicht den Überblick zu verlieren, werden dem Container-Nutzer eine Vielzahl von Container-Orchestrierungstools an die Hand gegeben, um Container über Servergrenzen hinweg zu verwalten und in eingeschränktem Maße deren Betrieb zu automatisieren. Diese Orchestrierungstools fassen dabei mehrere VMs als Gruppe von Master- und Worker-Nodes zusammen, deren Aufgabe sich entweder auf die Verwaltung (Master-Node) oder die Ausführung (Worker-Node) der Container bezieht. Die VMs können dabei auf demselben „harten" Server oder auf verteilten Anlagen laufen. Dadurch entstehen sogenannte Container-Cluster, auf denen mehrere Millionen Container gleichzeitig ausgeliefert und betrieben werden können. Das Orchestrierungstool erlaubt dabei oftmals eine automatisierte Lastbalancierung zwischen Containergruppen gleichen Typs. Da diese Kopien von Services sind, die jedoch auch die identischen Netzwerkeigenschaften mit sich bringen, werden sie als Replikas bezeichnet. Des Weiteren erlauben Orchestrierungstools oftmals eine automatisierte horizontale Skalierung (Anzahl Replikas wird je nach Anforderungen erhöht oder verringert) der Container sowie der VMs (VMs werden je nach Ressourcenauslastung konsolidiert oder hinzugefügt). In manchen Tools ist zudem eine vertikale Skalierung der Container möglich. Hierbei werden dem Container entsprechend seiner Anforderungen Hardware-Ressourcen zugesprochen oder entnommen. Man spricht bei vertikaler Skalierung auch oftmals von *Elastizität* oder dem *Scale up*, während man horizontale Skalierung unter anderem als *Scale out* bezeichnet.

Um den administrativen Aufwand bezüglich der Container zu verringern, fassen Orchestrierungstools Container in Containergruppen oder -Umgebungen zusammen. Je nach Anwendung werden diese oftmals unterschiedlich bezeichnet. Da diese Arbeit sich aber hauptsächlich mit dem Orchestrierungstool Kubernetes (im Folgenden auch K8) befasst, wird hier der dort verwendete Begriff des Pods verwendet. Pods fassen Container zusammen, die explizit miteinander in Verbindung stehen. Beispielsweise ein Container mit einem Webserver und ein Container mit einer dazugehörigen Datenbank (vergleiche

Abbildung 14).

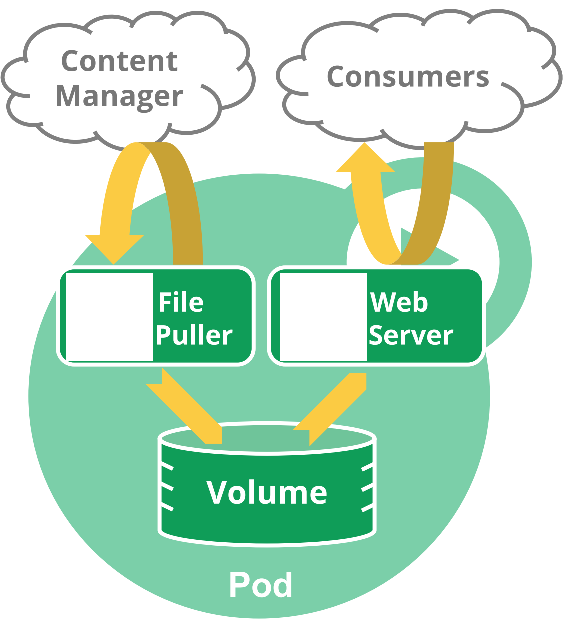

Abbildung 14: Beispiel für einen Pod als logische Einheit im Kubernetes Kontext [Kubernetes, 2021b]

Da diese Container sowieso eine logische Einheit bilden, verwenden sie innerhalb eines Pods die gleichen Namespaces. Dies führt zum einen zu einer verbesserten Kommunikation über das gemeinsame Loopback Interface (Container innerhalb eines Pods haben also die gleiche IP-Adresse) sowie dazu, dass die Applikationsprozesse innerhalb des Pods alle anderen Prozesse des Pods sehen können, unabhängig davon, ob sie sich im selben Container befinden. Netzwerk-Namespaces werden hierbei aber nicht überflüssig, da sich auch Pods, die nicht zu einer logischen Einheit gehören, ein virtuelles Netz teilen können. Beispielsweise, wenn ein eigenständiger Webservice mit Datenbank bereitgestellt wird, auf den wiederum andere eigenständige Webservices zur Informationsgewinnung zugreifen (vergleiche Abbildung 15).

Die inneren Mechaniken von Container-Clustern sind umfangreich und komplex. Im Verlauf dieser Arbeit wird auf diese regelmäßig verwiesen, da sie im Bereich der Cloud eine große Rolle spielen. Zudem wird zu einem späteren Zeitpunkt (siehe Abschnitt 3.4.6) intensiv auf die zuvor erwähnte Software Kubernetes und dessen automatisierte horizontale Skalierung eingegangen.

3.3.1.6 Fazit: Vor- und Nachteile von Containern sowie Abgrenzung zu anderen Virtualisierungstechniken

Wie in den letzten Kapiteln gezeigt, ist der Unterschied zwischen VMs und Containern im Detail deutlich ersichtlich. Daraus ergeben sich auch Vor- und Nachteile in der Nutzung von Containern. Im Folgenden werden diese kurz dargestellt und erläutert:

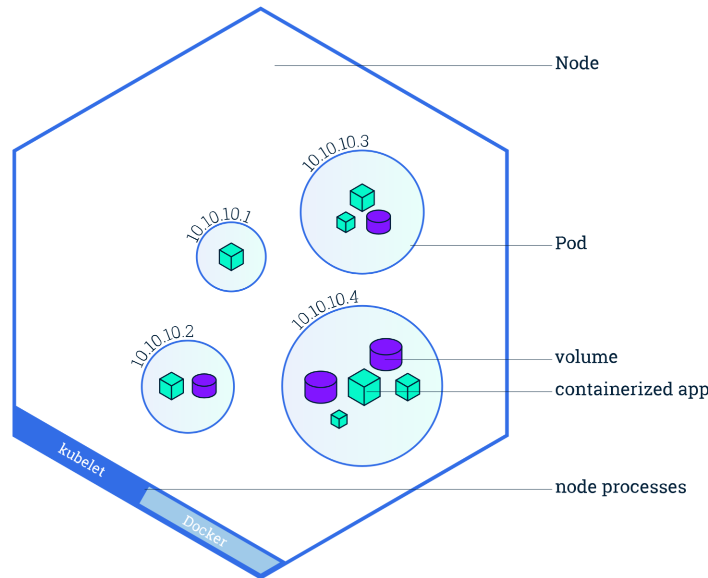

Abbildung 15: Beispiel für mehrere Pods im Kontext eines Kubernetes Nodes [Kubernetes, 2021a]

Vorteile:

i) **Ressourcen-Auslastung**
Die Auslastung der Hardware ist aufgrund des Aufbaus von Containern deutlich geringer als die von VMs, da weder virtualisierte Hardware noch BIOS/UEFI zur Ausführung notwendig sind. Da Software in isolierten Bereichen nebenläufig ausgeführt werden kann, erhöht sich dadurch die Packungsdichte im Vergleich zu VMs um ein Vielfaches. Darüber hinaus verbrauchen Container weniger Speicherplatz.

ii) **Skalierbarkeit**
Aufgrund der Größe eines Containers und der höheren Start-Geschwindigkeit wird die Skalierbarkeit im Rahmen von Containern erhöht. Im Falle von Engpässen können so neue Container fast just-in-time zur Verfügung gestellt werden.

iii) **Portabilität**
Da Container nicht an die Hardware, sondern nur an den Kernel des Host-Systems gebunden sind, ist die Übertragbarkeit eines Containers auf ein anderes System unproblematisch.

iv) **Continuous Integration**
Container unterstützen aufgrund des schnellen Startverhaltens agile Entwicklungsmodelle und somit auch Continuous Integration und Delivery.

Nachteile:

i) **Sicherheit**
Um die Sicherheit eines Containers zu gewährleisten, bedarf es einiger vorbereitender Arbeit. Dies betrifft den Umgang mit Daten, auf deren Persistenz hier im Besonderen geachtet werden muss, da Daten, die sich im Container befinden, auch mit diesem gelöscht werden können. Zum anderen müssen speziell die Netzwerke, in denen Container miteinander interagieren und die Dateisysteme, die außerdem für den Aufbau von Containern zuständig sind, beachtet werden. Oftmals werden dafür Containersoftware-unabhängige Plug-ins verwendet, die diese Bereiche zusätzlich absichern sollen.

ii) **Ressourcen-Verteilung**
Da Ressourcen über alle Container verteilt sind, können diese in Wechselwirkung zueinander treten und sich gegenseitig beeinflussen, obwohl sie faktisch voneinander getrennt sind. Aufgrund der geteilten Ressourcen wie Speicher, CPU oder Netzwerk können einzelne Container zu Single Point of Failures werden. Aus diesem Grund bedarf es bei der Nutzung einer großen Anzahl von Containern stetiger Überwachung durch ein übergeordnetes System in Form eines Orchestrierungstools. Ähnliche gibt es auch bei VMs, wobei der Hypervisor, der Voraussetzung für die VMs ist, ein solches Monitoringtool in den meisten Fällen mitbringt.

Hieraus ergibt sich, dass die Vorteile in der Nutzung von Containern besonders im Falle von Flexibilität und Ressourcenauslastung überwiegen, was wiederum ihren aktuellen Aufstieg rechtfertigt. Wie jedoch gezeigt, können einige Eigenschaften von Containern nur mithilfe von zusätzlicher Software gewährleistet werden. Um darüber hinaus eine Vielzahl von Containern parallel auf einem Server zu betreiben, bedarf es aufgrund hoher Komplexität einer Container-Cluster Software.

3.3.2 Aufbau und Funktionsweise von Docker

Nachdem in den vorherigen Abschnitten beschrieben wurde, wie Container im Kern funktionieren, beschäftigt sich der folgende Abschnitt mit der Umsetzung dieser Kernfunktionalitäten im Falle von Docker. Dabei wird auf spezielle Bestandteile, wie Dockerfiles und Build-Befehle besonders detailliert eingegangen, da diese für die in Kapitel 7 dargestellten Messungen essentiell sind und dort auch praktische verwendet sowie verändert werden. Weiter wird sich mit den Einzelbestandteilen des Docker-Ökosystems beschäftigt, die von containerd verwaltet werden. Dazu zählen vor allem die Images, das Networking, Volumes sowie das Layer-System und in Teilen auch die Orchestrierung und das Monitoring der Container.

3.3.2.1 Information zum Docker Prozess Stack

Wie in Abbildung 16 zu erkennen, hat sich die Prozessstruktur von Docker im Laufe der Zeit deutlich geändert. Mit Einführung von runC und containerd wurden zwar zwei neue Prozesse zum Prozess-

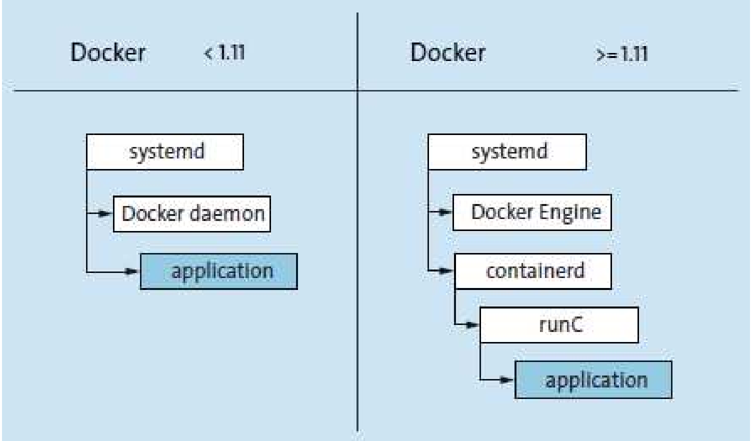

Abbildung 16: Funktionaler Docker Prozess Stack Pre-/Post 1.11 [Liebel, 2017, S. 82]

stack von Docker hinzugefügt, diese sind nun jedoch einfacher für Docker zu warten, nutzen darüber hinaus nicht mehr die VM-Feautures von LXC und sind ebenfalls Open Source. Die dabei aufgelöste Komponente, der Docker Daemon, griff zuvor auf LXC zu, um Container zu erzeugen. LXC wurde dabei durch ein eigenes Command Line Interface (CLI) für den Nutzer zugänglicher gestaltet. Nach Implementierung von containerd und runC änderte sich auch das CLI, welches nun Bestandteil der Docker Engine ist. Die Docker Engine dient zur Kommunikation mit dem Nutzer. Sie stellt ein CLI sowie die Konfiguration von Docker-Endpunkten zur Verfügung. Diese Endpunkte können dabei auf localhost liegen oder remote per TCP angebunden werden. Die Kommunikation zwischen CLI und dem darunterliegenden containerd findet dabei über gRPC (Go Remote Procedure Call) statt. Während die Docker Engine in erster Linie ein CLI-Paket ist, bildet containerd, wie schon erwähnt, das Herz von Docker. Die Runtime erledigt alle Low-Level-Container-Management-Aufgaben, Speicherung, Image-Verteilung sowie Netzwerkanbindung und verarbeitet Anfragen von der Docker Engine. Dabei ist zu beachten, dass containerd ohne Docker Engine lauffähig ist, Docker Engine ohne containerd jedoch nicht. Überdies wird heute oftmals containerd als alleinstehende Container-Runtime in Orchestrierungstools wie Kubernetes verwendet, da die Docker-Engine für diese nicht notwendig ist. Das weiter unter containerd liegende runC hingegen stellt Schnittstellen zu Linux-Funktionen wie cgroups und Namespaces zur Verfügung und kann durch andere Runtimes ausgetauscht werden. Genaueres zur Docker Engine und im Speziellen zur Docker CLI sowie zu containerd und runC wird an dieser Stelle nicht besprochen. Hier kann die ausführliche, offizielle Dokumentation von Docker zurate gezogen werden [Docker Inc., 2021e][Docker Inc., 2021b].

3.3.2.2 Aufbau von Images, der Build-Prozess und das Starten und Stoppen von Containern

Um Applikationen in Docker und anderen Containerumgebungen lauffähig zu machen, benötigt man sogenannte Images. Ein Image beschreibt dabei die als Basis dienende Ordnerstruktur innerhalb eines Containers. Um einen Container lauffähig zu gestalten, benötigt es daher mindestens das Null-Gerät /dev/null, so wie in POSIX-konformen Linux-Systemen üblich. Möchte man daher einen möglichst

kleinen Container bauen, muss man entweder /dev/null seines eigenen Linux-Systems in eine tar-Datei packen und diese in Docker importieren oder man nutzt das in jeder Docker-Installation enthaltene Basis-Image **scratch**. Benutzt man dieses Basis-Image, ist es sinnvoll, hieraus wiederum ein neues Image zu erzeugen. Hierfür schreibt man eine sogenannte Dockerfile, die als Basis scratch verwendet. In Listing 1 sehen wir die Erzeugung eines Containers, in welchem die ausführbare Datei *server* gestartet wird. Das Keyword *FROM* beschreibt dabei, welche Image-Basis genutzt werden soll. Der Befehl *EXPOSE* öffnet explizit Ports des Containers.

```
FROM scratch
WORKDIR /app
COPY server .
EXPOSE 8000
CMD ["./server"]
```

Listing 1: Dockerfile zur Erzeugung eines Images auf Basis von scratch

Dockerfiles können zusätzlich derart gestaltet werden, dass sie mehrstufig sind. So kann man beispielsweise in einem Container eine Applikation bauen, den Container automatisch stoppen, löschen und danach die erzeugte Applikation in einem Container mit Basis-Image scratch laufen lassen (siehe hierzu Listing 2).

```
FROM golang

WORKDIR /app
COPY server.go .
ARG CGO_ENABLED=0
RUN go build -o server .

FROM scratch

WORKDIR /app
COPY --from=0 /app .
EXPOSE 8000
CMD ["./server"]
```

Listing 2: Dockerfile zur Erzeugung eines Images auf Basis von scratch

Genauer soll an dieser Stelle nicht auf die Erzeugung von Dockerfiles eingegangen werden. Sie dienen jedoch im Verlauf der Arbeit, in Verbindung mit den in Absatz 3.3.2.4 vorgestellten Storage-Treibern, als Diskussionsgrundlage und sollten daher zumindest grob vorgestellt werden.
Das oben erzeugte Image verfügt nur über die Fähigkeit, ausführbare Dateien lauffähig zu machen. Benötigt man ein erweitertes Featureset, wie beispielsweise den Compiler einer Programmiersprache oder die Möglichkeit einen Paketmanager wie apt zu nutzen, kann man entweder die notwendigen Be-

standteile seines eigenen Systems als tar-Datei packen und in Docker importieren oder auf die Images des Docker Hub zurückgreifen. Die erste Variante gestaltet sich dabei für Nutzer ohne tiefgreifende Kenntnisse der Linux-Architektur als schwierig. Dokumentationen sind nur in eingeschränktem Maße zu finden und meist an Beispiele gebunden wie in [Liebel, 2017, S. 153]. Docker Inc. äußert hier zusätzlich klar den Wunsch, die Plattform Docker Hub zu nutzen, in welcher jeder Nutzer Images bereitstellen kann. Man findet an dieser Stelle auch von Docker bereitgestellte, zertifizierte und geprüfte Images wie exemplarisch alpine-Linux, ubuntu, golang, node, MySQL und viele mehr [Docker Inc., 2021g]. Möchte man aus einer Dockerfile ein Image erzeugen, nutzt man das Docker-CLI Kommando wie in Listing 3 dargestellt.

```
$docker build -f /OrdnerZurDockerfile/Dockerfile . -t myimage
```

Listing 3: Erzeugung eines Images mit Hilfe einer Dockerfile

Das in Listing 3 erzeugte Image erhält den Tag *myimage* und kann mit Hilfe des Kommandos *docker image ls* in der Liste der verwendbaren Images angezeigt werden. Im Docker-Kosmos spielen die Tags eine wichtige Rolle, haben allerdings keinerlei Einfluss auf diese Arbeit. Es ist nun möglich, einen Container aus dem oben erzeugten Image zu erstellen und zu starten. Listing 4 zeigt dabei ein nacheinander ausgeführtes Erzeugen und Starten eines Containers, mit Namen *mycontainer*, mithilfe des Kommandos *run*.

```
$docker run -d -p 8000:8000 --name mycontainer myimage
```

Listing 4: Starten eines Containers mit Basis-Image myimage

Es ist zu beachten, dass auf diese Weise immer nur ein Container nach dem anderen gestartet werden kann. Dies bietet sich im Allgemeinen für das Prototyping, Integration-Tests oder das Aufsetzen vorgefertigter Komplettsysteme (wie beispielsweise eine Owncloud oder Gitlab) an. Möchte man den Container nur stoppen, so reicht das Kommando *docker stop mycontainer* aus. Der Container kann danach ebenfalls neu gestartet werden, indem man den Befehl *docker start mycontainer* nutzt. Wird der Container nicht explizit mit dem Kommando *docker container rm mycontainer* vom System entfernt, kann er jederzeit gestoppt und gestartet werden.

3.3.2.3 Orchestrierung mit Docker-Compose

Wie bereits im Abschnitt zuvor erwähnt, bietet sich der Befehl *run* nur für die Erzeugung einzelner Container an. Möchte man Containerumgebungen mit mehreren Containern verwalten, nutzt man dafür das Docker-Subsystem **docker-compose**. Beide Fälle sind dabei in der praktischen Anwendung gleichermaßen zu finden. Einzelne Container stellen dabei beispielsweise statische Seiten oder unabhängige Services wie zum Beispiel Message Queues zur Verfügung, während Containerumgebungen eher für die Bereitstellung ganzer Software-Stacks dienen (Webseite, Datenbank, Mailer, etc.).

Auch bei der Nutzung von docker-compose werden Dateien (im yaml-Format) zur Beschreibung von Containern verwendet. In Listing 5 wird derselbe Container erzeugt und gestartet wie zuvor in Listing 4.

```
version: '3'
services:
        server:
                image:go_api
                ports:
                        - "8000:8000"
```

Listing 5: Beispiel für eine einfache docker-compose.yaml

Benutzt man docker-compose für den Start von Container-Umgebungen, kann dies exemplarisch wie in Listing 6 aussehen. Hier wird eine ownCloud mit dahinter liegender Datenbank aufgesetzt. Datenbank und Frontend werden einem privaten Netz zugeordnet, wodurch nur der ownCloud-Container auf die Datenbank zugreifen kann. Mit Hilfe von *links* wird der ownCloud ein alternativer Hostname der Datenbank mitgeteilt. Beide Container sind derart gestaltet, dass sie bei Ausfall automatisch neu gestartet werden. Zusätzlich werden die Daten der Datenbank in einem Volume gespeichert. Dadurch sind diese nicht persistent im Container, sondern auf dem Host-System verfügbar und werden im Falle des Löschens der Container ebenfalls nicht entfernt.

```
version: '3.7'

services:

  owncloud:
    image: owncloud
    restart: always
    ports:
      - 8080:80
    links:
      - "db:database"
    networks:
      - backend
  mysql:
    image: mariadb
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: example
    volumes:
       - db-data:/var/lib/mysql/data
    networks:
      - backend
```

```
volumes:
  db-data:

networks:
  backend:
    driver: bridge
```

Listing 6: docker-compose.yaml für eine Container-Umgebung mit Frontend und Datenbank

Es besteht auch die Möglichkeit, innerhalb einer Compose-Datei eine Dockerfile anzugeben. Bevor der Container gestartet wird, findet dann der klassische Build-Prozess von Docker statt.

Zum Starten der Container genügt es nun, mithilfe des Befehls *up* die Container-Umgebung vollständig zu starten (siehe Listings 6).

```
$docker-compose up -d
```

Listing 7: Starten der Container-Umgebung mit Docker Compose

Die Flag *-d* sorgt dafür, dass die Container im Hintergrund gestartet werden. Im Gegenzug kann die Umgebung mit dem Befehl *down* gestoppt werden.

3.3.2.4 Volumes und das Layer-System

Volumes und das Layer-System sind mit den zuvor bereits erwähnten Storage-Treibern von Docker fest verzahnt. Hauptaufgabe der Volumes ist die Exkludierung von Dateien und Ordnern aus dem Dateisystem des Containers, um so Dateien vor der Persistenz des Containers zu schützen. Ein weiterer Vorteil der Volumes ist die Inkludierung eines Volumes in mehrere Container. Ein Beispiel hierfür wären mehrere Container mit Webserver, die auf dieselben Dateien einer Webseite zugreifen.
Je nach Wahl des Storage-Treibers ändert sich die Layer-Struktur und der Aufbau der Volumes. Für den Nutzer ist die Änderung nicht ersichtlich, allerdings kann die Wahl des Treibers erhebliche Auswirkungen auf die Performance des Container-Systems haben. Die zuvor vorgestellten Dockerfiles spielen dabei ebenfalls eine Rolle, da jede in ihnen verfasste Zeile einem neuen Layer entspricht. Demnach gibt es effiziente und weniger effiziente Möglichkeiten, Dockerfiles aufzubauen (siehe Kapitel 8).
Docker bietet von Grund auf die Option, sich zwischen unterschiedlichen Treibern zu entscheiden. In Tabelle 2 werden die per Default in Docker zur Verfügung gestellten Treiber dargestellt. Da deren Nutzung abhängig vom Dateisystem des Hosts ist, finden sich in der Tabelle auch Anmerkungen zu den Voraussetzungen und den Eigenschaften eines jeden Treibers. Die meisten Treiber basieren auf dem Copy-On-Write-Verfahren (CoW), welches sich jedoch in jedem Treiber in seiner Ausprägung unter-

scheidet. Auf die unterschiedlichen Verfahren wird in Kapitel 7 in Form von Anwendungsbeispielen näher eingegangen.

Treiber	**Nutzungsvoraussetzung**	**Eigenschaften**
overlay2	xfs oder ext4 als Host-Dateisystem	unterstützt Shared Page Caching; hohe mögliche Packungsdichte und gute PaaS-Eignung
aufs	xfs oder ext4 als Host-Dateisystem	CoW ganzer Dateien; unterstützt Shared Page Caching; gut für PaaS und Anwendungsfälle mit hoher Container-Dichte
devicemapper	direct-lvm als Host-Dateisystem	langjährige und stabile Code-Basis; blockbasiertes CoW, daher performant beim Lesen und Schreiben; kein Shared Page Caching
btrfs	btrfs als Host-Dateisystem	performanter Copy-up-Vorgang; nicht geeignet für hohe Packungsdichte, da starke Fragmentierung; keine CoW
zfs	zfs als Host-Dateisystem	Block-Sharing; sehr performant und skalierbar; nutzt CoW-Mechanismen; stabile Code-Basis; gut für PaaS und Anwendungsfälle mit hoher Container-Dichte, aber Nutzung offiziell von Docker nicht empfohlen
vfs	Host-Dateisystem egal	kein CoW; kein Union Filesystem; geringe Performance und hoher Speicherverbrauch, aber robust und stabil

Tabelle 2: Überblick über die Docker Storage Treiber [Liebel, 2017, Seiten 307-341]

Das weiter oben und in Tabelle 2 erwähnte CoW kann auf der einen Seite je nach Ausprägung eine Verbesserung der Performance von Containern mit sich bringen. Hat man jedoch eine hohe Anzahl an Schreib- und Lesevorgängen in einem Container zu erwarten, verschlechtert dies wiederum die Leistung so sehr, dass die zuvor genannten Vorteile überlagert werden.
Aus diesem Grund findet man in der Dokumentation der Docker Storage-Treiber folgenden Eintrag (beispielsweise [Docker Inc., 2021h]):

> „**Use volumes for write-heavy workloads:** Volumes provide the best and most predictable performance for write-heavy workloads. This is because they bypass the storage driver and do not incur any of the potential overheads introduced by thin provisioning and copy-on-write. Volumes have other benefits, such as allowing you to share data among containers and persisting even when no running container is using them. “

Wie bereits erwähnt, dienen Volumes dazu, Daten aus der Persistenz des Containers zu „befreien“ und für Anwendungszwecke, die nach der Löschung eines Containers existieren, bereitzustellen. Alternativ ist auch so die Nutzung einer einzigen Datenbasis durch mehrere Container möglich. Der Einsatz

ist hier jedoch deutlich an die Applikation gebunden. Applikationen, die keine Daten beschreiben, benötigen in der Regel auch keine Volumes. Demnach gibt die Applikation den Grundaufbau eines Containers vor. Für Applikationen, die Schreibende Operationen ausführen möchten, jedoch keine Kollisionsauflösung oder Multi-Write-Operationen auf Clustersystemen beherrschen, bieten sich Volumes für das Datasharing nicht an, da dies sonst schnell zu Dateninkonsistenz führt. Für Webseiten oder Applikationen mit Kollisionsauflösung, wie Datenbanken, können Volumes für das Datasharing jedoch benutzt werden.
Um Volumes unabhängig von der Applikation für das Datasharing benutzen zu können, bedarf es dem Einsatz von Third Party Volume Plug-ins, die beispielsweise die Nutzung von verteilten Storage-Lösungen, wie exemplarisch Ceph [Ceph, 2021], erlauben.

3.3.2.5 Networking

Mit dem Networking stellt Docker eine Thematik bereit, die von ihrer Komplexität her Inhalte für eine eigenständige Arbeit bietet. Dies wird besonders deutlich, wenn man die Anzahl der externen „Network-Driver“ (im Folgenden auch Netzwerk-Treiber) betrachtet, die in Docker eingebunden werden können. Alleine die CNCF beheimatet mehr als 20 Unterprojekte, die sich nur um das Networking in Cloud Umgebungen kümmert [CNCF, 2021]. Aus diesem Grund kann in dieser Arbeit keine vollständige Erläuterung der technischen Details hinter dem Networking erfolgen. Wichtiger ist hier auch die Funktionsweise in Docker und die Unterschiede der verschiedenen Treiber. Standardmäßig stellt Docker vier Netzwerk-Treiber zur Verfügung, die im Folgenden kurz erläutert werden. Zuerst soll allerdings auf die Notwendigkeit der Netzwerk-Treiber eingegangen werden. Aufgrund der hohen Packungsdichte, die durch Container erzeugt wird, ergeben sich Probleme, denen entgegengewirkt werden muss:

i) Vergabe/Rücknahme von IP-Adressen

ii) Verteilung von Netzwerkressourcen

iii) Kommunikationssicherheit in geschlossenen Container-Umgebungen

Um die doppelte Vergabe von IP-Adressen an gestartete Container zu verhindern, bedarf es eines Systems, das diese Verteilung automatisiert durchführt. Nur so ist es möglich, dass Administratoren die Übersicht über die Container wahren können. Docker stellt zudem ein DNS-System zur Verfügung, sodass größtenteils auf IP-Adressen verzichtet werden kann. Wird ein Container ohne weitere Konfiguration gestartet, erhält er automatisch eine IP aus dem 172er-Netz (voreingestellt, lässt sich über Konfiguration des Daemons ändern). Ferner hat die Packungsdichte großen Einfluss auf die vorhandenen Netzwerkressourcen. Das interne Netzwerksystem von Docker muss in der Lage sein, entsprechende Ressourcen an die einzelnen Container zu verteilen. Besonders im Falle der Skalierung spielt diese automatisierte Verteilung im Sinne des Load-Balancings eine wichtige Rolle. Zu guter Letzt

muss das Netzwerksystem die Möglichkeit geben, Container-Umgebungen voneinander zu trennen, indem verschiedene Netzwerke mit unterschiedlichen Adressbereichen erstellt werden können. Per Default werden diese Netzwerke in Docker in Form eines **Bridge-Netzwerks** erzeugt. Das Bridge-Netzwerk entspricht einem Link-Layer-Device, welches mit Hilfe der *iptables* Pakete zwischen den zum Netz gehörigen Containern weiterleitet. Dabei fungiert das Bridge-Netzwerk als Gateway und kann demnach auch für eine Weiterleitung von Paketen zur Außenwelt genutzt werden. Hierzu besteht die Möglichkeit des Port-Forwardings auf das Host-System. Docker beherrscht darüber hinaus zur Vereinfachung der Administration NAT (Network Address Translation) und DNS (Domain Name Service). Besonders der DNS-Service spielt eine wichtige Rolle in der Konfiguration der Container-Umgebungen, um diese möglichst portabel zu gestalten.

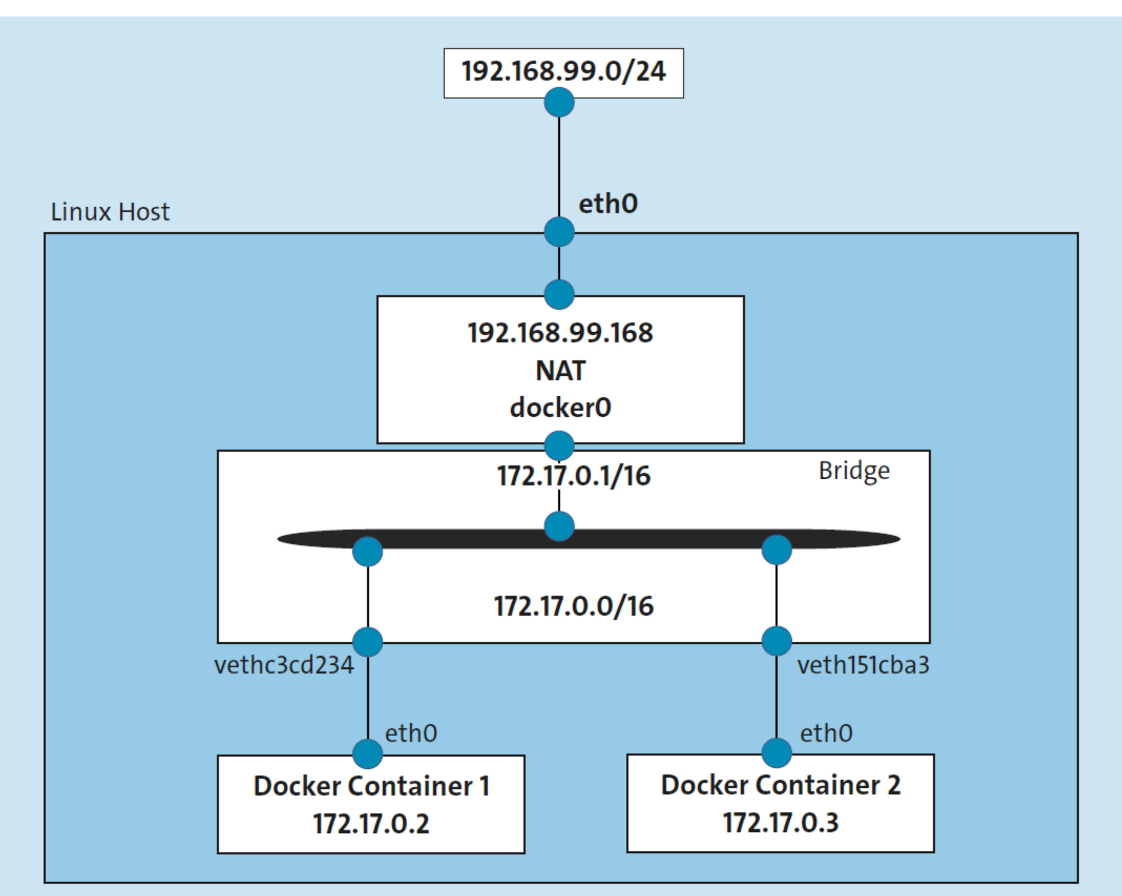

Abbildung 17: Verbindung zwischen Containern, Docker Bridge und dem Host-System [Liebel, 2017, S. 240]

Wie in Abbildung 17 zu sehen ist, bekommt auf diese Weise jeder Container seine eigene IP-Adresse (in Abhängigkeit des Netzwerks) zugewiesen. Per Default werden alle Container dem Netzwerk *docker0* zugeteilt, welches beim Starten von Docker automatisch erzeugt wird. Die Erzeugung anderer Bridge-Netzwerke zur Abtrennung von zusammengehörigen Container-Umgebungen ist ebenfalls möglich.

Alternativ zum Port-Forwarding bietet Docker das **Host-Netzwerk** an. Dieses Netz muss nicht ex-

plizit erzeugt werden, sondern existiert von Beginn an. Ein mit diesem Netz konnektierter Container wird unmittelbar mit dem Host-Netzwerk-Interface verbunden. Dadurch sind unter anderem direkte Zugriffe auf die Systemdienste des Hosts möglich. Aus diesem Grund gilt das System auch als unsicher und sollte produktiv nicht eingesetzt werden.
Manche Applikationen wie exemplarisch Netzwerk-Monitoring-Tools, benötigen direkten Zugang zum physischen Netzwerk. Aus diesem Grund bietet Docker **macvlan** Netzwerk-Treiber. Hierbei wird die MAC-Adresse eines Containers direkt mit dem virtuellen Netzwerk-Interface verbunden. In diesem Fall bedarf es jedoch einer erweiterten Konfiguration des Docker Hosts (Zuweisung eines physischen Interfaces, manuelle Festlegung von Subnetzen und Gateways) sowie der Container (manuelle Verbindung zu Subnetzen und Gateways). In der Dokumentation von Docker bezüglich des macvlan-Treibers wird zudem mit den folgenden Punkten vor dessen Nutzung gewarnt [Docker Inc., 2021c]:

i) „It is very easy to unintentionally damage your network due to IP address exhaustion or to “VLAN spread”, which is a situation in which you have an inappropriately large number of unique MAC addresses in your network.“

ii) „Your networking equipment needs to be able to handle “promiscuous mode”, where one physical interface can be assigned multiple MAC addresses. “

iii) „If your application can work using a bridge (on a single Docker host) or overlay (to communicate across multiple Docker hosts), these solutions may be better in the long term. “

Darüber hinaus bietet Docker den Netzwerk-Treiber **Overlay** an. Da Container-Systeme meist verteilt arbeiten, ist es mit diesem Netzwerk-Treiber möglich, mehrere Container auf unterschiedlichen Host-Systemen miteinander zu verbinden. Um den Overlay-Netzwerk-Treiber nutzen zu können, ist es notwendig, einen Docker Swarm zu betreiben. Ein Docker Swarm besteht aus mehreren verteilten Docker-Hosts. Mindestens einer dieser Hosts übernimmt dabei die Rolle des Managers, während die anderen Hosts als sogenannte Yorker fungieren. Hat man einen Docker Swarm initialisiert und erzeugt nun ein Overlay-Netz, so wird dieses automatisch auf allen anderen Docker-Hosts bekannt gemacht. Auf diese Weise können Container auf unterschiedlichen Hosts miteinander in einer isolierten Netzwerkumgebung kommunizieren. Die Docker-Hosts sind über ein Bridge-Netzwerk mit dem Namen *gwbridge* verbunden. Dieses wiederum koppelt die Overlay-Netze mit dem physischen Netzwerk-Interface. Demnach verschleiern Overlay-Netzwerke nur den Informationsaustausch der Container über den Host-TCP-Port 2377 für die Cluster-Management-Kommunikation, den TCP und UDP-Port 7946 für die Kommunikation der einzelnen Docker-Hosts und den UDP-Port 4789 für den overlay Netzwerk Traffic.
Auf Docker Swarm wird in dieser Ausarbeitung nicht weiter eingegangen, da Docker selbst mehr und mehr auf Kubernetes setzt, worüber exemplarisch in Abschnitt 3.4.6 gesprochen wird. Darüber hinaus

findet eine genaue Überprüfung der vorgestellten vier Treiber in Kapitel 7 statt.
Zusätzlich zu den bisher vorgestellten Treibern besteht die Möglichkeit, Container gänzlich vom Netz zu entfernen. Dies macht vor allem dann Sinn, wenn man Container für Build-Prozesse nutzt, die vollkommen isoliert sein sollen. Hier ist allerdings zu beachten, dass alle notwendigen Pakete für den Build-Prozess bereits im Image des Containers enthalten sein müssen.
Gerade im Bereich der Bridge- und Overlay-Netzwerke gibt es die Möglichkeit, Third-Party Plug-ins in Docker einzubinden. Wie bereits oben erwähnt, beinhaltet die CNCF-Landschaft mehr als 20 Projekte im Kontext des Networkings. Einige dieser Projekte wie Open vSwitch [vSwitch, 2021] und Weave [Weaveworks, 2021] können direkt in Docker eingebunden werden. Zur Einbindung gibt es dabei verschiedene Optionen: Während Open vSwitch auf dem Docker Host installiert werden muss, stellt Weave ein offizielles Plug-in zur Verfügung, welches innerhalb eines Containers bereitgestellt wird und damit den Docker Host nicht direkt beeinflusst. Auf beide Plug-ins sowie auf das Networking in Docker allgemein wird in Kapitel 8 erneut eingegangen.

3.3.2.6 Monitoring und Logging

Docker stellt von Grund auf Mittel zum Monitoring und zum Logging von Containern und Applikationen zur Verfügung. Für das Monitoring gibt es zum einen die Möglichkeit, alle Informationen zu einem Container via CLI abzufragen. Die Daten werden dann über die in Absatz 3.3.2.7 dargestellte Engine API angefragt und ausgeliefert. Es können Informationen zur CPU- und RAM-Auslastung sowie zu Netzwerk-IO und Block-IO erhalten werden. Darüber hinaus können laufende Prozesse und Zustand der Container betrachtet sowie Informationen zum Aufbau der Container erfragt werden.
Zusätzlich gibt es in Docker ein Healthcheck-System, das Containern drei Zustände zuweisen kann: Starting, Healthy und Unhealthy. Entsprechend dieser Zustände kann jeder Container überprüft und im Zweifelsfall gestoppt oder neu gestartet werden. Die Angabe des Gesundheitszustands eines Containers ist stets an denjenigen Prozess innerhalb des Containers gebunden, der für die permanente Ausführung des Containers sorgt (Beispiel: *nginx -g 'daemon off'* im Falle eines Nginx Containers). Diese Bindung findet dabei in einer Dockerfile und damit bei Erzeugung des Images und nicht beim Starten des Containers statt. Um darüber hinaus die Applikationen innerhalb eines Containers zu loggen, stellt Docker ein komplexes Logging-System zur Verfügung, welches im Folgenden genauer beschrieben wird.
Docker stellt in der Basisversion 12 unterschiedliche Logging-Treiber zur Verfügung, die entweder bereits in Linux implementierte Systeme, wie journald oder syslog, zur Sammlung der Logs nutzen oder auf TCP-Dienste wie Fluentd oder AWSlogs zurückgreifen. Alle vorhandenen Treiber inklusive ihres Ausgabetyps werden in Tabelle 3 aufgelistet.
Die Logging-Treiber verarbeiten alle Ausgaben, die innerhalb des Containers auf die *stdout* und *stderr* Interfaces ausgegeben werden. Um die geschriebenen Logs einzusehen, kann das Kommando *docker*

Driver name	Output format	Additional information
json-file	writes logs in json format	catches everything that is written to stdout and stderr inside the container, **default driver**, available via cli command *docker logs*
local	stores logs in a custom format with minimal overhead	catches stdout and stderr, available via cli command *docker logs*
syslog	writes logs to the syslog facility of the docker host	can use UDP, TCP and TLS
journald	writes logs to journald of the host system	uses standard system journals, available via cli command *docker logs*
fluentd	writes logs to the data collector software fluentd in json format	uses TCP
gelf	writes logs in Graylog Extended Log Format	uses UDP
awslogs	writes logs to Amazon CloudWatch Logs	uses TCP and HTTPS
splunk	writes messages to splunk plattform in its own format using HTTP Event Collector	uses TCP, HTTPS and HTTP
etwlogs	writes log messages as Event Tracing for Windows (EWT) events	uses Windows Event Store
gcplogs	writes logs to GoogleCloud	uses TCP and HTTPS
logentries	writes logs to Rapid7 Logentries in its own format	uses TCP and HTTPS
none	no format	no logs are streamed

Tabelle 3: Übersicht über die Docker Logging Treiber [Docker Inc., 2021d]

logs [containername] verwendet werden. Das Format der Ausgaben kann zusätzlich über eine beim Starten des Containers mitgegebene Formatsbeschreibung angepasst werden. Die Komplexität des Systems kommt erst zum Vorschein, wenn man die unterschiedlichen Treiber im Speziellen betrachtet. Hierzu ein Beispiel:

Docker arbeitet per Default mit dem json-file Treiber. Dieser legt ohne weitere Konfiguration alle auf stdout und sterr geschriebenen Inhalte mit Zeitstempel in einer JSON-Datei ab. Wählt man nun als Alternative Fluentd [Fluentd Project, 2021], so muss zuerst ein Fluentd-Host auf dem Host-System installiert werden, der alle Nachrichten über einen vorher festgelegten Socket abfängt. Der große Vorteil hierbei besteht in der problemlosen Weiterleitung der Logs an MongoDB, Elasticsearch, AWS und viele mehr.

Zusätzlich zur notwendigen Konfiguration mancher Treiber kommen weitere Probleme hinzu. So können manche Treiber wie beispielsweise Syslog, Container am Starten, Stoppen und Deployen

hindern. Zudem können bei fehlender Verbindung zu Remote-Logging-Servern Logeinträge verloren gehen. Zusätzlich ist es bisher in der Community Version nur möglich, einen Logging-Treiber zu wählen, was die Nutzung mehrerer parallel laufender Logging Systemen erschwert. Aus diesem Grund gibt es zusätzliche Plug-ins, die die in Absatz 3.3.2.7 beschriebene Docker API zum Auslesen von Logs nutzen und diese dann über externe Systeme weiterleiten.

3.3.2.7 Docker Engine API

Die Docker Engine beinhaltet neben einer API-Schnittstelle auch die Docker CLI und ein System zur Kommunikation mit der Container Runtime. Im Kern handelt es sich bei der Engine daher um ein Client-Server System, welches sich um das Deployment und das Monitoring der Container kümmert. Standardmäßig findet die Kommunikation zwischen dem Terminal des Nutzers und einem lokalen Linux-Socket (*unix:///var/run/docker.sock*) statt. Dieser kann jedoch durch einen oder mehrere Remoteserver ersetzt werden. So ist es möglich, mehrere Docker-Installationen von einem Terminal zu verwalten. Da es sich bei den Kommunikationspartnern auf der einen Seite um die Docker Engine und auf der anderen Seite um die Container-Runtime containerd handelt (siehe Unterkapitel 3.2), findet die Kommunikation mittels Remote Procedure Calls (im Folgenden RPC) statt und genauer mit gRPC, einer von Google entwickelten Open Source Bibliothek für unterschiedliche Programmiersprachen.
Die über RPCs erhaltenen Informationen, kann die Docker Engine nun über die CLI sowie die API an den Nutzer zurückliefern. Beispielsweise gehören hierzu Informationen über den Ressourcenverbrauch der Container, die über die CLI mittels *docker stats* und über die API mithilfe von Http Requests durch den Nutzer bezogen werden können.
Die Engine API dient als standardisiertes Interface, um so externe Tools, wie exemplarisch Web-Apps, zur Überwachung und zum Deployment der Container zu nutzen. In Unterkapitel 6.2 wird vertieft darauf eingegangen, wie die Engine API für das Monitoring des Ressourcenverbrauchs verwendet werden kann.

3.4 Docker als Basis der modernen Cloud

Wie bereits zuvor erwähnt, waren mit Microsoft und Google zwei der größten Cloudanbieter der Welt durch Finanzierungen in die Entwicklung von Docker eingebunden. Betrachtet man die Cloud-Landschaft heute, verschwinden Container und Docker häufig hinter einer Vielzahl von Services und Angeboten, die einen Blick auf die eigentliche Basis kaum noch zulassen. Um die Wichtigkeit und den Einfluss von Containern und im Speziellen Docker auf die Cloud darstellen zu können, muss daher zuerst ein Überblick über die Cloud gegeben werden, angefangen bei Diensten die jeder Mensch täglich nutzt, bis hin zu den Services, die nur von Administratoren verwendet werden.

3.4.1 Die Cloud im täglichen Leben

Nahezu jeder Mensch sorgt täglich für den Start mehrerer Container. Wie bereits in Kapitel 1 angedeutet, reicht es dazu aus, ein Video über einen Streaming-Dienst anzuschauen. Vorreiter in der Verwendung von Microservice-Architekturen basierend auf Container im Streaming ist Netflix, die bereits seit Beginn der 2010er in Kooperation mit AWS an Container-Lösungen arbeiten und mit Titus eine mächtige Container Management Plattform entwickelt haben, welche mittlerweile open source zur Verfügung steht. Doch nicht nur beim Streaming sind Container im Einsatz. Unsere Smartphones greifen täglich auf eine Vielzahl von Services zu, die mit Hilfe von Containern und dazugehöriger Skalierung für eine stetige Erreichbarkeit dieser Services sorgen. Webseiten, die von Millionen Nutzern am Tag besucht werden, werden mit Hilfe von Containern ausgeliefert und wer im Internet spielt wird mit Hilfe von Containern mit einem geeigneten Gegner verbunden. Doch nicht nur in der Freizeit begleiten uns Container täglich. Wer Service-Software mit vorausgesetzter Internetanbindung nutzt, verbindet sich mit Containern, um seiner Arbeit nachgehen zu können und wer sich im Bereich der Kryptowährungen aufhält, welche mit Smart-Contracts arbeiten, wird ebenfalls unbewusst Container verwenden. Container durchziehen jeden Bereich, der mit dem Internet in Verbindung steht. Die Cloud ist mehr als der Remote-Datenspeicher, den die meisten Menschen unter ihr verstehen. Die Cloud ist ein Konstrukt, der eine Architektur zu Grunde liegt, die klassische Rechenzentren ersetzt und verschiedene Services für Endnutzer, Entwickler und Administratoren zur Verfügung stellt. Ihre Basis sind Container.

3.4.2 Dienste der Cloud

Wie bereits in Kapitel 1 angesprochen und in Abbildung 1 dargestellt, bietet die Cloud eine Vielzahl von Services an, die von Endnutzern verwendet werden. Endanwender, die weder Entwickler noch Administratoren sind, beschränken sich hier auf die Software as a Service. Bekannte Vertreter sind beispielsweise die Office-Pakete von Microsoft, die Produkte von Adobe oder das CRM Salesforce. Allen gemeinsam ist das Hosting der eigentlichen Software in der Cloud. Der Nutzer greift über den Browser oder eine entsprechende Desktop-Oberfläche auf die Module der Software zu und kann selbst von unterwegs an seinen Daten arbeiten. Oftmals wird der Service pro Monat, pro Nutzer oder pro Arbeitseinheit bezahlt. SAAS ersetzt somit klassische Software mit Lizenzmodell.

Die weiteren Bereiche der Cloud, angefangen bei Function as a Service bis hin zur Infrastructure as a Service, stehen nur Entwicklern und Administratoren zur Verfügung. Wie Abbildung 18 zeigt, ist das Angebot der Cloud-Dienste hierarchisch nach Umfang der Kontrolle durch den Nutzer aufgebaut. Während der Nutzer bei SAAS eine vorkonfigurierte Software erhält, die oftmals nur wenige Einstellungen erlaubt, kann er bei FAAS zumindest den auszuführenden Code selbst bestimmen. Dies eignet sich vor allem für zeitlich wiederkehrende oder getriggerte Aufgaben wie einen Newsletter-Service oder einen Watcher, der einen anderen Service kontrolliert. Der Nutzer kann zwar hier nicht über

die Art und Weise (z.B. welche Container-Konfiguration der Ausführung entscheiden, bezahlt im Gegenzug aber auch nur einen Bruchteil eines Cents für eine Ausführung. FAAS basiert dabei zu 100% auf Containern.

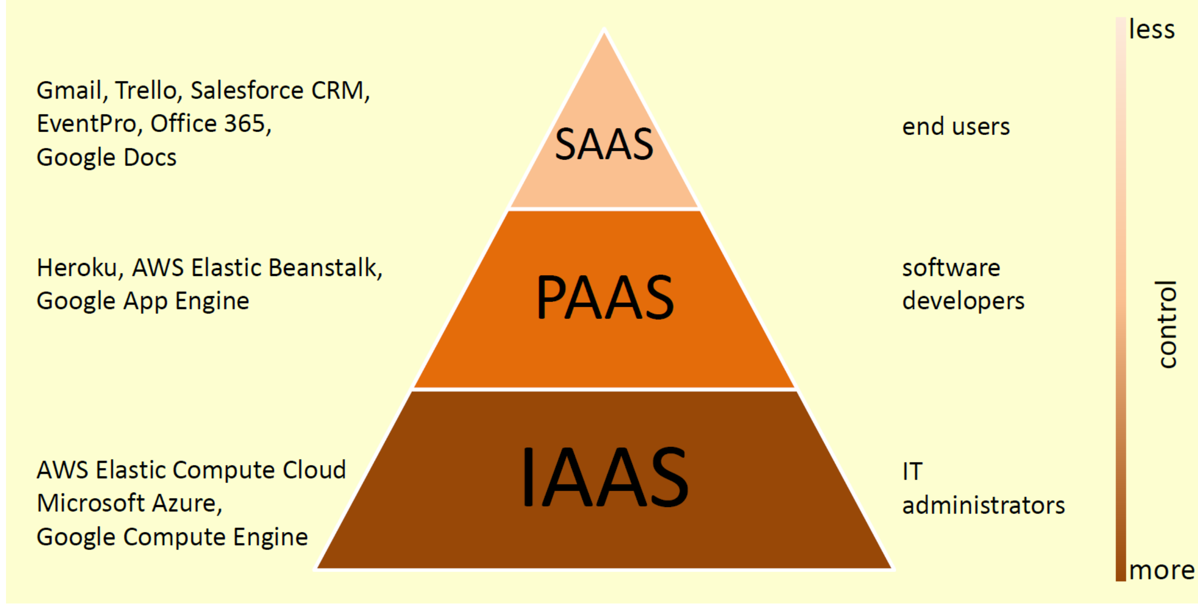

Abbildung 18: AWS Cloud Services und Beispiele (eigene Abbildung gemäß [Amazon, 2018])

PAAS erlaubt dem Nutzer schwer zu administrierende Software auf einfache Art und Weise zu verwenden, da die Administration der Server sowie der Netzwerkeinstellungen entfällt. Grundkonfigurationen sind der Software sind vorgegeben, können jedoch auch selbst getroffen werden. Beispiele für solche „hosted-Software“ sind GitLab oder Kubernetes, welches später noch genauer erläutert wird. Im Bereich von PAAS werden oftmals Container eingesetzt. Einige Software setzt jedoch eine eigene VM voraus.

Betrachtet man darüberhinausgehende Services, wie beispielsweise WebApps, SQL-Services, Storage oder Network-Service, sind diese eine Anpassung des Container as a Service. In allen Fällen wird ein Container mit darin befindlicher Software gestartet. Während man beispielsweise bei der Webapp schon einen angepassten Webserver mit entsprechenden Einstellungsmöglichkeiten erhält, eignet sich ein Container as a Service, um genau einen solchen einrichten zu können. Allen Services dieser Art ist gemein, dass sich zwar der Container einstellen lässt, die Container-Runtime jedoch ist nicht anpassbar.

Die in der Hierarchie der Cloud Dienste am niedrigsten eingestufte Infrastructure as a Service ist der einzige Dienst, der nicht grundlegend auf Container aufbaut. Dem Administratoren sind hier keine Grenzen gesetzt. Er erhält eine oder mehrere VMs und falls gewünscht sogar einen Bare-Metal-Server. Alle getroffenen Einstellungen sind somit am Ende auf den Administratoren und nicht auf den Cloud-Anbieter zurückzuführen. Es ist zu beachten, dass auch VMs die Grundlage für FAAS,

PAAS oder CAAS sind. Diese VMs sind jedoch vom Cloud-Anbieter gemanaged und müssen nur durch den Endnutzer gebucht werden. Doch auch im Bereich des IAAS finden sich oft Container wieder. Enterprise-Lösungen werden häufig mit IAAS umgesetzt, verwenden jedoch trotzdem Container. Um größtmögliche Flexibilität in der Entwicklung zu haben, administrieren große Unternehmen ihre Container-Cluster daher oftmals selber. Zeitgleich werden durch die Nutzung der Cloud-Dienste aber Server vor Ort eingespart.
Mit Ausnahme von IAAS unterliegen alle Dienste der Cloud Native Architecture, die wie bereits in Kapitel 1 erläutert, modularisierte Software, bestehend aus kleinen, autonomen Diensten, die über APIs miteinander kommunizieren, voraussetzt. Aus diesem Grund wird der infrastrukturelle Aufbau, auf dem solche Software ausgeliefert wird, Cloud Native Stack genannt.
Wie ein CAAS-Service in der Cloud von Microsoft angelegt werden kann, wird exemplarisch in Abschnitt 3.6.2 dargestellt.

3.4.3 Docker Lifecycle

Für jeden Dienst, der auf Containern basiert, spielt der Container Lifecycle eine wichtige Rolle. Durch seine Simplizität und Geschwindigkeit bestimmt er den immer schnelleren Update-Zyklus von Software.
Der Docker Lebenszyklus, der die Blaupause für einen Container Lifecycle darstellt, wird in Abbildung 19 aufgezeigt. Er besteht letztlich aus der Erzeugung beziehungsweise dem Herunterladen eines Images sowie dem Bauen, Starten, Stoppen und Löschen eines Containers. Wie [sysdig.com, 2021] zeigt, ist ein Durchlauf des hinteren Teils des Lebenszyklus (Bauen, Starten, Stoppen, Löschen) in einem Viertel der Fälle nach weniger als zehn Minuten beendet. Images hingegen werden in zwei Drittel der Fälle nach einer Woche erneuert. Die Geschwindigkeit in der Erneuerung der Images profitiert stark von den durch Docker eingeführten Dockerfiles, die schon in Absatz 3.3.2.2 genauer beschrieben wurden.
Allgemein gibt es im Docker Lifecycle mehrere Angriffspunkte für die Erzeugung eines „grünen“ Docker Lebenszyklus, die jedoch in der gängigen Literatur bezogen auf Docker aber auch in der Forschung kaum Beachtung finden. Dabei bietet, wie im Verlauf der Ausarbeitung gezeigt wird, jeder Bestandteil des Docker Lebenszyklus Potential für die Verbesserung der Energieeffizienz. Neben der Untersuchung der Bestandteile von Docker, die zuvor dargestellt werden, wird daher in Kapitel 7 auch der Lebenszyklus untersucht.

3.4.4 Open Container Initiative

Wie bereits zuvor erwähnt, profitiert der Container Lifecycle stark von den Dockerfiles, mit denen ein einfaches Container-Format beschrieben ist. Aus diesem Grund kümmert sich die Open Container Initiative (OCI) um die Einführungen eines Runtime- beziehungsweise Image-Standards. Die OCI

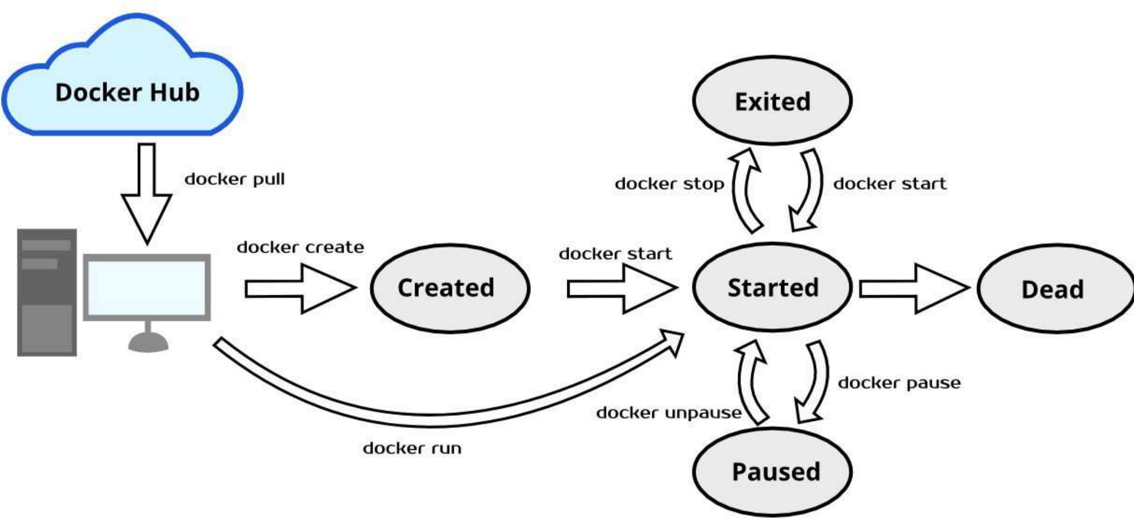

Abbildung 19: Docker Lifecycle [Docker Inc., 2019c]

wurde 2015 von einer Gruppe von Konzernen aus dem Container-Umfeld gegründet. Zu diesen Unternehmen zählten Docker und CoreOs, die zu diesem Zeitpunkt zwei Konkurrenzprodukte, nämlich Docker und rkt, auf dem Markt hatten. Heute gehören zu ihr Unternehmen wie Cisco, Amazon und Google. Die OCI beschreibt sich auf ihrer Webseite selbst wie folgt:

> „An open governance structure for the express purpose of creating open industry standards around container formats and runtime“ [Open Container Initiative, 2016]

Der durch die OCI gegebenen einheitlichen Spezifikation für Container Runtimes und Images entsprechen neben Docker ebenfalls andere Container-Systeme wie zum Beispiel *kata*. Wie bereits zuvor erwähnt, spendete Docker seine Runtime runC an die OCI, um die Basis für eine Spezifikation zu legen. Aktuell liegt der Fokus der Initiative auf der Weiterentwicklung von runC und der Reduzierung von Container Registries. Im Bereich der Container Cluster und allgemein des Cloud Native Stacks, kümmert sich die Cloud Native Computing Foundation um eine Standardisierung.

3.4.5 Cloud Native Computing Foundation

Wie bereits in Unterkapitel 1.1 erwähnt, vereint die Cloud Native Computing Foundation mehr als 1200 Unternehmen und Projekte, mit dem Ziel, die Cloud in den verschiedensten Bereichen weiter zu entwickeln. Dazu zählen neben Container-Runtimes auch Cloud Apps (zum Beispiel GitLab), Storage-, Netzwerk- und Monitoring-Lösungen. Kernbestandteil der CNCF ist jedoch Kubernetes,

das als eigenständiges Vorhaben von Google übernommen wurde. Als führende Plattform im Bereich der Container-Orchestrierung (vergleiche beispielsweise [Degenmann, 2021]) vereint Kubernetes die oben genannten Lösungsansätze in einer Plattform und bietet so den Projekten der CNCF eine Basis. In gleicher Weise können beispielsweise Distributed Storage Systeme (wie zum Beispiel Ceph) oder Netzwerk-Treiber (wie OvS) direkt in Kubernetes eingebunden werden.
Zur CNCF gehört neben Kubernetes auch containerd, das von Docker übernommen wurde. Sowohl Kubernetes als auch containerd gelten innerhalb der CNCF als „graduated projects" und bilden daher die Aushängeschilder der CNCF. Als zugrunde liegendes System von Docker formt containerd somit die Verbindung zwischen Docker und der CNCF. Weiter wird containerd neben der hauseigenen Runtime cri-o standardmäßig als Runtime in Kubernetes verwendet. Die in dieser Arbeit besprochene Thematik hat somit indirekten Einfluss auf Kubernetes und damit auf alle Container, die innerhalb von Clustern laufen, welche mit Kubernetes orchestriert werden und Docker oder containerd als Runtime verwenden.
Die CNCF hat zur Vereinheitlichung, das Kubernetes CRI (Container Runtime Interface) entwickelt. Alle Container-Runtimes, die in der Lage sind mit dem CRI zu kommunizieren, können ebenfalls zur Bereitstellung von Containern in Kubernetes verwendet werden.
Die CNCF beschäftigt sich aktuell verstärkt mit dem Marketing von Kubernetes, entwickelt eine eigene Container-Runtime (cri-o) und sucht nach weiteren Projekten zur Weiterentwicklung von Kubernetes.

3.4.6 Container-Cluster-Software Kubernetes

Container werden, wie zuvor in Absatz 3.3.1.5 beschrieben, üblicherweise in großer Anzahl auf Container-Clustern ausgeführt. Zur Verwaltung und zum Monitoring wird dafür Software wie Kubernetes der CNCF verwendet. Als eines der meist verwendeten Tools macht es daher Sinn, nachfolgend oberflächlich die Funktionsweisen und Besonderheiten von Kubernetes sowie dessen Zusammenhang mit Docker darzustellen. Darüber hinaus soll im weiteren Verlauf der Einfluss des Stromverbrauchs von Docker auf Kubernetes erläutert werden.

3.4.6.1 Funktionsweise von Kubernetes

Wie ebenfalls in Absatz 3.3.1.5 angedeutet, fasst Kubernetes zusammengehörige Container in Pods zusammen, in welchen zusammengehörige Komponenten, wie beispielsweise ein gemeinsames Netzwerk, geteilt werden. Dadurch vereinfacht sich auf der einen Seite die Kommunikation innerhalb des Pods und auf der anderen Seite stellt der Pod eine einfacher zu verwaltende Abstraktion vieler Container dar. Insgesamt kann K8 selbst als eine Abstraktion von Docker verstanden werden. So wird zum Beispiel eine Web-UI zur Verfügung gestellt, die alle Informationen grafisch aufbereitet und dem Nutzer den Zugriff vereinfacht. Kubernetes wird dabei nicht mit einer perfekten Konfiguration ausgeliefert, sondern muss im Falle des eigenen Hostings in einem aufwändigen Prozess eingerich-

tet werden. Einzelne Bestandteile wie Monitoring, Networking, Runtime oder Storage müssen durch Third-Party-Plugins erweitert werden. Demnach ist K8 auch keine eigene Container-Runtime und greift standardmäßig auf Docker zurück. Alternativ können allerdings auch andere Runtimes, wie exemplarisch containerd (also Docker ohne Docker Engine) oder cri-o, verwendet werden, über welche der Administrator selbst entscheiden kann. Einzige Vorgabe ist eine zum K8 Container Runtime Interface kompatible Kommunikation. K8 gibt hier keine Richtlinie vor, nennt jedoch in seiner Dokumentation zuerst containerd, gefolgt von Docker und der hauseigenen Runtime cri-o (vergleiche [Kubernetes, 2021c]).
Hauptaufgabe von K8 ist es nun, die reibungslose Erzeugung, den Start als auch Ausführung der Pods als logische Einheit zu gewährleisten und im Falle eines Fehlers, schnellstmöglich Gegenmaßnahmen einzuleiten. Erzeugung und Start spielen hierbei eine ebenso wichtige Rolle wie die Ausführung, da die durchschnittliche Lebensdauer eines Containers oder Pods bei circa einem halben Tag liegt [sysdig.com, 2021]. Ein Grund hierfür ist die hohe Updategeschwindigkeit von Images. Circa 70% aller Images werden jede Woche eines Updates unterzogen [datadoghq.com, 2018]. Um den reibungslosen Ablauf zu gewährleisten, nutzt K8 ein zu Docker ähnliches Lifecycle Status System. Auch kümmert sich K8 um ein automatisiertes Load-Balancing zwischen den Replikas eines Services. Replikas werden dabei entweder manuell beim Start eines Services oder während der Laufzeit erstellt. Zusätzlich dazu können Replikas automatisiert mit Hilfe des in K8 integrierten Autoscalers erzeugt werden. Hierbei sind drei Scaler zu unterscheiden:

i) **Cluster Autoscaler [Kubernetes, 2021f]**
Der Cluster Autoscaler sorgt dafür, dass Pods von K8 Nodes auf andere Nodes verschoben werden, wenn die Ressourcen eines Nodes ausgeschöpft sind oder diese aufgrund fehlender Auslastung konsolidiert werden können. Demnach ist es möglich, dass zwei Replikas auf unterschiedlichen Maschinen laufen können. Ziel ist die optimale Ausschöpfung der vorhandenen Ressourcen. Man spricht in diesem Fall auch von *Kubernetes Elasticity*.

ii) **Vertical Pod Autoscaler [Kubernetes, 2021g]**
Im Gegensatz zum Cluster Autoscaler reagiert der Vertical Pod Autoscaler nicht auf die Ressourcenauslastung der Nodes, sondern auf die Auslastung der Pods. Benötigt ein Pod weniger oder mehr Ressourcen, werden durch vertikale Skalierung Ressourcen des Pods entfernt oder hinzugefügt.

iii) **Horizontal Pod Autoscaler [Kubernetes, 2021d]**
Bei der horizontalen Skalierung steht die Leistung des Services im Vordergrund. Kann nun ein Service eine hohe Anzahl von Zugriffen nicht verarbeiten, so wird eine Replika des Services erzeugt. Auf beide Replikas wirkt dann wiederum das automatisierte Load-Balancing. Um auf die Leistungsengpässe eines Services zu reagieren, kann auf verschiedene Metriken zurückgegriffen werden. Hierzu zählen die CPU- und RAM-Auslastung des Services, Network I/O, die Zugriffe

auf den Service pro Sekunde oder Pakete pro Sekunde. Auch selbst erstellte Metriken, basierend auf externen Messwerten (wie etwa der Energieverbrauch), sind nutzbar. Der Punkt der Skalierung (zum Beispiel: Skalierung bei 50% CPU-Auslastung) wird durch den Nutzer gewählt. Diese Werte basieren dabei auf Erfahrungswerten der Administratoren oder der Entwickler des Services.

Kubernetes umfasst noch weitere Funktionalitäten in verschiedenen Bereichen. Dazu zählen unter anderem Monitoring, Storage und Networking sowie die Möglichkeit von Rolling Updates der Replikas eines Services. Außerdem existiert unter dem Namen Helm ein eigenes Tool zur Bereitstellung ganzheitlicher Service/Kubernetes-ready Apps, (z.B. Gitlab inklusive Datenbank) die darüber hinaus in einer Art Bibliothek angeboten werden [Helm, 2021].

3.4.6.2 Kubernetes im Kontext des Cloud Native Stack

Kubernetes und auch andere Orchestrierer befinden sich im Cloud Native Stack, wie in Abbildung 20 dargestellt, oberhalb der VMs. Auf jeder VM ist dabei, wie weiter oben beschrieben, der Client des Orchestrierungstools installiert. Eine VM übernimmt die Rolle des Masternodes und ist für das Ausrollen und Skalieren der Pods zuständig. Um VMs zu skalieren, muss der Orchestrierer mit dem Hypervisor kommunizieren. Je nach Auslastung können so VMs ab- oder angeschaltet werden. Verwaltet man

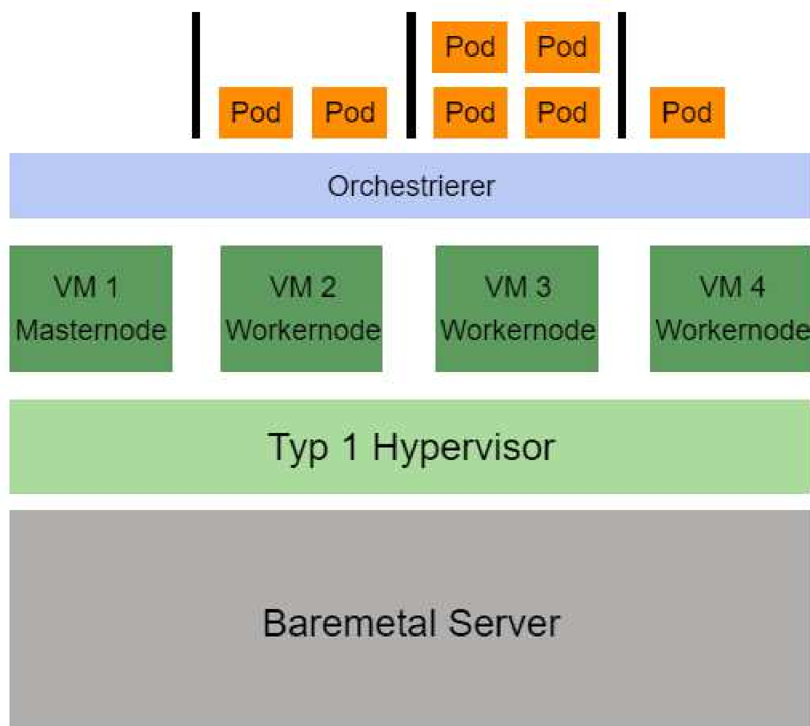

Abbildung 20: Orchestrierer im Cloud Native Stack (eigene Abbildung)

eine Kubernetes-Installation selbst, sind Skalierer, Deployment und Kommunikation sowie Storage

und Security selbst zu verwalten. Nutzt man Dienste in der Cloud, wie CAAS oder PAAS, sind diese bereits vorkonfiguriert. Eigens verwaltete Kubernetes-Installationen sind daher eher im Enterprise-Bereich zu finden. Die Komplexität mancher Enterprise-Lösungen verlangt die Flexibilität der eigenen Konfiguration des Orchestrierers. Wie die Erzeugung einer Web-App in der Cloud mit vordefiniertem Skalierer aussieht, wird in Abschnitt 3.6.2 erläutert.

3.4.7 Chancen und Risiken von Docker als Basis der modernen Cloud

Docker als Basis der modernen Cloud ermöglicht eine Vielzahl neuer Dienste, die es Entwicklern und Administratoren erlauben, schneller zu entwickeln, auszuliefern und zu skalieren. Grund hierfür ist in erster Linie der Docker beziehungsweise Container Lifecycle. Aufgrund der Standardisierung von Runtime, Paketierung und Orchestrierung durch die OCI und die CNCF wird diese Entwicklung auch weiter voran getrieben. Opfer der schnelleren und einfacheren Cloud-Services sind Möglichkeiten zur individuellen Konfiguration. Der Kunde muss sich somit entscheiden, ob er auf die Vorteile der Dienste zu Gunsten erweiterter Konfigurationsmöglichkeiten verzichten möchte. Somit muss der Kunde beispielsweise auf die Energieeffizienz der genutzten Dienste vertrauen, da diese nur geringfügig durch geeignete Konfiguration anpassbar ist. Im nachfolgenden Kapitel wird daher erläutert werden, welche Aussagen es bezüglich des Energieverbrauchs von Docker gibt und welchen Einfluss dieser auf den Energieverbrauch von Orchestrierungstools hat.

3.5 Energieverbrauch von Docker und des Cloud Native Stacks

Wie im letzten Kapitel deutlich werden sollte, besteht der Cloud Native Stack oberhalb der VMs aus Container-Runtime und Container-Cluster. Demnach ist der Stromverbrauch des Cloud Nativ Stacks, neben der eigentlich Software, die im Container läuft, abhängig von diesen Komponenten. Über den Stromverbrauch von beispielsweise Docker als de facto Standard gibt es allerdings keine Aussage.

3.5.1 Energieverbrauch von Docker

Generell wird in der Dokumentation von Docker keine Aussage zum Energieverbrauch getätigt. Eine allgemeine Aussage diesbezüglich wäre auch kaum möglich, da, wie im weiteren Verlauf der Arbeit gezeigt wird, die Software, die im Container angewendet wird, ebenfalls einen Einfluss auf den Energieverbrauch eines solchen hat. Eine geeignete Konfiguration bezogen auf einen Anwendungsfall kann jedoch großen Einfluss auf den Energieverbrauch haben. Je nach den verwendeten Komponenten der Containerumgebung kann sich der Gesamtverbrauch des Containers verändern. Kapitel 7 wird darstellen, wie sich der Energieverbrauch von Containern in evaluierten Anwendungsszenarien mit der Anwendung unterschiedlicher Systemkomponenten äußert.

3.5.2 Auswirkungen von Docker auf Orchestrierer

Um einen Cluster zu initialisieren, muss der Orchestrierer inklusive der ausgewählten Container-Runtime auf jedem Node installiert sein. Bei den Worker-Nodes, auf denen die Pods laufen, wird hier jedoch nur eine API-Schnittstelle zum Master-Node, über welche die Administration aller Container geregelt wird, hergestellt. Die Schnittstelle greift dabei über ein Interface, im Falle von Kubernetes das CRI, auf die entsprechenden Container zu und leitet alle weiteren Information zum Master-Node. Im Falle von Kubernetes geschieht dies durch eine überschaubare Anzahl an Prozessen (kubeadm, kubectl, kubelet), deren CPU- und RAM-Auslastung im Rauschen der Container verschwindet. Demnach ist die Container-Runtime und die in den Pods angesiedelte Software ausschlaggebend für den Energieverbrauch des Container-Clusters. Verwendet man demnach, wie bereits in Abschnitt 3.4.5 erwähnt, Docker, so entscheiden dessen Systeme über den durch die Software induzierten Energieverbrauch des Clusters. Dies gilt natürlich ebenfalls für die Orchestrierer, die bei CAAS, FAAS oder PAAS verwendet werden.

3.6 Einsatzszenarien und Praxisbeispiele für die Nutzung von Containern und Container-Clustern

Um aufzuzeigen, wie Container in der Praxis verwendet werden, werden im Folgenden Einsatzszenarien dargelegt, die zum einen mit gängiger Literatur zum Thema Container und Docker, wie zum Beispiel [Miell and Hobson Sayers, 2019] und [Öggl and Kofler, 2018], übereinstimmen und zum anderen aus einer eigens erstellten Umfrage hervorgehen, die auf den Konferenzen *OpenStack Tage 2019* [B1 Systems Gmbh, 2021] und *ContainerConf 2019* [heise Verlag, 2021] durchgeführt wurde. Beide Konferenzen richten sich an Praktiker und werden sowohl von Entwicklern als auch von Administratoren besucht. Weiteres zum Aufbau und den Ergebnissen der Befragung ist in Kapitel 5 zu finden. Zusätzlich zu den Einsatzszenarien wird nachfolgend anhand zweier Beispiele erläutert, wie Container in der Cloud und in der Software-Entwicklung eingesetzt werden können.

3.6.1 Typische Einsatzszenarien für Container

Im Falle der Container-Software sprechen unter anderem [Agarwal, 2017] und [Suleman, 2019] von acht übergeordneten Use Cases (siehe Abbildung Abbildung 21), wobei diese in drei Hauptgebiete unterteilt werden können: Konsolidierung, Auslieferung und Entwicklung.
Zur Konsolidierung zählt zum einen die reine Serverkonsolidierung, die mit der besseren Auslastung der Hardwareressourcen einhergeht und zum anderen die Auflösung von Maschinen, die veraltete Treiber oder Betriebssysteme für Legacy-Apps bereitstellen. Zusätzlich vereinfachen Container die gemeinsame Nutzung einer Plattform durch mehrere Nutzer (Multi-Tenancy). Zur Auslieferung zählen das Rapid Deployment sowie das Code Pipeline Management, welche gerne unter den Stich-

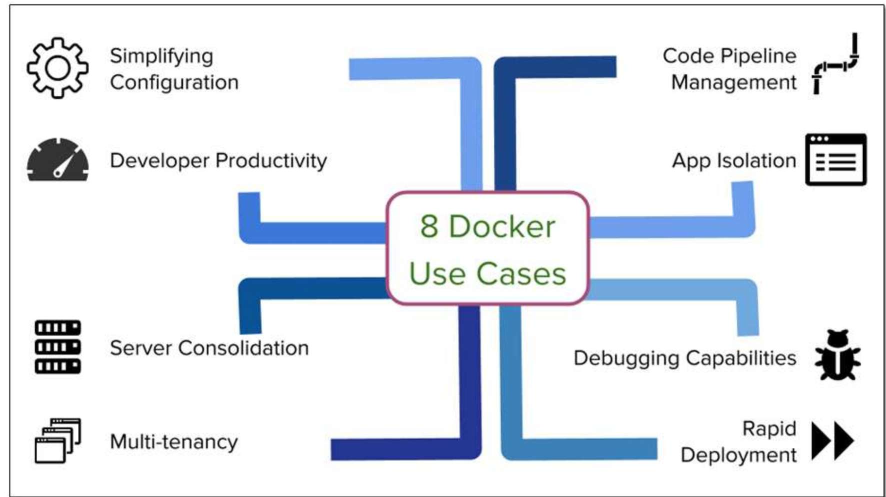

Abbildung 21: Acht Use Cases für Docker [Agarwal, 2017]

worten CI und CD zusammengefasst werden.
Im Bereich der Entwicklung entstehen Vorteile durch die vereinfachte Konfiguration von Containern im Vergleich zu VMs. Darüber hinaus erzeugen Container eine automatische Isolierung der Applikationen innerhalb der vom Container bereitgestellten Sandbox, geben Möglichkeiten für bessere Integrationstest (siehe zum Beispiel [Wittek, 2019]) und erhöhen resultierend daraus die Entwicklungsproduktivität.
Im Folgenden sollen ein Beispiel des CAAS sowie ein kurzes konkretes Fallbeispiel aus der Praxis zeigen, wie sich die Anwendung von Containern im Gebiet von CI und CD sowie in der Softwareentwicklung auswirken.

3.6.2 Praxisbeispiel I: Eine Webapp in Azure

Um in Microsoft Azure eine Webapp anzulegen, bedarf es nur weniger Schritte und Einstellungen. Eine Webapp besteht dabei aus Webserver, Datenbank und Software. Wie in Abbildung 22 zu erkennen, bietet Azure eine große Palette unterschiedlicher Dienste aus verschiedensten Kategorien.
Die Webapp ist dabei nur einer von vielen Diensten, der auf Containern basiert. Erzeugt man eine Webapp, muss man zuerst, wie in Abbildung 23 zu erkennen, eine sogenannte Ressourcengruppe und einen App-Service-Plan wählen. Basis eines App-Service-Plans ist eine VM, die jedoch wie in der Cloud zu erwarten, nicht selbst administriert werden muss. Hier wählt man nur das Betriebssystem und die Ausstattung der VM bezüglich CPU, RAM und Speicher. Im Falle von Azure, bieten VMs auf Windows-Basis mehr Möglichkeiten als Linux-VMs. Microsoft ist hier jedoch am nachrüsten. Auf einer VM können mehrere Webapps und andere Dienste ausgeführt werden. Weiter steht die Wahl zwischen Docker-Container und Code. Im Falle des Codes hat der Nutzer die Möglichkeit zwischen verschiedenen Software-Stacks zu wählen. Hier können beispielsweise NodeJS, PHP oder ähnliche ausgewählt werden. Der Code wird im Nachgang über ein Repository in den gestarteten Container

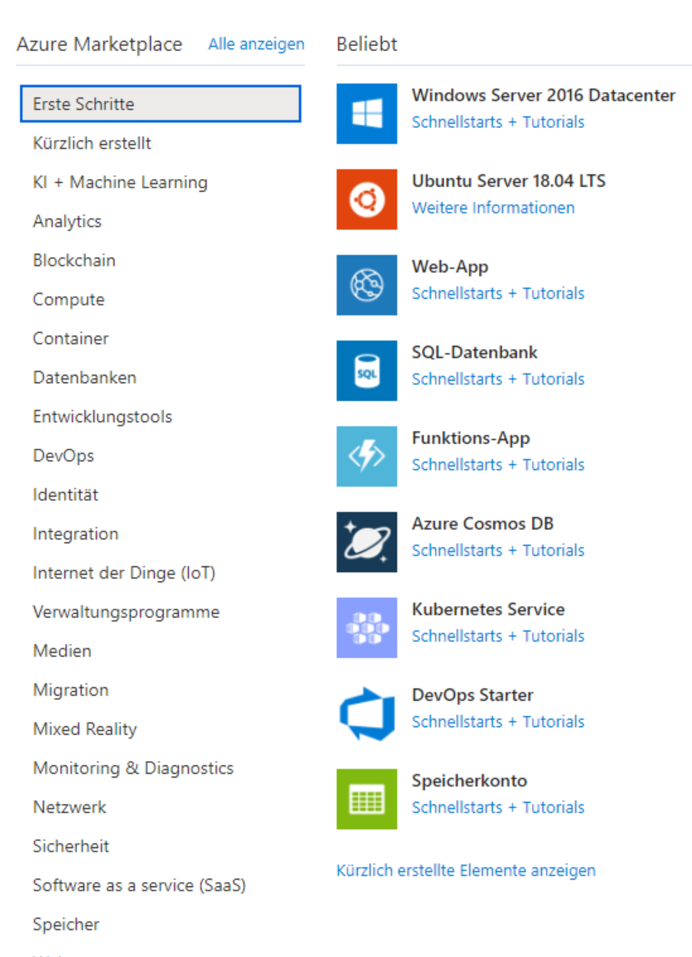

Abbildung 22: Azure stellt eine Vielzahl an Services aus unterschiedlichen Kategorien (eigene Abbildung)

geladen.
Wählt der Nutzer den Docker-Container, kann er hier entweder einen vorgefertigten, offiziellen NGINX-Container starten oder einen Container aus einer Container-Registry herunterladen und starten (vergleiche Abbildung 24).
Zusätzlich kann der Nutzer hier eine Docker-Compose Datei laden und den oder die Container hierüber starten (vergleiche Abbildung 25.
Ist die Webapp konfiguriert, können im Nachgang noch weitere Einstellungen getroffen werden. Neben Netzwerk- und Sicherheitseinstellungen, können auch Datensicherungen und Automation konfiguriert werden. Darüber hinaus ist es ebenfalls möglich die App automatisiert oder manuell zu skalieren. Azure betreibt hier für jeden App-Service-Plan einen eigenen Skalierer. Selbst ohne beispielsweise Kubernetes ist es somit möglich, mit Hilfe von unterschiedlichen Metriken, wie beispielsweise die CPU-Auslastung oder die Antwortzeit des Dienstes, die Containerinstanzen der Webapp zu skalieren. Die Lastbalancierung zwischen diesen Instanzen wird ebenfalls automatisiert übernommen und muss

Web-App erstellen

Projektdetails

Wählen Sie ein Abonnement aus, um bereitgestellte Ressourcen und Kosten zu verwalten. Verwenden Sie Ressourcengruppen wie z. B. Ordner zum Organisieren und Verwalten all Ihrer Ressourcen.

Abonnement * az-sub

Ressourcengruppe * rg-development

Neues Element erstellen

Instanzendetails

Name * dockerenergy

.azurewebsites.net

Veröffentlichen * Code Docker-Container

Betriebssystem * Linux Windows

Region * Germany West Central

Sie finden ihren App Service-Plan nicht? Versuchen Sie es mit einer anderen Region.

App Service-Plan

Der App Service-Tarifplan bestimmt Standort, Features, Kosten und Computeressourcen für Ihre App. Weitere Informationen

Linux-Plan (Germany West Central) * ASP- (neu)

Erstellen

SKU und Größe * Premium V2 P1v2

ACU gesamt: 210, 3.5 GB Arbeitsspeicher

Größe ändern

Abbildung 23: Azure erlaubt die direkte Auswahl von Docker-Container (eigene Abbildung)

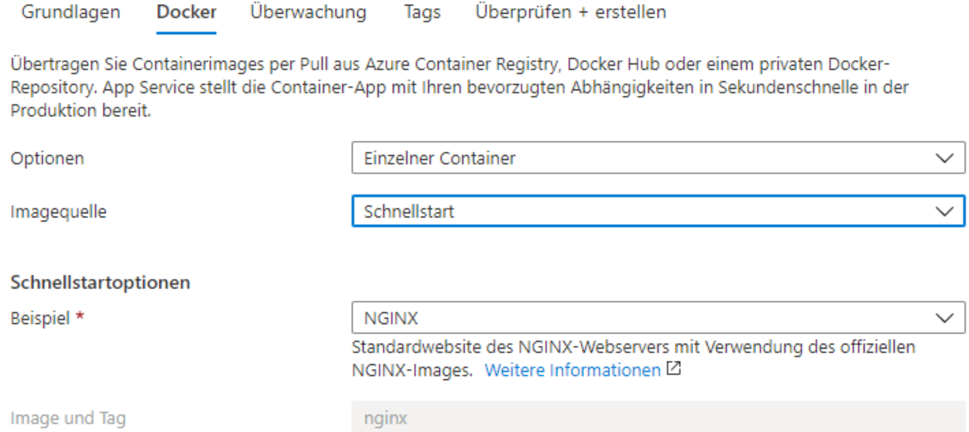

Abbildung 24: Es können Container aus unterschiedlichen Registries gewählt werden (eigene Abbildung)

nicht weiter konfiguriert werden.

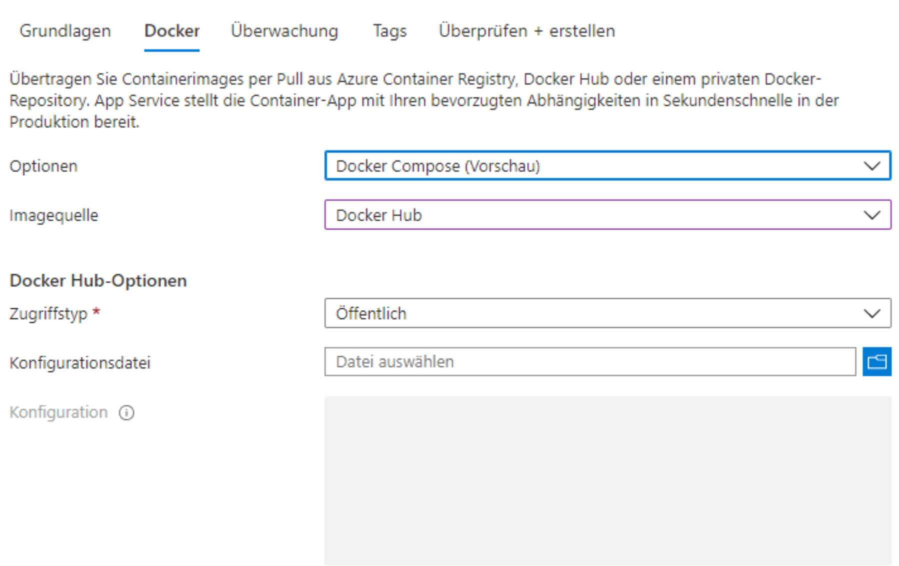

Abbildung 25: Auch Docker Compose Dateien können angegeben werden (eigene Abbildung)

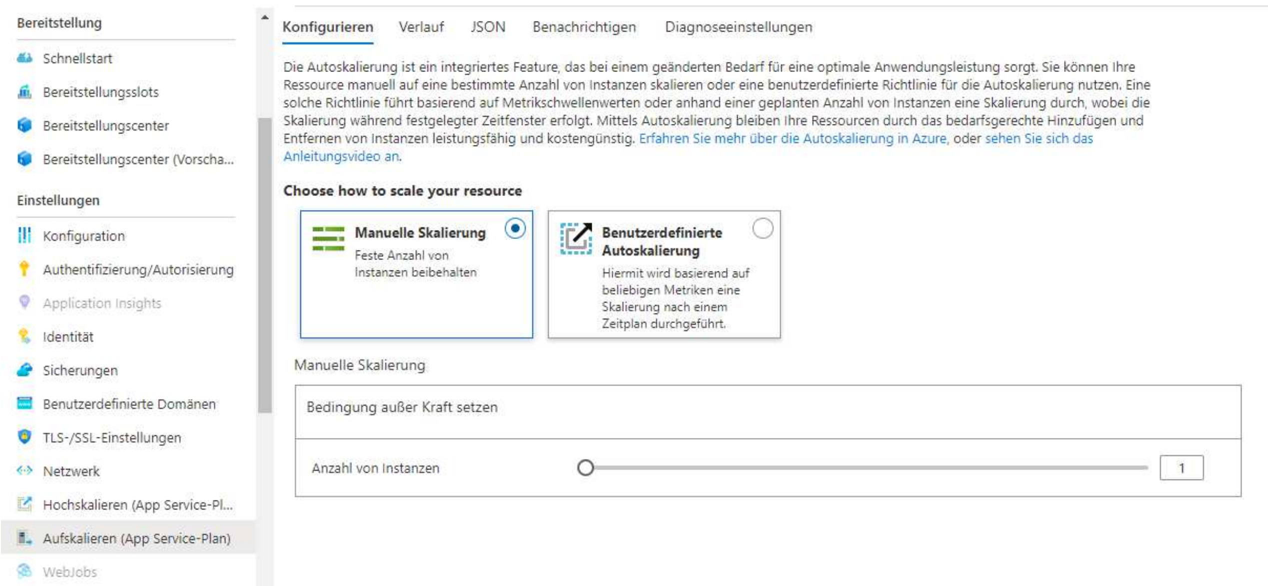

Abbildung 26: Die erstelle Webapp kann automatisch und manuell Skaliert werden (eigene Abbildung)

3.6.3 Praxisbeispiel II: Containereinsatz in der Software-Entwicklung

Container können nicht nur am Ende einer Software-Entwicklung zur Bereitstellung des fertigen Produkts genutzt werden. Vielmehr spielen sie innerhalb des Entwicklungsprozesses eine immer größere Rolle. Der Entwickler ist durch Einbindung von Container und VM-Systemen in die Entwickler-IDE in der Lage, sein Endprodukt in der finalen Umgebung zu testen. Diese Tests können den Auslieferungsprozess, das Bauen der Applikation und die Funktionalität der Software betreffen. Nachfolgend

wird kurz beschrieben, wie diese Tests im Detail aussehen und welche Folgen sie in Bezug auf den Energieverbrauch haben können.

3.6.3.1 Continuous Integration

In der modernen, agilen Softwareentwicklung führt kein Weg an Software-Versionisierung vorbei. Dabei unterstützt entsprechende Software (wie zum Beispiel Gitlab) Code-Pipelines, die nach jedem commit, merge, push oder pull ausgeführt werden können (siehe Abbildung 27. Code-Pipelines die-

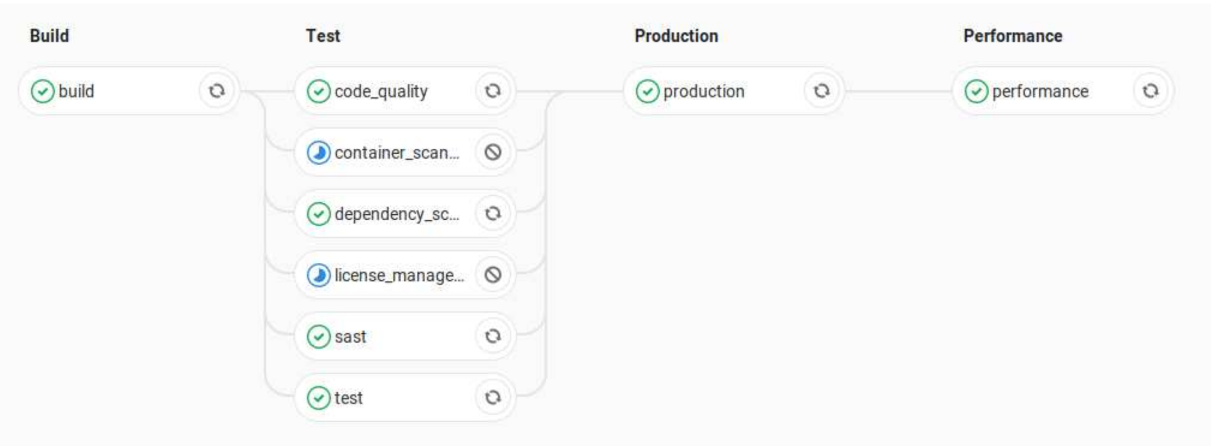

Abbildung 27: Gitlab CI Code Piplines [Gitlab, 2021]

nen dazu, nach jeder Änderung an der Code-Basis, automatisierte Tests durchzuführen. Container können hierbei genutzt werden, um den Build-Prozess und damit Fehler zu testen, um so die Integration fehlerhaften Codes zu vermeiden. Darüber hinaus lässt sich auch der Bau eines Containers zur Auslieferung der Software simulieren. Eine Code-Pipeline kann hier über eine Konfigurationsdatei, wie beispielsweise in Abbildung 28 dargestellt, festgelegt werden.

```
performance:
  stage: performance
  image: docker:git
  variables:
    URL: https://example.com
  services:
    - docker:stable-dind
  script:
    - mkdir gitlab-exporter
    - wget -O ./gitlab-exporter/index.js https://gitlab.com/gitlab-org/gl-performance/raw/master/index.js
    - mkdir sitespeed-results
    - docker run --shm-size=1g --rm -v "$(pwd)":/sitespeed.io sitespeedio/sitespeed.io:6.3.1 --plugins.add
      ./gitlab-exporter --outputFolder sitespeed-results $URL
    - mv sitespeed-results/data/performance.json performance.json
  artifacts:
    paths:
      - sitespeed-results/
    reports:
      performance: performance.json
```

Abbildung 28: Gitlab CI Code Piplines Konfiguration [Gitlab, 2021]

Weiter können Container im Bereich von Continuous Integration für reine Integrationstests genutzt werden. Hierfür gibt es für eine Vielzahl von Programmiersprachen vorgefertigte Bibliotheken (siehe dazu [Wittek, 2019]).

3.6.3.2 Continuous Deployment

Wie im vergangenen Abschnitt beschrieben, können Code-Pipelines die Integration neuer Features mit Test unterstützen, die durch Container ausgeführt werden. Auch kann die automatisierte Auslieferung von Softwareupdates durch Container gefördert werden. Werden neue Features in den Master-Branch der Software-Versionisierung gemerged, findet eine automatisierte Auslieferung der Software in Containern statt. Dabei wird jede vorhandene Infrastruktur automatisch mit den Containern beliefert, wobei der Auslieferungsprozess aufgrund der Plattformunabhängigkeit von Containern einmalig zentral hinterlegt werden kann. So kann ein deutlich höherer Update-Zyklus von Software erreicht werden, der heutzutage besonders im Bereich mobiler Anwendungen erkennbar ist.

3.6.3.3 Folgen für den Energieverbrauch von Containersystemen

Im Bereich von CI kann es in Großprojekten zu mehreren hundert Commits pro Tag kommen. Dies führt zu einer enormen Anzahl an kurzzeitig gestarteten Containern. Da der Build-Prozess von Containersystemen je nach Konfiguration, wie sich im Verlauf der Arbeit zeigen wird, erhebliche Effizienzlücken aufweisen kann, wird hier ein großes Potenzial an Energieersparnis verschenkt. Hierzu zählen natürlich auch automatisierte Auslieferungen im Bereich des CD, die in den meisten Fällen in Nightly Builds, also nachts während geringer Lastspitzen, vollzogen werden.

3.7 Zwischenfazit

Im Verlauf des letzten Kapitels wurde gezeigt, welchen Einfluss Docker auf die Entwicklung der modernen Cloud hatte und wie Docker heute in Form von Cloud-Services verwendet wird. Des Weiteren wurde gezeigt, wie Containersysteme funktionieren und wie sich dies im Falle von Docker äußert. Dabei wurde im Besonderen Wert auf die Darstellung der wichtigsten Features von Docker, wie beispielsweise die Dockerfile und das Build-System als absolute Neuheiten in der Containerwelt, gelegt. Hierbei wurde die hohe Komplexität der Software Docker deutlich, die eine Vielzahl von Angriffspunkten bezüglich einer energieeffizienten Konfiguration bietet. Wie ebenfalls gezeigt wurde, ist dies von großer Wichtigkeit, da die Software durch ihren geringen Overhead zwar zur Verbesserung der Ressourcenauslastung beiträgt, jedoch jeden Tag mehr und mehr Container ausgeliefert werden, die nahezu jeden Bereich der Cloud durchziehen. Verdeutlicht wurde dies durch die beispielhafte Darstellung verschiedenster Dienste, die auf Containern basieren. Hierbei wurde zudem klar, dass der Nutzer durch die Verwendung von Cloud-Services auf die Transparenz bezüglich einer energieeffizienten Konfiguration der verwendeten Container verzichtet. Weiter wurde präsentiert, dass es zum aktuellen

Zeitpunkt keinerlei Aussage über den Energieverbrauch von Docker gibt. Somit hat der Nutzer, sofern er die Software nicht selbst konfiguriert, nur überschaubare Möglichkeiten den Energieverbrauch anzupassen. Durch die schiere Masse an Containern ist dies ein folgerichtiger Schritt, der in den nachfolgenden Kapiteln in Form eines Modells angegangen wird.

4 Vorhandene Forschungsergebnisse

Im folgenden Kapitel werden verwandte Veröffentlichungen vorgestellt, die innerhalb des Themenbereichs dieser Ausarbeitung behandelt werden. Dabei werden diese Artikel in die Bereiche Green ICT und Energieverbrauch von Software, Energieverbrauch von Rechenzentren, Energieverbrauch von Containern und, im Speziellen, Docker sowie inhaltsverwandte Arbeiten, die sich mit der Verbesserung von Container-Orchestrierung und Skalierung beschäftigen, aufgeteilt.

4.1 Verwandte Arbeiten im Bereich Green ICT und Energiemessung von Software

Energieverbrauchsmessungen im Bereich von Software haben in den letzten Jahren, wie in Unterkapitel 1.3 gezeigt, an Bedeutung zugenommen, sodass eine ISO-Norm für diesen Anwendungsfall geschaffen wurde. Die ISO/IEC-Norm 14756 (Information technology — Measurement and rating of performance of computer-based software systems) beschäftigt sich ausschließlich mit der Messung und Bewertung von Softwaresystemen und wurde in [Dirlewanger, 2006] vorgestellt. Diese Messmethode dient als Grundlage für die Messmethodik, die in dieser Ausarbeitung gewählt wurde (siehe dazu Unterkapitel 6.2). [Kern et al., 2018] nutzt ebenfalls diese Norm, um Kriterien für energieeffiziente Software aufzustellen und zu verallgemeinern. Dabei ist zu beachten, dass die Anzahl der Arbeiten in diesem Bereich überschaubar ist, wie [Penzenstadler et al., 2012] darstellt, obwohl es mehrere Umwelt-Einflussfaktoren von ICT, und im Speziellen Software, gibt. Insgesamt zeigt [Hilty and Aebischer, 2015] jedoch, dass das Forschungsfeld der Green ICT oder der ICT für Nachhaltigkeit wächst. Eine Sammlung aktueller Informationen über Energiemessungen im Bereich ICT, Messkriterien und energieeffiziente Programmierung stellt [Guldner et al., 2021] dar.

4.2 Verwandte Arbeiten im Bereich der Effizienz von Rechenzentren

Ähnlich der Energiemessung im Bereich Software werden zum aktuellen Zeitpunkt zwei Normen für energieeffiziente Rechenzentren aufgeführt. Hierzu findet man zum einen die DIN EN 303470:2018-10, die sich mit Messverfahren und Metriken für Server auseinandersetzt, und zum anderen die ISO/IEC CD 23544 für Application Platform Energy Effectiveness. Letztere befindet sich aktuell im Aufbau.

Insgesamt lassen sich einige Studien finden, die sich unter anderem mit der Energieeffizienz von Rechenzentren auseinandersetzen. Neben [Cisco, 2018], welche ausschließlich die Cloud analysiert und [Andrae, 2017] sowie [Andrae, 2019], in welchen Rechenzentren nur einen kleinen Teil einnehmen, vergleicht [Hintemann and Hinterholzer, 2019] mehrere Vorhersagemodelle zum Stromverbrauch von Rechenzentren und stellt abschließend ein eigenes Modell vor. Effektiv nach Lösungen für die in den

vorherigen Veröffentlichungen vorgestellten Probleme sucht beispielsweise [Mohammad Ali et al., 2017], die einen Vergleich zwischen monolithischen Serverstrukturen und Pool-Lösungen präsentiert und ein Modell für die effiziente Zuweisung von VM innerhalb eines Pools einführt und damit den Gesamtstromverbrauch im Vergleich zum monolithischen System um bis zu 42% senken kann. [Patterson, 2008] wiederum erläutert die Auswirkungen der Temperatur auf den Stromverbrauch von Rechenzentren und versucht eine optimale Temperatur für Serveranlagen zu finden. Ein Großteil weiterer Veröffentlichungen befasst sich mit der effizienten Ausschöpfung von Ressourcen durch Skalierung und Orchestrierung, auf die im Speziellen in Unterkapitel 4.4 eingegangen wird.

4.3 Verwandte Arbeiten im Bereich Energiemessung von Docker und Containern

Obwohl es, wie in den vorherigen Abschnitten dargestellt, eine anwachsende Zahl an Untersuchungen des Energieverbrauchs von Rechenzentren und Software gibt, ist die Anzahl an Arbeiten im Umfeld von Docker und Containern überschaubar. So zeigt beispielsweise [Santos et al., 2018], dass Docker insgesamt Auswirkungen auf den Energieverbrauch von Systemen hat. In diesem Fall wird unterschiedliche containerisierte Software betrachtet. Dabei wird zudem darauf eingegangen, dass Ausführungszeiten von Programmen in Containern länger dauern können und daher ebenfalls ein energetischer Overhead entstehen kann. Ebenfalls zeigt [Tadesse et al., 2017], dass der Großteil des Energieverbrauchs in Docker durch den CPU-Verbrauch erzeugt wird.
Etwas allgemeiner mit Containern befasst sich [Felter et al., 2015], der KVM (Kernel Based Virtual Machine) und Container miteinander vergleicht und zum Ergebnis kommt, dass der Overhead von Containern insgesamt niedriger ist. [Congfeng Jiang et al., 2016] geht einen Schritt weiter und vergleicht Docker direkt mit anderen Hypervisoren wie Xen oder Hyper-V. Hier wird gezeigt, dass Container-Technologien zwar leichtgewichtiger, aber trotzdem nicht viel energieeffizienter als herkömmliche Virtualisierungstechniken sind. [Morabito, 2015] relativiert diese Ansicht wiederum und stellt dar, dass der Unterschied bei geringen Arbeitslasten größer ist uns sich erst bei hoher CPU- und RAM-Auslastung angleicht. Allen zuvor genannten Arbeiten ist gemein, dass nur reduzierte Aussagen über Messumgebung im Allgemeinen und im Speziellen über die getesteten Szenarien und die Generierung der Arbeitslast getroffen werden. In [Santos et al., 2018] findet man zumindest eine genauere Beschreibung der Software, die im Container läuft. Gewählt wurden hier Redis, WordPress und Postgres nativ und im Container. Mit einem bestimmten Szenario setzt sich [Rehmann and Folkerts, 2018] im Detail auseinander. Hier werden Database Management Systeme verglichen, wobei ebenfalls der genaue Messaufbau nicht zu genüge beschrieben wird und daher nicht replizierbar ist.
Im Gegensatz zu den bisher vorgestellten Arbeiten befasst sich [Asnaghi et al., 2016] direkt mit dem Stromverbrauch von Docker Containern. Allerdings wird der Schwerpunkt hier auf die Anwendung von Containern auf IoT-Geräten gelegt. Ziel ist die automatisierte Ressourcenbeschneidung der Con-

tainer zur Erzeugung eines geringeren Stromverbrauchs des IoT-Geräts. Hiermit soll beispielsweise auf geringere Energiekapazitäten reagiert werden (Akkustand fällt, Solarpanel bekommt nur wenig Sonne).

4.4 Verwandte Arbeiten im Bereich der Container-Orchestrierung und Skalierung

Im Gegensatz zum Stromverbrauch von Container-Systemen findet man im Bereich der Skalierung und Orchestrierung einige Forschungsarbeiten. Dabei ist jedoch zu beachten, dass hier weniger die direkte Ersparnis von Energie als viel mehr die effizientere Auslastung der Hardware-Ressourcen eine Rolle spielt. Auch wenn die effiziente Auslastung von Hardware indirekt den Energieverbrauch beeinflusst, liegt der Fokus hier meist auf der Performance des Service.

Intensiv mit allgemeinen Skalierungstechniken setzen sich [Lorido-Botran et al., 2014] und [Al-Dhuraibi et al., 2018] auseinander. Allerdings liegt der Fokus hier auf Elastizität, also der automatisierten Zuweisung von Systemressourcen. In der Literatur wird oftmals keine Unterscheidung zwischen Skalierung und Elastizität gemacht (vergleiche dazu Absatz 3.3.1.5). Genauer mit der Skalierung befasst sich [Adolfsson, 2019] in seiner Masterthesis. Hier werden beispielhaft die Skalierungsrichtlinien von DigitalOcean bezüglich VMs und Containern evaluiert. Es wird gezeigt, dass eine Skalierung auf reiner Container-Basis oftmals zu Fehlern sowie in der Folge zu Ausfällen führt, während eine Mischung aus VMs und Containern die beste Skalierung gewährleistet.

Explizite Lösungen für eine verbesserten Skalierungsprozess liefert beispielsweise [Hanafy et al., 2019], wobei auch hier eher auf vertikale als auf die horizontale Skalierung eingegangen wird. In vielen Fällen unterscheidet sich im Bereich der Skalierung die Perspektive der Forschenden. So kann die Skalierung explizit auf eine bessere Auslastung von CPU und RAM zielen, wie in [Hanafy et al., 2019] oder wie in [Mao et al., 2017] auf eine bessere Auslastung der heterogenen Server eines Clusters. Beide verfolgen ein ähnliches Ziel, legen jedoch den Schwerpunkt anders. Vergleichbar damit ist das Ziel von [Imdoukh et al., 2019], mithilfe von Machine Learning eine effizientere horizontale Skalierung zu garantieren oder die Ausarbeitung von [Liu et al., 2018] unter Verwendung unterschiedlicher Optimierungstechniken die vertikale Skalierung von Docker Services zu verbessern. In beiden Fällen wird der Energieverbrauch nicht weiter beachtet. Unter Ausnutzung von Informationen zum weltweiten Stromnetz versucht hingegen [James and Schien, 2019], eine nachhaltige vertikale Skalierung zu erreichen. Je nach Tageszeit werden Services in denjenigen Regionen ausgerollt, in denen Strom am klimaneutralsten erzeugt wird. [Alzahrani et al., 2016] betrachtet ebenfalls den Energieverbrauch und schlägt einen neuartigen Skalierer vor, jedoch bezieht sich die Arbeit nicht auf Container, sondern analysiert Skalierung auf der Hardwareebene im Kontext der zur Verfügung stehenden CPU-Kerne.

Die Ausarbeitung [Casalicchio, 2019] betrachtet zwar nicht den Energieverbrauch, erforscht jedoch

den Kubernetes Scaler, der auch in dieser Arbeit evaluiert und adaptiert wird.[Casalicchio, 2019] zeigt dabei den Unterschied zwischen einer absoluten und relativen Betrachtung der CPU-Auslastung eines Containers und passt den Kubernetes Scaler entsprechend seiner Erkenntnisse an.
Im Gegensatz zu den bisherig vorgestellten Arbeiten, die sich stets nur auf Energieeffizienz oder Container beziehen, betrachtet [Piraghaj et al., 2015] beide Gesichtspunkte. Die Arbeit begutachtet die Konsolidierung von Containern auf virtuellen Maschinen. Dabei werden sowohl der Energieverbrauch von Container als auch die Verletzung von Service-Level-Agreements betrachtet, was abschließend in einem eigenen Modell und einem Algorithmus mündet.

4.5 Fazit und Abgrenzung

Wie im letzten Abschnitt gezeigt, schneiden viele Forschungsergebnisse die in dieser Ausarbeitung behandelte Thematik an, wobei manche Ergebnisse aufgrund der Versuchsaufbauten als kritisch zu betrachten sind. Wie bereits in Unterkapitel 1.3 angedeutet, wird sich diese Arbeit im Vergleich zu den meisten anderen Arbeiten explizit nur mit Docker beschäftigen und die Messexperimente auf die vorherige Recherche von Anwendungsszenarien aufbauen. Hier dienen die Ergebnisse von Tadesse et al [Tadesse et al., 2017] und Santos et al [Santos et al., 2018] als Grundlage. Durch Darstellung dieser Anwendungsszenarien soll gewährleistet werden, dass die Messergebnisse jederzeit repliziert und somit nachvollzogen werden können. Besonders im Bereich der Messmethodik wird auf die vorherigen Forschungsergebnissen von Kern et al [Kern et al., 2018] aufgebaut.

5 Nutzungsanalyse von Containersystemen und Docker

Im Verlauf der Arbeit wurde eine Umfrage zur Nutzung von Containern und im Speziellen Docker angefertigt, um aus deren Ergebnissen Nutzungsszenarien für Container zu entwickeln. Die Umfrage wurde an den Fachkonferenzen *Deutsche OpenStack Tage* sowie der *Container Conf*, die im Rahmen der *Continuous Lifecycle* stattfindet, durchgeführt. Die Erhebung enthält zwölf Fragen, die sich im Wesentlichen darum bemühen, herauszufinden, für welche Zwecke Entwickler und Administratoren Container nutzen und ob diese mit gängiger Literatur wie Liebel [2017], Öggl and Kofler [2018] und Miell and Hobson Sayers [2019] übereinstimmen. Auch soll überprüft werden, ob Nutzer an den zuvor vorgestellten Systemen von Docker Anpassungen vornehmen oder diese in ihren Standardeinstellungen belassen. Darüber hinaus ging es darum, im Sinne einer Sensibilitätsanalyse in Erfahrung zu bringen, ob Entwickler und Administratoren eine minimal schwächere Erreichbarkeit ihrer Services akzeptieren, wenn sie im Gegenzug Kosten einsparen können. Nachfolgend werden diese zwölf Fragen inklusive einer Darstellung und Auswertung der Ergebnisse dargestellt. An der Umfrage teilgenommen haben zwanzig IT-Profis aus unterschiedlichen Bereiche der Informatik.

5.1 Fragenkatalog der Umfrage

i) Zu welcher Berufsgruppe gehören Sie?
Antwortmöglichkeiten: Administrator; Entwickler; IT-Consultant; DevOps; Weitere

ii) Welche der folgenden Container-Systeme nutzen sie?
Antwortmöglichkeiten: Docker; rkt; Flockport; Podman; LXD; Weitere

iii) Wozu nutzen Sie Container?
Antwortmöglichkeiten: Auslieferung von Software; Software-Entwicklung; Bereitstellung von gesamten Systemen; Bereitstellung einzelner Services; Testing; Weitere

iv) Nutzen Sie Container hauptsächlich in Kombination mit Orchestrierungs-Software oder nativ mit Systemen wie Docker?
Antwortmöglichkeiten: Ja, hauptsächlich mit Orchestrierung; ausgeglichen; Nein, hauptsächlich nativ

v) Wo nutzen Sie Containersoftware?
Antwortmöglichkeiten: lokal; In der Cloud; Sowohl als auch; Ich nutze aktuell keine Containersoftware

vi) Welche der folgenden Punkte entsprechen in Ihrer Vorstellung typischen Einsatzszenarien für Container?
Antwortmöglichkeiten: Bereitstellung von REST-APIs; Load-Balancing und Skalierung von

Web-Services; einfach Bereitstellung ganzheitlicher Systeme; (Offline-)Entwicklung von Web-Services; Testing; CI/CD; Weitere

vii) Welche der folgenden Docker-Logging-Treiber nutzen Sie?
Antwortmöglichkeiten: Ich nehme keine Veränderungen am Default-Treiber vor; json-file; syslog; journald; gelf; fluentd; awslog; splunk; etwlogs; gcplogs; logentries; Weitere

viii) Welche der folgenden Docker-Netzwerk-Treiber nutzen Sie?
Antwortmöglichkeiten: Ich nehme keine Veränderungen am Default-Treiber vor; Weave; OVS; Default-Treiber; Contiv; Weitere

ix) Welche der folgenden Docker-Storage-Treiber nutzen Sie?
Antwortmöglichkeiten: Ich nehme keine Veränderungen am Default-Treiber vor; overlay2; aufs; devicemapper; btrfs; zfs; vfs; Weitere

x) Glauben Sie, dass Container über lange Zeit im Trend bleiben werden?
Antwortmöglichkeiten: Ja; Nein; keine Angabe

xi) Halten Sie es für einen guten Ansatz, den Energieverbrauch von Container-Systemen und, im Speziellen, Docker zu untersuchen?
Antwortmöglichkeiten: Ja; Nein; Ich habe keine Meinung dazu

xii) Denken Sie an die automatische Skalierung durch Orchestrierungs-Tools wie Kubernetes. Würden Sie eine minimal höhere Antwortzeit ihrer Services akzeptieren, wenn dadurch die Kosten dauerhaft gesenkt werden könnten?

Antwortmöglichkeiten: Ja; Nein; Vielleicht

5.2 Auswertung der Ergebnisse

Im Folgenden werden die Ergebnisse der Umfrage dargestellt. Die Grafiken zeigen dabei die jeweiligen Antworten im Verhältnis zueinander. Genauere Auswertungen der einzelnen Teilnehmerinnen und Teilnehmer (im Folgenden nur Teilnehmer) sind begleitend in prosaischer Form zu finden.
An der Umfrage partizipierten zwanzig Teilnehmer der Fachkonferenzen *OpenStack Tage* und *ContainerConf/Continuous Lifecycle*. Wie in Abbildung 29 zu sehen ist, verteilen sich die Teilnehmer dabei über sechs Berufsgruppen. Auffällig ist die relativ hohe Zahl an Berufstätigen im neu aufkommenden DevOps-Sektor. Wie sich gezeigt hat, verwenden alle Teilnehmer Docker, von denen zusätzlich drei auch Podman nutzen. Zwei dieser drei Teilnehmer setzen darüber hinaus ebenfalls LXD ein. Es zeigte sich im Verlauf der Auswertung, dass die Verwender alternativer Container-Software auch zur Nutzung von alternativen Storage-, Network und Logging-Treibern tendieren.

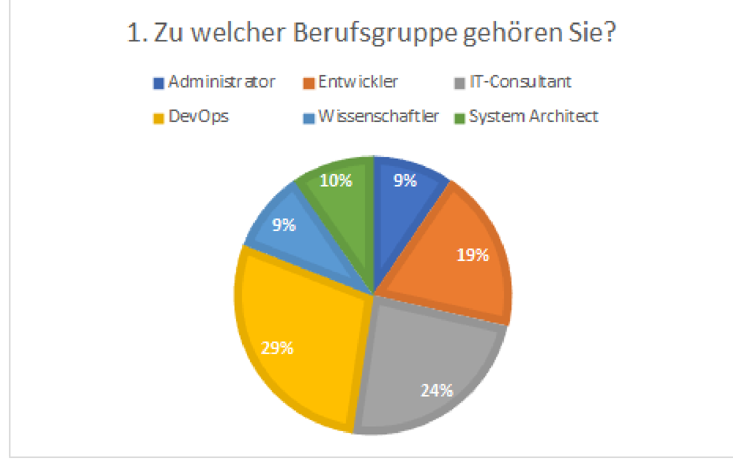

Abbildung 29: Verteilung der Umfrageteilnehmer auf verschiedene Berufsgruppen

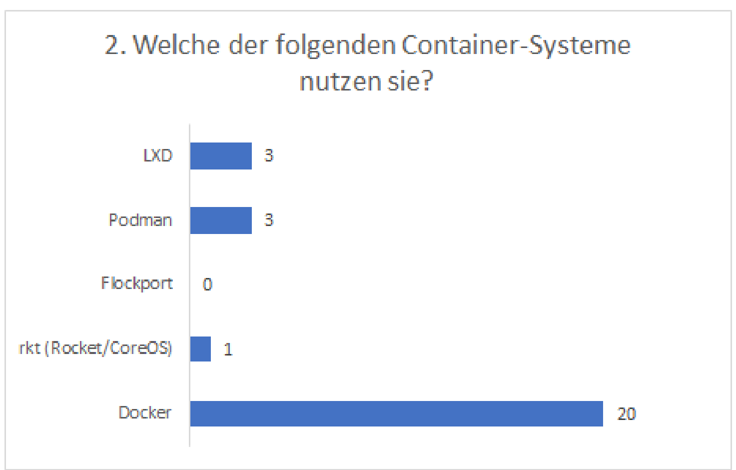

Abbildung 30: Nutzung verschiedener Containersysteme

In Abbildung 31 zeigt sich, dass die Teilnehmer Container nicht nur für einen Anwendungsfall nutzen, sondern Container in verschiedenen Anwendungsgebieten einsetzen. Lediglich drei Teilnehmer kreuzten dabei bloß einen Anwendungsfall an. Besonders ins Auge fallen hier jedoch Continuous Integration und Delivery. Es zeigt sich, dass elf von fünfzehn Teilnehmern, die CI/CD ankreuzten, auch Testing als Anwendungsfall betrachten. Selbes stellte sich bei der Softwareentwicklung dar. Auch hier kreuzten zehn von zwölf Teilnehmern, die Container in der Softwareentwicklung einsetzen, ebenfalls Testing an. Darüber hinaus wird ersichtlich, dass die Auslieferung von einzelnen Services und gleichfalls die einfache Bereitstellung ganzheitlicher Systeme (zum Beispiel ein kompletter Webshop oder eine Versionierungsplattform wie Gitlab) eine wichtige Rolle spielen.
Auch erkennbar ist, dass Machine Learning nur einmal als Anwendungsfall genannt wurde. Dies kann damit zusammenhängen, dass beispielsweise Docker, erst mit Version 19.03 (Herbst 2019) CUDA unterstützt.

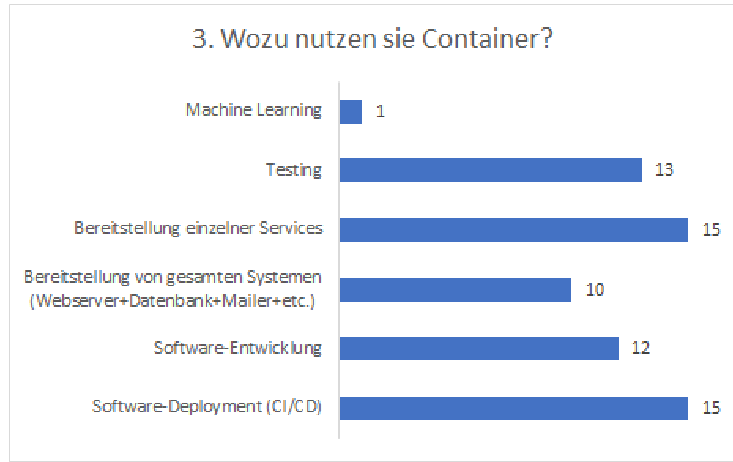

Abbildung 31: Verwendungszweck von Containern

Es zeigt sich ebenfalls eine ausgewogene Nutzung von Containern in Orchestrierungstools und nativ, wie in Abbildung 32 zu sehen ist. Hierbei ist zu beachten, dass sechs Nutzern, die Container hauptsächlich nativ nutzen, diese auch für die Software-Entwicklung verwenden.
Weiter ist aus Abbildung 33 ableitbar, dass Container meistens auf lokalen Servern eingesetzt werden, wobei acht von zwanzig Teilnehmenden Container sowohl lokal als auch in der Cloud verwenden.

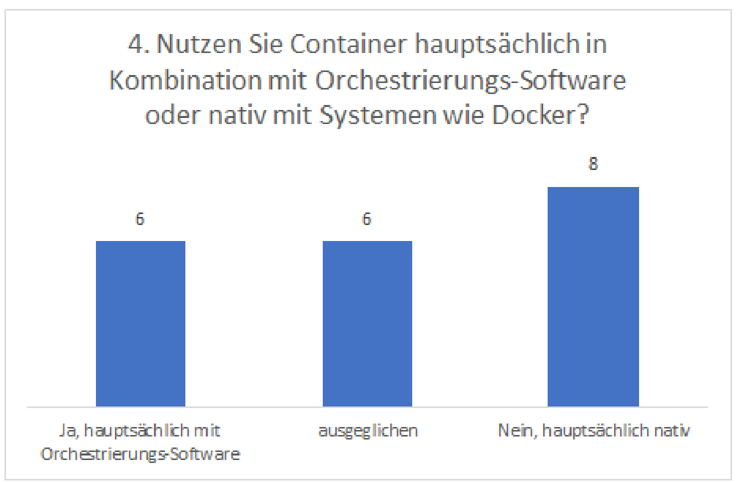

Abbildung 32: Native oder orchestrierte Verwendung von Containern

Bei typischen Einsatzszenarien von Containern stellt sich große Diversität dar. Aus den Ergebnissen, dargestellt in Abbildung 34, wird wie schon zuvor bei Abbildung 31 ersichtlich, dass sich CI und CD sowie das Testing bei den Teilnehmern zusammen genutzt werden. Zusätzlich wird verstärkt auch die einfache Bereitstellung ganzheitlicher Systeme genannt. Als konkrete Services ergeben sich REST-APIs und Webservices, die skaliert und lastbalanciert werden müssen, wobei REST-APIs meistens genau zu diesen Webservices zählen. Dies äußert sich ebenfalls darin, dass elf von dreizehn Befragten

die Skalierung ankreuzten, auch REST-APIs als charakteristisches Einsatzszenario für Container erachten. Auffällig ist, dass hier nur sieben Teilnehmer die Entwicklung als typisches Einsatzgebiet von Container auffassen, obwohl zuvor im Vergleich zu Abbildung 31 noch zwölf Befragte angaben, Container in der Software-Entwicklung zu nutzen. Zu den Vorteilen bei der Software-Entwicklung kann auch die einmalige Angabe des „compile once, run everywhere“-Prinzips gezählt werden, zu welchem exemplarisch Listing 2 aus Abschnitt 3.3.2 gehört.

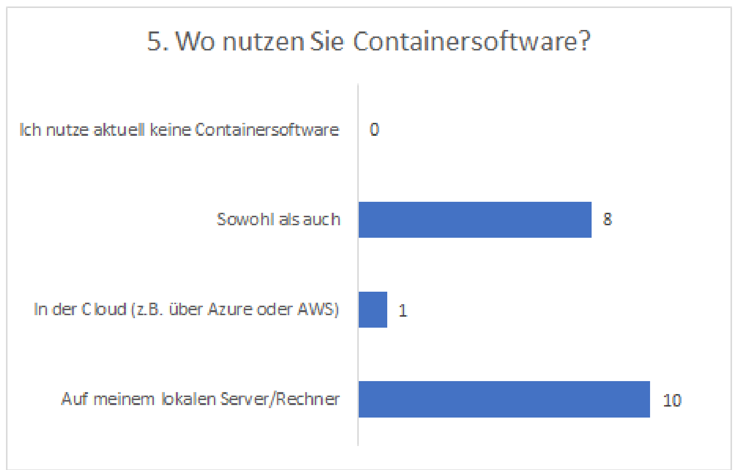

Abbildung 33: Verwendung von Containern in der Cloud oder lokal

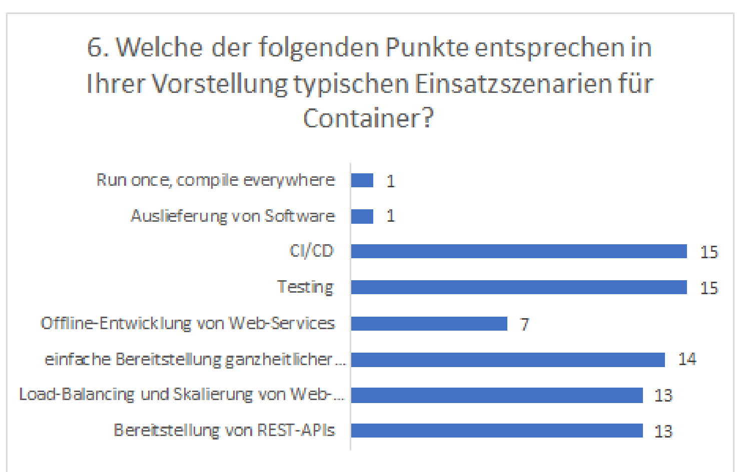

Abbildung 34: Typische Einsatzszenarien von Containern

Die Ergebnisse, die in Abbildung 35, Abbildung 36 und Abbildung 37 präsentiert werden, machen deutlich, **dass ein Großteil der Teilnehmer Docker nur in seinen Standardeinstellungen verwendet**. Somit rechtfertigt sich eine genaue Betrachtung der Docker Standardkonfiguration und weiterer Konfigurationsmöglichkeiten sowie eine Untersuchung des Docker Lifecycles auf Effizienzlücken. Ausschließlich bei Logging-Treibern ist ein leichter Trend zur Nutzung alternativer Logging-Treiber

erkennbar. Besonders *fluentd* und *syslog* stechen hierbei hervor. Zudem ist abzuleiten, dass die Logging-Treiber für Google Cloud (*gcplogs*) und AWS (*awslogs*) gar keine Anwendung bei den Befragten finden.

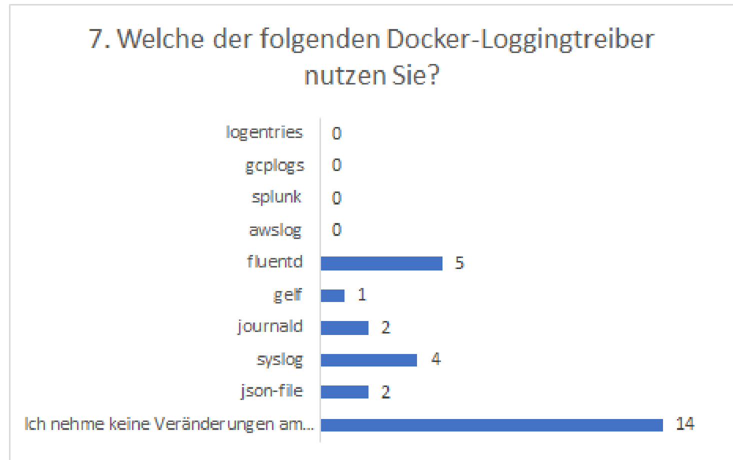

Abbildung 35: Welche Logging-Treiber werden verwendet?

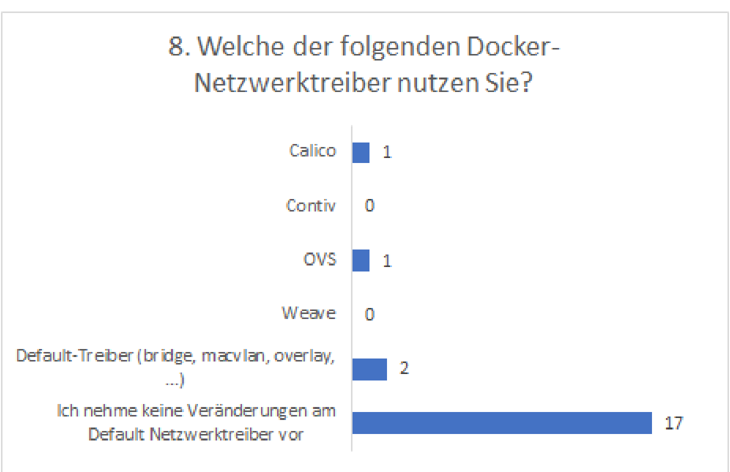

Abbildung 36: Welche Netzwerk-Treiber werden verwendet?

Unabhängig von der verwendeten Software ist in Abbildung 38 erkennbar, dass achtzehn von zwanzig Teilnehmer daran glauben, dass Container im Trend bleiben werden.
Weiter halten es neunzehn von zwanzig Befragten für sinnvoll, den Energieverbrauch von Containersystemen zu überprüfen. Abschließend zeigen die Ergebnisse von Frage 12, dass dreizehn der zwanzig Teilnehmer eine etwas geringere Erreichbarkeit ihrer Services zu Gunsten eines merklich besseren Energieverbrauchs akzeptieren würden. Weitere sechs Teilnehmer würden über ein solches Vorgehen nachdenken, wobei kein Teilnehmer sich dagegen aussprechen würde.

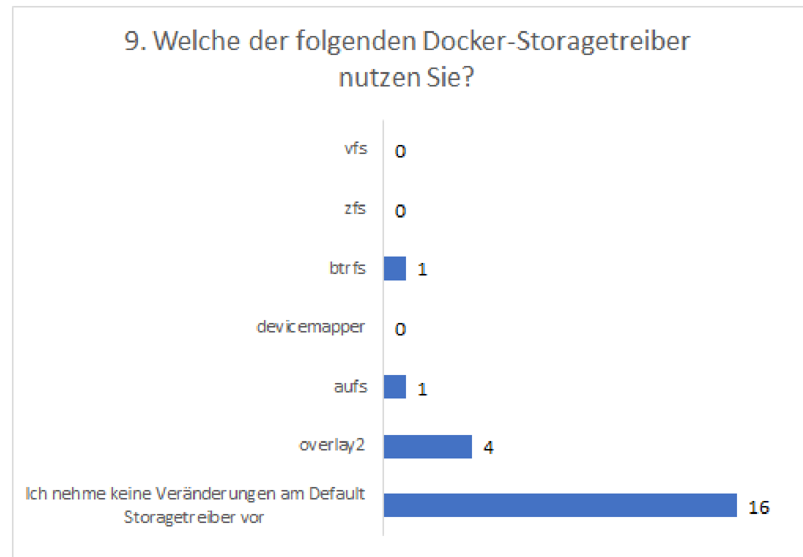

Abbildung 37: Welche Logging-Treiber werden verwendet?

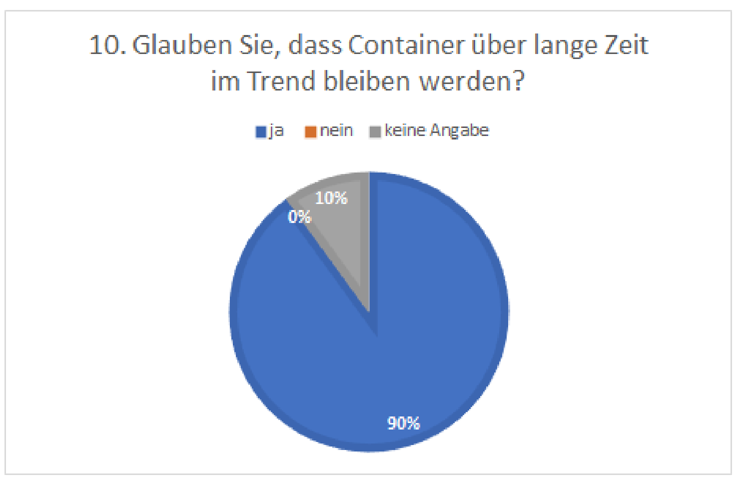

Abbildung 38: Bleiben Container ein Trend?

5.3 Interpretation der Ergebnisse

Obwohl die Befragung nicht repräsentativ war, weisen die Umfrageergebnisse trotzdem einige deutlich erkennbare Aspekte im Containerumfeld auf. Es ist zusätzlich zu bedenken, dass die Umfrage bei praxisbezogenen Fachkonferenzen unter Experten durchgeführt wurde. Die Erhebung zeigt unter anderem, dass Container von sehr unterschiedlichen Berufsgruppen verwendet werden (vergleiche Frage 1). Ob DevOps, Entwickler oder Consultant spielt dabei keine Rolle. Hier zeigen sich die vielfältigen Einsatzgebiete von Containern, wobei sich unerwarteterweise verstärkt der Einsatz in der Softwareentwicklung herausstellt. Speziell beim Testing scheinen Container in Zukunft immer mehr eingesetzt zu werden, wie beispielsweise das Tool Java Testcontainers unterstreicht, welches auf ähnlich einfa-

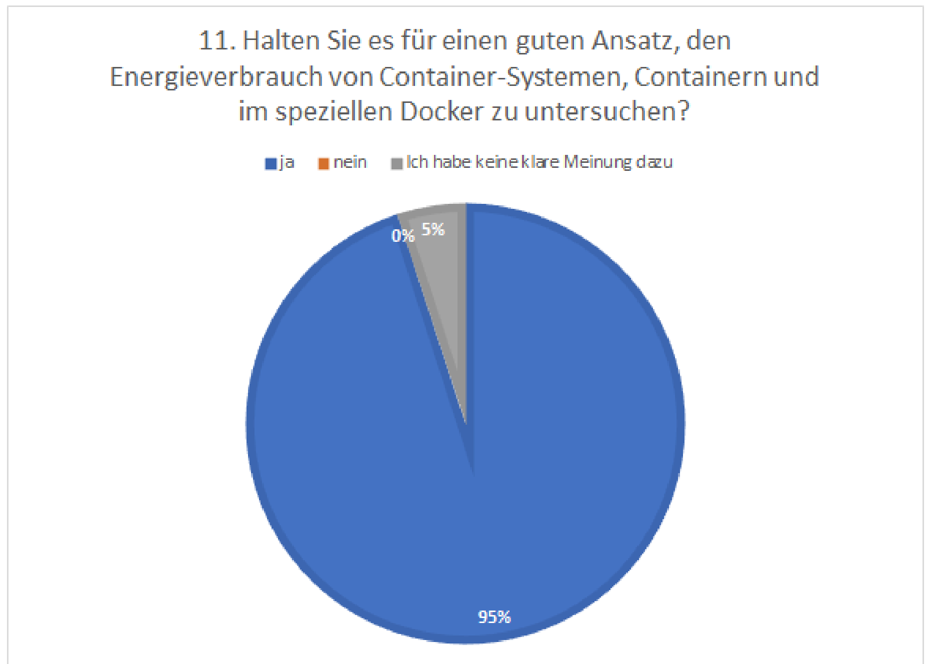

Abbildung 39: Sinnhaftigkeit von Energiemessungen im Containerkontext

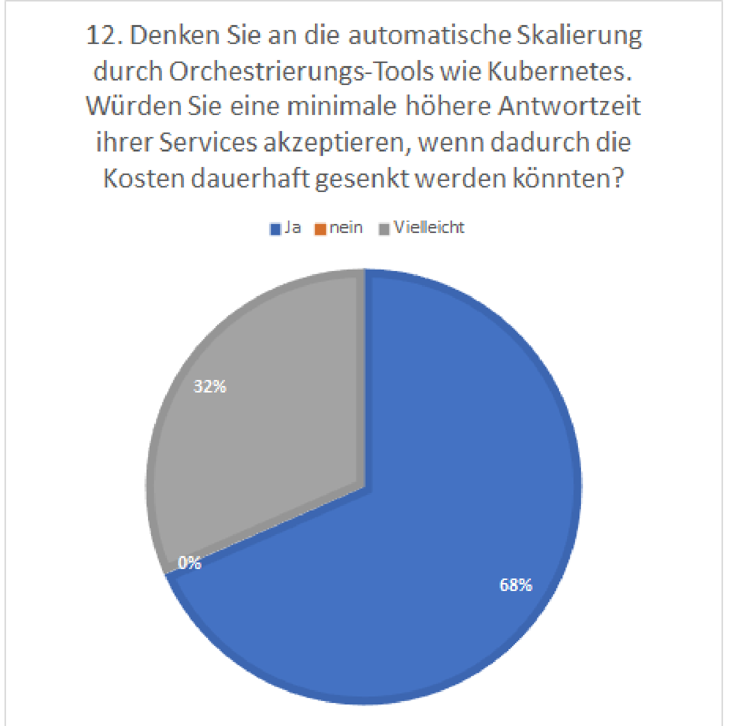

Abbildung 40: Sensibilität der Umfrageteilnehmer gegenüber einer Verschlechterung der Erreichbarkeit von Diensten mit gleichzeitigen Vorteilen beim Stromverbrauch

che Weise wie JUnit-Tests eingebunden werden kann und prominent in Fachmedien präsentiert wird (siehe exemplarisch [Wittek, 2019]). Zusätzlich ergibt sich auch, dass die Auslieferung von Software und ganzheitlichen Systemen eine wichtige Rolle für Nutzer spielt, was mit den in [Öggl and Kofler, 2018] und [Miell and Hobson Sayers, 2019] vorgestellten Szenarien absolut übereinstimmt. Besonders die REST-API als konkretes Beispiel scheint einen Stellenwert im Container-Umfeld zu haben.
Alle Befragten setzen verstärkt Docker ein, was dessen Einfluss als de facto Standard unterstreicht. Auch ist sehr auffällig, dass die Mehrheit der Umfrageteilnehmer die Standardeinstellungen von Docker verwendet. Da nahezu alle Container in den Standardeinstellungen laufen, ist es möglich, Docker „out of the box“ zu verwenden, was deutlich zum Erfolg der Software beigetragen hat. Wie sich im Verlauf der Ausarbeitung dargestellt hat, ist dies jedoch nicht immer von Vorteil. Zudem ist erwähnenswert, dass die Nutzer es für sinnvoll erachten, Containersysteme auf ihren Energieverbrauch hin zu untersuchen. Wie sich in Gesprächen zeigte, hatten die Teilnehmer ein gutes Verständnis dafür, welch immense Anzahl an Containern tatsächlich auf den weltweiten Servern laufen. Trotzdem bestätigten alle, dass bei der Entwicklung nur selten die Ressourceneffizienz in jeglicher Art betrachtet wird. Darüber hinaus ist erkennbar, dass keiner der Befragten einer geringeren Erreichbarkeit der in den Containern laufenden Services widersprechen würde, wenn im Gegenzug der Energieverbrauch spürbar gesenkt werden würde. Hier wurde in Kombination mit Gesprächen der Fachkonferenzteilnehmer deutlich, dass ein Umdenken hin zu einem umweltbewussteren Umgang mit Technik stattfindet. Die Teilnehmer äußerten unmissverständlich, dass die Sensibilität für das Thema Klimaschutz größer zu werden scheint. Alles in allem kann festgehalten werden, dass die Umfrage trotz fehlender statistischer Relevanz eine klare Tendenz aufweist, dass Docker selbst bei professionellen Entwicklern und Administratoren hauptsächlich in seinen Grundeinstellungen verwendet wird. Somit wird das Ziel, Handlungsempfehlungen für einen energieeffizienten Umgang mit Containern und im Speziellen Docker zu ermitteln, durch die Ergebnisse gestützt.

6 Modell zur energieeffizienten Nutzung von Containern und Messmethodik

Im nachfolgenden Kapitel werden die Bestandteile des angestrebten Modells zur energieeffizienten Nutzung von Containern inklusive der dazugehörigen Messmethodik vorgestellt.

6.1 Aufbau des Modells zur energieeffizienten Nutzung von Containern

Container selbst sind bereits ein Lösungsansatz für den steigenden Energieverbrauch von Rechenzentren. Sie helfen bei der Konsolidierung von Servern, indem sie den Overhead von virtuellen Maschinen reduzieren und somit die Packungsdichte auf Servern steigern. Dadurch soll parallel den steigenden Anforderungen an Rechenzentren in Form der Erreichbarkeit von Services entgegengewirkt werden. Außerdem erleichtern Container die Administration von Anwendungen sowie der Skalierung. Trotz dieser Tatsachen sind Container aber auch ein Teil des Problems. Durch Container wird der steigenden Nachfrage an Erreichbarkeit entgegengewirkt, was zu besseren Erfolgen der Services führt. Da Erfolge stets wiederholt werden wollen, resultieren daraus noch mehr Services, die auf die Erreichbarkeit angewiesen sind. Somit steigt die Anzahl an Berechnungseinheiten in Form von Containern stetig weiter. Will man also aktuell den Energieverbrauch von Rechenzentren und speziell in Cloud-Rechenzentren verbessern, reicht es nicht aus, sich auf die Hardware zu beziehen, sondern muss ebenfalls die Software untersuchen. Dazu zählt zum einen die Cloud-Software der Rechenzentrumsbetreiber, aber auch diejenige Software, die Entwickler, die auf PAAS, IAAS und FAAS setzen, programmieren. Beiden zugrunde liegen Container. Daher macht es Sinn, entweder die Containersoftware durch Neuentwicklungen zu verbessern oder bestehende Systeme bezüglich ihres Energieverbrauchs zu untersuchen. Aus diesem Grund wird im Folgenden ein Modell zur energieeffizienten Nutzung von Containertechnologien dargestellt, welches in den nachfolgenden Kapiteln umgesetzt und überprüft wird. Die **Kernbestandteile dieses Modells** sind dabei:

i) Eine geeignete Messmethodik zur Energieverbrauchsmessung von Containern (Vorstellung in Unterkapitel 6.2)

ii) Aus entsprechenden Messungen in Standardnutzungsszenarien erzeugte, praktisch durch Administratoren und Entwickler anwendbare Handlungsempfehlungen einer energieeffizienten Konfiguration von Containern (Vorstellung in Kapitel 8)

iii) Tools zur weiteren Verbrauchssenkung von Containern (Vorstellung in Abschnitt 8.1.4)

Da im Zuge dieser Arbeit nicht alle verfügbaren Containersysteme untersucht werden können, konzentriert sich das Modell auf den de facto Standard Docker. Hierbei wird Docker unorchestriert untersucht, da selbst in der Cloud 50% der Container ohne Tools wie Kubernetes administriert werden (vergleiche

hierzu Umfrage des Cloudmonitoring und Cloud Analysten Datadog [datadoghq.com, 2018]). Wie in den Umfrageergebnissen in Kapitel 5, [sysdig.com, 2021] und [datadoghq.com, 2018] ersichtlich, verwenden die meisten Nutzer von Container-Systemen Docker und verändern die Grundkonfigurationen von Containern und Docker selbst nicht. Dies wird, wie in Abschnitt 3.4.2 gezeigt, durch die Einstellungsmöglichkeiten in der Cloud begünstigt. Daher wird in Kapitel 7 Docker im Sinne eines Bottom-Up Ansatzes vom Kleinen zum Großen untersucht, indem zuerst der Docker Lifecycle entsprechend Abschnitt 3.4.3, bestehend aus Build-Prozess sowie Starten, Stoppen und Löschen von Containern, analysiert wird, bevor auf die Kernkomponenten von Docker gemäß Abschnitt 3.3.2, wie Networking, Volumes, Storage-Treiber und Skalierung eingegangen wird. Untersucht werden diese Bestandteile in Standardnutzungsszenarien, die über die Ergebnisse der Umfrage aus Kapitel 5 sowie gängige Literatur evaluiert wurden. Aus den Untersuchungen soll folgen, dass selbst Nutzer mit geringen Kenntnissen einen energetischen Vorteil aus den Handlungsempfehlungen zur Konfiguration von Docker ziehen können.

Am Ende des Vorgangs werden die Handlungsempfehlungen, die in Kapitel 8 final dargestellt werden, in Form einer Simulation in Unterkapitel 8.1 auf ihre Auswirkungen getestet, indem ein Einsatzszenario von Containern, welches durch die Umfrage in Kapitel 5 sowie [Miell and Hobson Sayers, 2019] und [Öggl and Kofler, 2018] ermittelt wurde, mit Hilfe von Default-Containern aus Docker Hub umgesetzt wird. Nachdem der Energieverbrauch des Szenarios entsprechend der Messmethodik bestimmt wurde, soll das Testszenario entsprechend der Handlungsempfehlungen angepasst und erneut getestet werden. Beide Ergebnisse werden in Form einer Simulation auf ein längeres Zeitintervall angewendet, um so die zu erwartende Energieersparnis innerhalb des Zeitintervalls zu bestimmen. Daran anschließend werden die Handlungsempfehlungen in den Kontext anderer Container-Systeme gebracht, um deren Anwendbarkeit außerhalb des Docker-Ökosystems zu zeigen. Nachfolgend wird in Abschnitt 8.1.4 ebenfalls im Sinne des Bottom Up Ansatzes die Ebene von der Containersoftware zur Containerorchestrierung gewechselt. Hier soll mithilfe der durch die Versuche erhaltenen Ergebnisse ein Tool in Form einer Erweiterung einer weit verbreiteten Skalierungsmethode konzipiert werden. Diese soll sowohl auf den Energieverbrauch als auch auf die Erreichbarkeit von Services im Cluster achten. Sie soll dem Nutzer die Möglichkeit geben, aktiven Einfluss auf beide Faktoren zu haben. Es ist zu erwähnen, dass kein eigener Scaler entwickelt wird. Die hierfür notwendigen Vorgänge wären ausreichend für eine eigenständige Ausarbeitung.

Um einen ersten Einblick in den Energieverbrauch von Docker zu finden, wird in einem Vorexperiment untersucht, ob die Programmiersprache Go, in welcher Docker geschrieben ist und die heute bei Programmierern für Cloud-Anwendungen großen Zuspruch findet, energieeffizient arbeitet. Dafür werden bestimmte Features, die zur Nutzung mehrerer Container notwendig sind, analysiert.

6.2 Methodik und Messumgebung (Modell-Kernbestandteil I)

In einem Schritt zur Erzeugung des Modells, wurden anhand der Standardnutzungsszenarien eine Messmethodik und Messumgebung entwickelt, die es erlaubt, genau diese Szenarien bezüglich ihres Energieverbrauchs zu überprüfen. Im Verlauf des Arbeitsprozesses wurden dabei verschiedene Messumgebungen und Methodiken evaluiert, die sich in Genauigkeit und Realitätsnähe unterscheiden. Dabei wurden sowohl die Messgeräte als auch die Datenanalysevorgänge verändert und angepasst, um diese einem realen Einsatzszenario anzugleichen und in einem Rechenzentrum oder bei einem Entwickler replizierbar und lauffähig zu machen. Die Replizierbarkeit wird durch die nachträgliche Veröffentlichung der Messskripte und GNU Lizenz zusätzlich gesteigert. Die Anforderungen an die Messumgebung sahen dabei wie folgt aus: Hohe Messgenauigkeit bei gleichzeitiger praktischer Anwendbarkeit ohne zusätzliche Kosten. Darüber hinaus wurde Wert darauf gelegt, dass die Messmethodik auf anderen Container-Systeme anwendbar ist. Im Folgenden werden zuerst die Iterationen des physischen Messaufbaus beschrieben, die indirekten Einfluss auf die Umsetzung des Datenanalysevorgangs hatten. Die Iterationen des Datenanalysevorgangs werden im zweiten Abschnitt ausgeführt. Jede Iteration erfüllt dabei die Anforderungen an die Genauigkeit einer Messung, eignet sich jedoch besser für die zu untersuchenden Messszenarien. Demnach kann jeder Iterationsschritt als Weiterentwicklung und Verbesserung des vorherigen Schritts angesehen werden.

6.2.1 Iterationen des physischen Messaufbaus

Die Messumgebung wurde bereits in der ersten Iteration nach Vorgaben der ISO/IEC-Norm 14756 (Information technology — Measurement and rating of performance of computer-based software systems) gestaltet, die in Dirlewanger [2006] vorgestellt wurde. Zudem wurde sich an die Leitlinien aus Kern et al. [2018] gehalten, welche sich explizit mit Energiemessungen im Bereich von Softwareprodukten auseinandersetzen. Hier wurde zudem ein entsprechender Versuchsaufbau präsentiert (siehe Abbildung 41), der die Bestandteile einer Energiemessumgebung definiert.
Im Zentrum des Messaufbaus steht das System Under Test (SUT), auf welchem die zu untersuchende Software läuft. Das SUT wird mit einer der Software entsprechenden Arbeitslast belegt, um so den Energieverbrauch mit Hilfe eines Power Meters (PM) messen zu können. Ein weiteres System (Data Aggregator and Evaluator (DAE)) sammelt dabei Daten über die Ressourcenauslastung des SUT, den Energieverbrauch sowie die verrichtete Arbeit. Abschließend erzeugt das DAE einen Energieeffizienzbericht, der Aufschluss über den Energieverbrauch der Software gibt. Um reproduzierbare Ergebnisse liefern zu können, empfiehlt [Kern et al., 2018] die Definition eindeutig wiederholbarer Testszenarien. Diese sollen derart gestaltet sein, dass sie zum Vergleich auf unterschiedliche Software angewendet werden können. Hierdurch sollen sowohl systematische als auch messtechnische Fehler bei der Wiederholung der Tests vermieden werden. Im Falle dieser Arbeit wurden die Testszenarien anhand von Best Practices verschiedenster Quellen wie [Miell and Hobson Sayers, 2019] und [Öggl and Kofler,

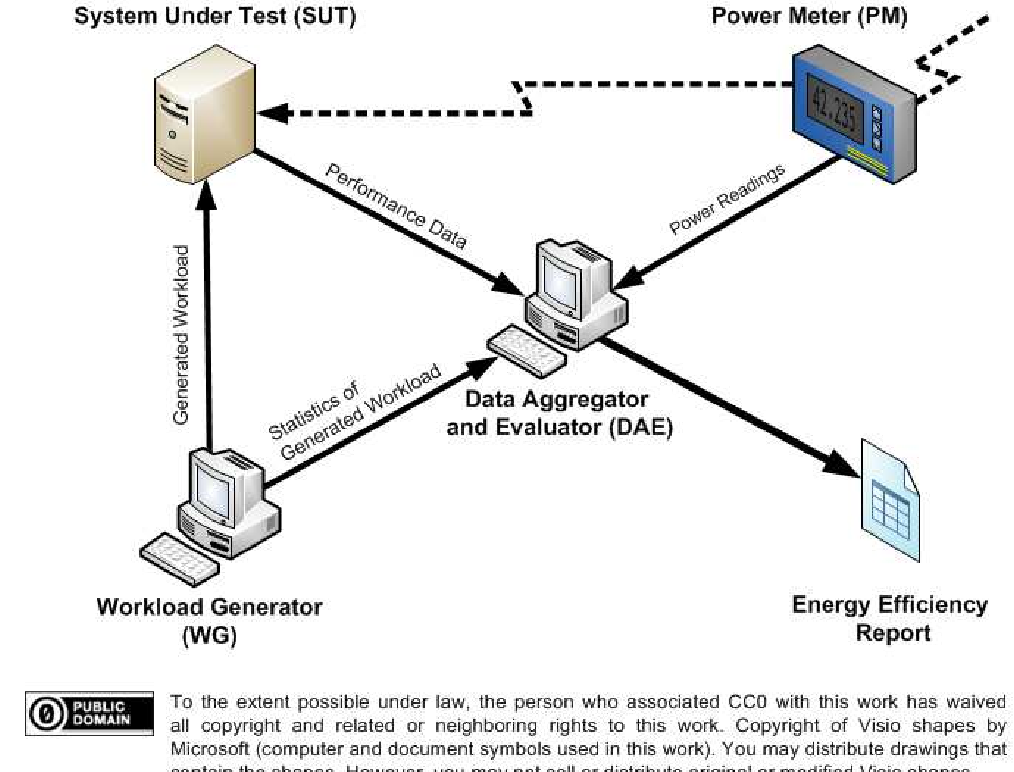

Abbildung 41: Messumgebung zur Bestimmung des Energieverbrauchs von Software [Kern et al., 2018]

2018] sowie den Ergebnissen der Umfrage aus Unterkapitel 5.2 erzeugt. Sie werden in Kapitel 7 vor jedem Abschnitt kurz dargestellt.

Bei der Auswahl der Komponenten für die erste Iteration des Messaufbaus wurde vor allem Wert auf die Genauigkeit der Messungen gelegt. Daher wurde zu Beginn auch vor allem auf die Wahl des SUT geachtet. Es wurde von vorneherein ausgeschlossen, ein ausgewachsenes Serversystem zu verwenden, da hier alleine im Leerlauf mehrere 100 Watt an Energie verbraucht werden können. Da der Fokus zu Beginn auch auf einzelne Container gelegt werden sollte und diese im Allgemeinen nur einen geringen Stromverbrauch im einstelligen Bereich erzeugen, bestünde die Möglichkeit, dass Änderungen im energetischen Rauschen des SUT untergehen. Aus diesem Grund fiel die Wahl im Verlauf der Arbeit auf ein Kleinstsystem mit der Vorgabe einer x86 Architektur, welches sich bestmöglich mit Servern vergleichen lässt. Vorhergehende Messungen wurden auf Desktop-PCS durchgeführt. Auch stellte sich die Frage, ob die Messungen durch ein Software- oder Hardware-Messinstrument durchgeführt werden sollten. Im Bereich der Software-Strommessungen besteht aktuell nur ein Tool, welches von der Genauigkeit her akzeptable Ergebnisse liefert. Hierbei handelt es sich um Intels RAPL (Running Average Power Limit), mithilfe dessen der Energieverbrauch der einzelnen Kerne einer CPU bestimmt werden kann. Das Tool ist für alle Intel CPUs ab der Sandy Bridge Generation lauffähig und funktioniert daher auch im Server-Bereich auf modernen Xeon CPUs. Für den ersten Messaufbau wurde bewusst auf Intels RAPL verzichtet, da die Bindung an einen CPU-Hersteller vermieden werden sollte

und weitere Komponenten des Rechners, die durch den Container verwendet werden, mit in den Energieverbrauch eines Containers einfließen. Darüber hinaus müssten DAE und SUT eine Einheit bilden, was wiederum energetischen Overhead und mögliche Störeffekte mit sich bringen könnte. Alternativ müsste die Möglichkeit eines Remotezugriffs auf die Werte des RAPL-Tools erzeugt werden. Ferner würde dies bedeuten, dass das aus der Arbeit resultierende Tool zur Skalierung eine Veränderung der Grundsysteme auf Servern mit sich bringen würde, um auf die Messwerte zugreifen zu können. Dies wurde von Beginn an kategorisch ausgeschlossen, um Administratoren vor zusätzlicher Arbeit zu bewahren. Abschließend wurde daher ein externes Messgerät in Betracht gezogen. In der ersten Iteration des Messaufbaus sahen die Komponenten daher wie folgt aus:

- PM: Janitza UMG 604 Netzanalysator [Janitza, 2021]
- SUT: Intel i5-3320 (2 Kerne, 2,6 GHz, 8 GB RAM)
- Bestimmung von CPU-Verbrauch und RAM-Verbrauch über Windows Performance Monitor auf dem SUT
- WG: analoges Zweitsystem zum SUT

Dieser Messaufbau inklusive der Auswertungsmethodik wurde unter anderem in [Mancebo et al., 2019] mit dem Messaufbau eines spanischen Teams verglichen. Trotz großer Unterschiede im Aufbau waren die Ergebnisse stets vergleichbar, was für die Genauigkeit des Messaufbaus und die Reproduzierbarkeit der Messergebnisse spricht.
Nach dem ersten Messaufbau wurde davon abgesehen, die Daten zum CPU- und RAM-Verbrauch direkt auf dem SUT vorzunehmen und so weiteren Overhead zu vermeiden. Bei den Messungen, die Docker direkt betreffen, wurde dies über die Docker-API umgesetzt. So ist es möglich, dass der DAE die Daten direkt über Https vom System erfragt. In Container-Orchestrierungstools findet diese Vorgehensweise ebenfalls Anwendung und ist daher deutlich praxisnäher. Auch hierbei ist der energetische Overhead durch die Https-Anfragen an den Docker-Host zu beachten. Um diesen Overhead während der Messung ausschließen zu können, wurde das System derart verändert, dass jede Abfrage des Ressourcenverbrauchs genau nach jeder Strommessung vollzogen wird. Außerdem wird die Anzahl an Abfragen statisch festgelegt und ist somit in jeder Testrunde gleich und folglich vergleichbar. Der Performance Monitor von Windows lässt sich auf diese Weise nicht konfigurieren, da dieser unabhängig vom restlichen Messverfahren ist. Demnach sah die zweite Iteration der Messumgebung wie folgt aus:

- PM: Janitza UMG 604 Netzanalysator
- SUT: Intel i5-3320 (2 Kerne, 2,6 GHz, 8 GB RAM)
- DAE: analoges Zweitsystem zum SUT erfragt Ressourcenverbrauch über Https vom SUT

- WG: analoges Drittsystem zum SUT

Obwohl das Janitza UMG 604 ein sehr genaues Messgerät ist (Spannung +- 0,2%, Strom +- 0,25% und garantiert lückenlose Messung), wurde es in der dritten Iteration des Messaufbaus durch ein anderes Messgerät ersetzt. Das Janitza Messgerät muss durch einen Experten am Stromkasten befestigt werden, was zu höheren Kosten bei Rechenzentrumsbetreibern führen würde. Da sich jedoch in den meisten Serverschränken sogenannte switched PDUs befinden, eine Art Mehrfachsteckdosen, die über TCP/IP gesteuert werden können und ebenfalls den Stromverbrauch mitloggen, können diese gleichermaßen für die Messungen verwendet werden. Wie beim Janitza liegt die Genauigkeit im durch die DIN Norm EN 61557-12:2008 geforderten Rahmen. Ebenfalls musste der Messaufbau die Konfiguration eines Clusters gewährleisten, auf dem die Skalierung der Container durchgeführt werden konnte, auch, wenn die Messungen grundsätzlich nur auf einem System durchgeführt wurden. Daher wurde das SUT an dieser Stelle durch das vorher erwähnte Kleinstsystem ersetzt, welches mehrfach angeschafft wurde. Ein weiterer Vorteil des Geräts ist die niedrige Leistungsaufnahme im IDLE, die nur circa 4,5 Watt beträgt. Dadurch können selbst Messungen an einzelnen Containern mit geringen Arbeitslasten durchgeführt werden, ohne dass die Messungen im energetischen Rauschen des Geräts untergehen. Daraus ergibt sich die dritte und finale Iteration des Messaufbaus:

- PM: Gude Expert Power Control 1202 [Gude, 2021]
- SUT: 2x Udoo x86 Advanced Plus (Intel Celeron N3160 2.24 GHz (4 Kerne), 4 GB RAM)
- DAE: Lenovo Thinkpad U430
- WG: Lenovo Thinkpad U430

Wie zu sehen ist, wurden DAE und WG zusammengefasst, um so parallel zur Erzeugung der Arbeitslast eine Strommessung durchzuführen und effektiver die Auswirkungen von steigender Arbeitslast auf den Stromverbrauch bestimmen zu können. Dadurch wurde auch die Anzahl an benötigten Geräten verringert, ohne die Qualität der Messungen zu beeinflussen.
Auf dem SUT läuft in diesem Fall Ubuntu 18.04 LTS und Docker in der Version 19.03.1. Die Skripte in Python 2.7 verfasst.

6.2.2 Iterationen des Datenanalysevorgangs

Bedingt durch die erste Iteration des physischen Messaufbaus, wurden sowohl die Energie-Messwerte als auch die Performance-Messwerte des SUT in Form von CSV-Dateien abgespeichert. Erst im Nachhinein der Messungen war es dadurch möglich, mithilfe eines zusätzlichen R-Skripts, einen Analysebericht der Messung anzufertigen. Im Sinne der praktischen Anwendbarkeit und der weiteren Verwendung der Messwerte für eine automatisierte Skalierung, wäre eine solche nachträgliche Analyse

kontraproduktiv gewesen. Aus diesem Grund wurde bei den folgenden Iterationen darauf geachtet, dass die Messwerte über ein entsprechendes Ethernet-Protokoll auslesbar sind. Standardmäßig bieten Energie-Messgeräte, so auch das Janitza UMG 604, die Möglichkeit, Messwerte remote über Protokolle wie Modbus, SNMP oder HTTP auszulesen. Im Falle des Janitza Messgeräts gestaltete sich dies als problematischer als beim daraufhin gewählten Gude Expert 1202. Beim Gude Messgerät war es möglich, die Messwerte bezüglich Strom, Spannung und Phase über SNMP auszulesen und diese sofort zu verarbeiten. Daher wurde ab der dritten Iteration ein eigens verfasstes Messskript in Python verwendet, welches im Folgenden genauer beschrieben wird.

6.2.3 Aufbau des Messskripts

Da das selbstverfasste Messskript zum einen den Lastgenerator als auch den Daten-Aggregator darstellt, war es wichtig, den Moment der Lasterzeugung an den Moment der Messung und umgekehrt anzupassen. Aus diesem Grund fiel die Wahl auf Python als Programmiersprache. Hier stellte sich die Arbeit mit Nebenläufigkeit und dem zu Python gehörenden Event-System, im Vergleich zu den durch die Messungen aus Unterkapitel 7.1 gesammelten Erfahrungswerten, als am angenehmsten heraus im Vergleich zu den durch die Messungen aus Unterkapitel 7.1 gesammelten Erfahrungswerten. Go wäre hierbei aufgrund des Channel-Systems eine gute Alternative gewesen.
Das Messskript erfasst in der aktuellsten Iteration CPU und RAM-Auslastung der laufenden Container auf dem SUT. Dies wird mit Hilfe eines in Python zur Verfügung stehenden Wrappers für die Docker API umgesetzt. Nachdem der Docker Host in Python initialisiert wurde, erhält man ein Array, in dem die ID der Container gespeichert ist. Über diese ID wiederum lassen sich direkt beim Messpunkt des Energieverbrauchs Informationen zum Ressourcenverbrauch der Container im JSON-Format abfragen. Hierfür bedarf es einer einfachen Anpassung des Docker-Hosts, um diese Informationen über einen expliziten Port auslesen zu können.
Um parallel dazu Last auf dem SUT zu erzeugen, wird unter anderem das Open Source Tool Siege [Joe Dog Software, 2021] verwendet, welches Stresstest für Server ermöglicht. Diese Software wurde im Zuge dieser Ausarbeitung angepasst, um sowohl auf durchgeführte Messungen zu reagieren, als auch die Messergebnisse verarbeiten zu können. Im Verlauf der Arbeit wurde ebenfalls ein eigenes Python-Tool verfasst, das Siege ersetzen soll. Aufgrund der Verlässlichkeit und Reproduzierbarkeit der Lastgenerierung des Tools Siege wurde jedoch in den weiter unten beschriebenen Messungen auf Siege gesetzt.
Außerdem mussten die Energie-Messwerte mit Hilfe des Linux Command Line Tools SNMPGET vom Messgerät ausgelesen werden. Um diese Messwerte auslesen zu können, muss das Messgerät eine MIB Tabelle bereitstellen. In dieser sind Adressen des Speicherregisters des Messgeräts abgelegt, auf die direkt zugegriffen werden kann. Im Falle des Gude Expert 1202 sieht dies wie folgt aus:

```
snmpget -v1 -c private 192.168.0.72 1.3.6.1.4.1.28507.43.1.5.1.2.1.6.1
```

Listing 8: Auslesen der Spannung vom Messgerät über SNMP

Auf diese Weise ist es möglich, Messkurven direkt am DAE/WG zu speichern und beispielsweise für die Skalierung zu verwenden. Aus diesem Grund steht das Messskript in drei Ausführungen bereit. So gibt es ein Skript für den Test einzelner Container sowie ein Skript für skalierte Container. Darüber hinaus existiert ein Skript, welches die Messwerte zur Überprüfung in eine lokale Datenbank speichert. Hierfür bieten sich Datenbanken für Sensordaten wie InfluxDB an. Ein großer Vorteil von InfluxDB ist die einfache Anbindung an Graphana zur visuellen Darstellung der Messergebnisse. Zur weiteren Verdeutlichung des Ablaufs ist in Abbildung 42 eine Skizze des Skripts in Form eines UML Activity Diagramms dargestellt.

6.2.4 Ablauf einer Messung

Auch wenn der Messablauf stets an das Messszenario angepasst werden muss, soll an dieser Stelle grob beschrieben werden, wie der generelle Messaufbau gestaltet ist. Anpassungen werden jeweils für die einzelnen Messungen in Kapitel 7 beschrieben.

Zu Beginn einer jeden Messung wird der grundlegende Stromverbrauch des Systems im IDLE, ohne laufenden Container gemessen. Die Messung wird zehn Mal durchgeführt und dauert jeweils eine Minute. Zwischen jedem Messdurchgang liegt eine Pause von mindestens dreißig Sekunden. Der so ermittelte Stromverbrauch wird *Baseline* genannt. So kann im Nachgang die reine, durch die zu untersuchende Software induzierte, Leistungsaufnahme gezeigt werden.

Nachdem die Baseline des Systems bestimmt wurde, wird die Messung eines Messszenarios angestoßen. Auch diese Messung wird zehn Mal wiederholt. Die Dauer eines Messdurchlaufs ist dabei abhängig vom Messszenario. So gibt es Messszenarien, bei denen die Dauer des Messdurchlaufs fest vorgegeben wird, während andere Prozesse nach einer gewissen Zeit abgeschlossen werden. Diese Dauer wird ebenfalls gemessen. Unabhängig von der Dauer des Messdurchgangs findet zwischen jedem Durchgang erneut eine mindestens dreißigsekündige Pause statt. Durch diese Vorgaben und die genaue Beschreibung eines Messszenarios ist die Messung später einfacher reproduzierbar. Weiter lassen sich die Messrunden miteinander vergleichen, was zu einer besseren Interpretierbarkeit der Daten führt.

Mit Ausarbeitung der Messumgebung und der Messmethodik, wurde bereits die Basis für das beschriebene Modell gelegt. Die Methodik kann nun auf die evaluierten Standardnutzungsszenarien, die in Kapitel 7 entsprechend des zu messenden Gegenstandes vorgestellt werden, angewendet werden. Demnach stellen die nachfolgenden Kapitel 7, 8 und 9 die Vervollständigung des in Unterkapitel 6.1 dargestellten Modellansatzes dar.

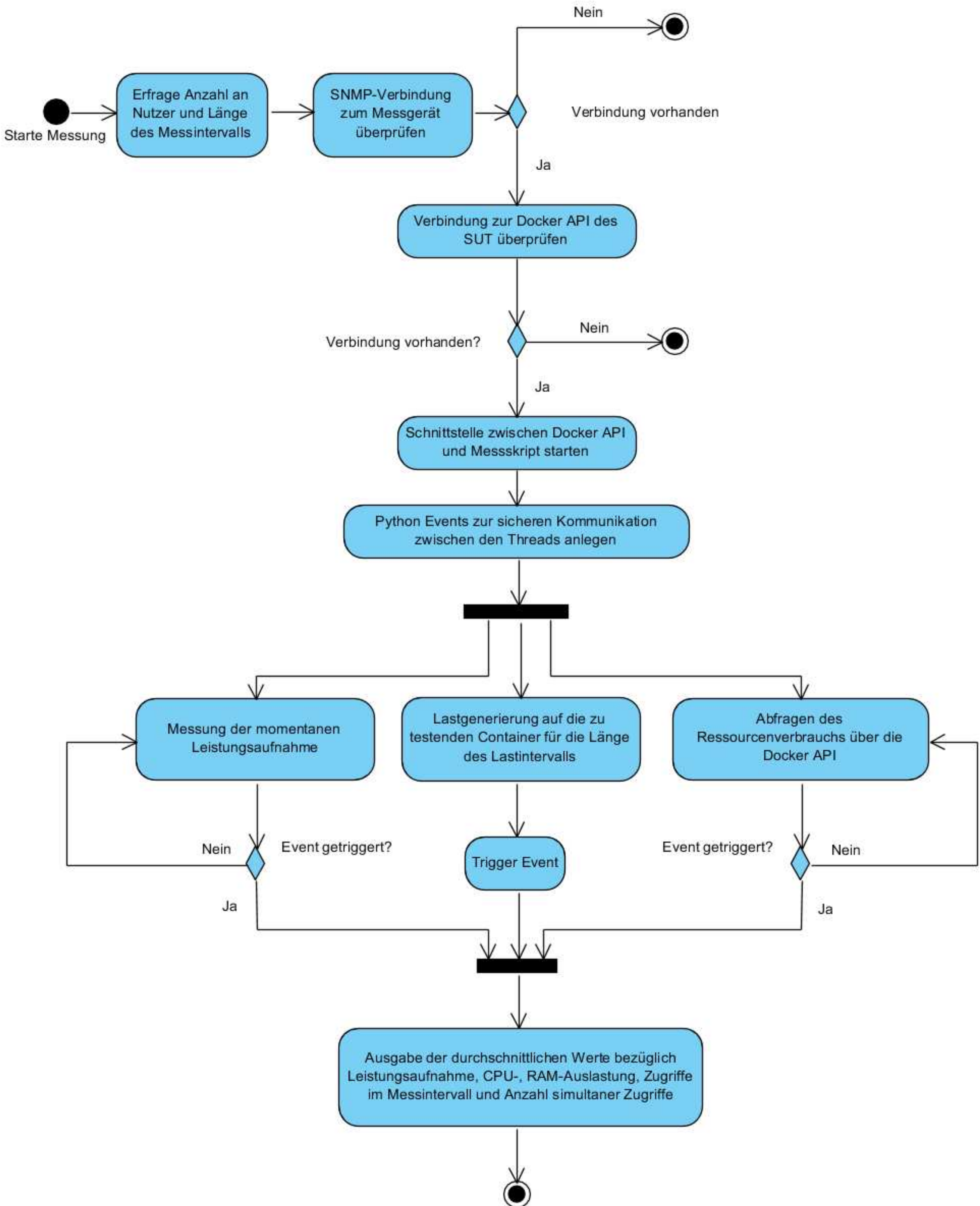

Abbildung 42: UML Activity Diagramm zum groben Ablauf des Messkripts (eigene Abbildung)

7 Darstellung der einzelnen Messszenarien, der Messergebnisse und deren Interpretation

Im folgenden Kapitel werden die Messungen mit ihren Messszenarien und Messumgebungen vorgestellt. Dabei wird hier ein besonderer Fokus auf die Auswahl der Messszenarien gelegt, die für die Messungen einen wichtigen Faktor darstellen. Die Messszenarien müssen nämlich so gewählt werden, dass die im Container befindliche Software einem typischen Szenario von Docker, evaluiert in Unterkapitel 5.2, entspricht. Gleichzeitig soll die Software selbst so einfach gestaltet sein, dass sie möglichst wenig Energie verbraucht. So soll gewährleistet werden, dass der Fokus der Messung auf den zu untersuchenden Bestandteilen von Docker liegt. Das bedeutet gleichzeitig, dass der Energieverbrauch einer Software, wie zum Beispiel Apache oder nginx als Webserver oder MySQL sowie postgres als Datenbanksysteme, unabhängig gemessen werden müsste. Hierzu kann beispielsweise [Santos et al., 2018] befragt werden, obwohl die Messszenarien hier für unterschiedliche Software gleich sind, was zu einer geringen Vergleichbarkeit führt. Alle Ergebnisse wurden in fünf unabhängigen Messreihen validiert. Die dargestellten Ergebnisse entsprechen jeweils einer dieser fünf Messreihen. Alle weiteren Ergebnisse sind im Anhang der Ausarbeitung zu finden. Insgesamt fielen Ergebnisse aus mehr als zwanzig Stunden an Messungen an. Es ist zu erwähnen, dass alle Messungen auf Einzelsystemen und nicht auf verteilten Systemen stattfanden. Die Ergebnisse lassen sich jedoch in den meisten Fällen auf verteilte Systeme übertragen. Insgesamt soll die genaue Beschreibung der Messszenarien sowie der Last als Abgrenzung zu beispielsweise [Tadesse et al., 2017] und [Santos et al., 2018] dienen und somit die Messergebnisse einfacher reproduzierbar und in der Praxis nutzbar machen.
Im nachfolgenden Kapitel wird beginnend mit einer Analyse der Programmiersprache gestartet, welche die Grundlage von Docker bildet, um davon ausgehen zu können, dass Docker auf energiesparende Features setzt und keinen unnötigen Overhead mit sich bringt. Teile der im Folgenden präsentierten Ergebnisse finden sich auch in reduzierter Form in [Kreten, 2019] wieder.

7.1 Energieeffizienz von Docker

Um zu überprüfen, ob Docker auf energieeffiziente Programmierfeatures aufsetzen könnte, sollte die Programmiersprache Go, in welcher Docker programmiert ist, mit anderen Sprachen verglichen werden (vergleiche hierzu [Kreten and Guldner, 2017]). Das Herz von Docker, containerd, ist, bis auf einige Ausnahmen, in Go geschrieben und dient als Wrapper für low-level Runtimes wie runC oder kata. Sie verschleiert die sonst umständlichen Systemaufrufe mittels einer umfangreichen Go-Bibliothek [The Linux Foundation, 2019b]. Um zu zeigen, dass Docker auf effiziente Features setzt, wurde der Fokus beim Experiment auf die Nebenläufigkeitsfeatures der Programmiersprachen Go, C# und Clojure gelegt, da zur Verwaltung der Container intensiv mit Nebenläufigkeit gearbeitet wird. Go gilt in Fachkreisen als besonders geeignet für die Entwicklung der Cloud (siehe auch [Parbel,

2021b] und [Parbel, 2021a]), was ebenfalls eine Vielzahl an Entwicklungen in dieser Sprache im Kontext der Cloud zeigen. Go setzt im Bereich der Nebenläufigkeit auf den klassischen Mutex, ein Channel-System zur Kommunikation zwischen Threads, und auf atomare Operationen, um kritische Bereiche aufzulösen und Seiteneffekte zu verhindern.
Im Vergleich dazu steht C#, das ebenfalls mit für die Cloud geeigneten Features des .NET Frameworks aufwarten kann. Im Bereich der Nebenläufigkeit setzt C# auf das Task-System sowie auf die Schlüsselwörter *async* und *await*. Auch hier werden unter anderem atomare Operationen und Mutexe/Semaphoren verwendet. Als dritte Programmiersprache wird Clojure gewählt, die zum einen funktional ist, und zum anderen auf der Java Virtual Machine (JVM) ausgeführt wird. Damit bedient sie eine weitere Klasse von Programmiersprachen. Funktionale Programmiersprachen gewinnen immer mehr Anhänger (vergleiche Anzahl funktionaler Sprachen in den Top 50 des TIOBE Index [TIOBE, 2021]) und bieten sich aufgrund ihrer Nebenläufigkeitsfeatures auch für Anwendungen im skalierten Containerbereich an. So setzt Clojure auf Software Transactional Memory (Keywords dosync, alter, ref, etc.), die atomare und damit von Seiteneffekten unbeeinflusste Operationen auf Daten gewährleistet und dadurch Locking mit Mutexen oder Semaphoren überflüssig macht.

7.1.1 Messmethodik und Messszenario

Als Messaufbau wurde Iteration 1 aus Abschnitt 6.2.1 (Janitza Messgerät sowie voneinander getrennte SUT/WG und DAE) gewählt. Wichtig war es dabei, ein einfach zu replizierendes Messszenario zu wählen, welches in allen drei Programmiersprachen umzusetzen war. Hierfür wurde das bekannte Cigarette Smokers Nebenläufigkeitsproblem gewählt, welches sich folgendermaßen gestaltet:

i) An einem Tisch sitzen ein Dealer und drei Raucher

ii) Weiter gibt es drei Bestandteile, um eine Zigarette herzustellen und zu rauchen: Papier, Tabak und Streichhölzer

iii) Jeder Raucher besitzt einen unendlichen Vorrat an jeweils genau einer Zutat

iv) In einem festgelegten Abstand fügt der Dealer zwei zufällige Zutaten zum frei zugänglichen Vorrat der Zutaten hinzu

v) Die Raucher versuchen die zwei Zutaten vor den anderen Rauchern vom Vorrat zu nehmen und eine Zigarette herzustellen sowie zu rauchen (kritischer Abschnitt)

vi) Während des Rauchens kann ein Raucher keine weiteren Zutaten vom Vorrat nehmen

Diese Komponenten des Problems werden mit Hilfe der zur Verfügung stehenden Nebenläufigkeitsfeatures in allen drei Sprachen umgesetzt. Ziel ist es dabei, jeden kritischen Abschnitt

vollständig und ohne Fehler aufzulösen. Beim Aufbau der Messung wird sich insgesamt an den groben Aufbau aus Abschnitt 6.2.4 gehalten. Um eine möglichst hohe Anzahl an Arbeit zu generieren, werden drei Dealer und jeweils 500 Raucher des gleichen Typs erzeugt. Danach wird das Programm zehn Mal für genau 180 Sekunden gestartet. Zwischen jedem Messdurchlauf findet eine dreißigsekündige Pause statt. Die Pause während des Rauchens beträgt 100 Millisekunden.
Auch wird das System zu Beginn für zehn Minuten im Leerlauf gemessen, um durch Bildung der Differenz den genauen Energieverbrauch der Software zu bestimmen. Dabei wird darauf geachtet, dass alle möglichen Hintergrundprogramme am SUT ausgeschaltet sind.
Folgende Daten werden während eines Messdurchlaufs erhoben:

i) CPU Auslastung in Prozent

ii) RAM Auslastung in Prozent

iii) Energieaufnahmen in Watt

iv) Anzahl gerollter Zigaretten (Anzahl Kollisonsauflösungen)

Im nachfolgenden Kapitel werden die entsprechenden Resultate der Messungen dargestellt und interpretiert.

7.1.2 Darstellung der Messergebnisse

Die in Tabelle 4 gezeigten durchschnittlichen Leistungsaufnahmen sind von der grundsätzlichen Leistungsaufnahme des Testsystems im IDLE befreit. Demnach ist gut zu erkennen, dass C# die geringste CPU-Auslastung und die geringste RAM-Auslastung sowie die geringste Leistungsaufnahme der drei getesteten Sprachen besitzt. Clojure hingegen ist hier in allen drei Fällen am ineffizientesten.

	C#	Go	Clojure
durchschnittliche CPU-Auslastung	0,97	2,96	3,13
durchschnittliche RAM-Auslastung	15,45	15,99	19,44
durchschnittliche Leistungsaufnahme in Watt	7,81	8,67	9,0
durchschnittliche Anzahl an Kollisionsauflösungen	6044	7066	6377

Tabelle 4: Ergebnisübersicht der Messung bezüglich des Energieverbrauchs der Sprachen C#, Go und Clojure (vgl. Kreten and Guldner [2017])

Daraus ergibt sich der in Tabelle 5 dargestellte Energieverbrauch bezüglich der drei getesteten Sprachen
Betrachtet man jetzt jedoch die Anzahl der Kollisionsauflösungen im Testintervall, bedingt durch den jeweiligen Umgang mit Nebenläufigkeit, so ergibt sich ein anderes Bild, wenn man diese in Relation zum Energieverbrauch setzt und damit die funktionale Einheit „Kollisionsauflösungen pro Ws“ bildet.

	C#	Go	Clojure
Verbrauchte Energie im Testintervall in Ws	1405,8	1560,6	1620

Tabelle 5: Energieverbrauch der Sprachen C#, Go und Clojure im Testintervall (vgl. Kreten and Guldner [2017])

	C#	Go	Clojure
Kollisionsauflösungen pro Ws	4,3	4,53	3,9

Tabelle 6: Kollisionsauflösungen pro Ws der Sprachen C#, Go und Clojure im Testintervall (vgl. Kreten and Guldner [2017])

Mit dieser funktionalen Einheit, abgebildet in Tabelle 6, zeigt sich, dass Go die effizienteste der drei Sprachen bezüglich der Kollisionsauflösung ist.

7.1.3 Interpretation der Ergebnisse

Wie im letzten Abschnitt gezeigt, hat Go einen im Vergleich zu anderen Sprachen effizienteren Umgang mit Nebenläufigkeit. Somit bildet Go eine gute Ausgangsbasis und entsprechende Voraussetzungen, um Docker energieeffizient gestalten zu können. Es ist hier jedoch zu beachten, dass eine Vielzahl weiterer Faktoren, unabhängig von der Nebenläufigkeit, Auswirkungen auf die Effizienz der Programmierung haben können. Zudem ist zu beachten, dass die Testprogramme in ihrer Programmierung bereits optimiert waren. Setzt man bei Docker nur auf Channels und nicht auf klassische Locks, so sinkt die Anzahl an verarbeiteten Kollisionen stark, während der Energieverbrauch steigt. Es ist hier wichtig, stets den Anwendungsfall der einzelnen Features zu beachten und entsprechend abzuschätzen.

7.2 Untersuchung der Kernkomponenten von Docker

Im Folgenden werden die in Abschnitt 3.3.2 vorgestellten Bestandteile von Docker hinsichtlich ihres Energieverbrauchs untersucht. Wie in Unterkapitel 5.2 aufgezeigt, wird Docker in den meisten Fällen, und besonders unter Entwicklern, in den Standardeinstellungen verwendet. Daher soll die nachfolgende These in diesem Kapitel belegt werden:

> Die Standardeinstellungen von Docker führen nicht zum optimalen Stromverbrauch der Container. Durch die Anpassung dieser Einstellungen ergibt sich ein hohes Einsparpotential.

Außerdem soll folgende Aussage durch den Vergleich unterschiedlicher Messszenarien verdeutlicht werden:

> (Energie-) Messungen im Bereich von Container lassen sich, aufgrund der im Container laufenden Software, nur schwer verallgemeinern.

Die Ergebnisse werden kurz dargestellt sowie interpretiert. Es ist zu erwähnen, dass manche Messreihen in Gänze in Form von Tabellen zum besseren Vergleich dargestellt werden, während auf manche Ergebnisse nur prosaisch eingegangen wird. Alle Messergebnisse in Form der Messreihen werden im Anhang zu finden sein.
Die Ergebnisse dieses Kapitels dienen in Kapitel 8 zur Aufstellung von Handlungsempfehlungen im Sinne einer energieeffizienten Nutzung von Docker. Darüber hinaus wird eine Simulation die Einsparmöglichkeiten durch Nutzung der Handlungsempfehlungen an einem reduzierten realistischen Beispiel zeigen.

7.2.1 Docker im IDLE

In diesem Abschnitt wird deutlich werden, wie sich Docker im IDLE verhält und ob bereits durch die Installation von Docker ein energetischer Overhead existiert.

7.2.1.1 Messszenario und Messmethodik

Um zu überprüfen, ob Docker bereits im IDLE (ohne Last auf die im Container befindliche Software) Energie verbraucht, wurde Ubuntu 18.04 auf dem in Abschnitt 6.2.1 vorgestellten Udoo x86 neu installiert und auf den aktuellsten Stand geupdatet. Das Betriebssystem wurde dabei in einem minimalen Stand installiert, der keinerlei Drittanbieter-Software oder einen Desktopmanager enthält. Anschließend wurde die energetische Baseline des Systems in 20 einminütigen Messrunden bestimmt, die jeweils von einer einminütigen Pause getrennt wurden. Um Seiteneffekte zu verhindern, wurde das System vom Internet abgekoppelt. Im zweiten Schritt wurde Docker auf dem System installiert und die Baseline-Messung wurde wiederholt.

7.2.1.2 Darstellung der Messergebnisse

Sowohl mit als auch ohne Docker verbraucht das System 4,45 Watt im Mittel. Des Weiteren ergibt sich bei beiden Messreihen eine Standardabweichung von unter 0,03 Watt. In Abschnitt 7.2.2 wird exemplarisch eine Baseline-Messung mit laufenden Docker-Prozessen präsentiert, um diese zu weiteren Vergleichen zu nutzen. Aus diesem Grund wird an dieser Stelle auf eine tabellarische Darstellung der Messergebnisse verzichtet.

7.2.1.3 Interpretation der Ergebnisse

Es kann festgehalten werden, dass Docker im IDLE keine System-Ressourcen und demnach auch keine weitere Energie verwendet. Somit kann davon abgesehen werden, den Service auszuschalten, sofern dieser nicht verwendet wird.

7.2.2 Container im IDLE

Wie in Abschnitt 7.2.1 gezeigt, besteht kein erkennbarer Unterschied zwischen einem System, auf dem Docker nicht läuft und einem System, bei dem der Service *dockerd* aktiviert ist.
In diesem Kapitel wird eine ähnliche Überprüfung präsentiert, die sich jedoch nicht mit dem Grundsystem, sondern mit der Leistungsaufnahme von Docker-Containern beschäftigt. Besonders hier wird ein großer Wert auf die Auswahl der Messszenarien gelegt, die im nachfolgenden Abschnitt kurz dargestellt werden.

7.2.2.1 Messszenarien und Messmethodik

Im Vorhinein war anzunehmen, dass zwar Container selbst für einen energetischen Overhead sorgen, so wie beispielsweise in [Santos et al., 2018] gezeigt, die Software, die im Container läuft, jedoch einen erheblichen Anteil am Energieverbrauch hat. Aus diesem Grund reicht es nicht aus, so wie in [Santos et al., 2018], den Stromverbrauch von unter Last stehender Software in Containern mit denen von Software zu vergleichen, die nicht containerisiert ist. Daher werden in diesem Kapitel zwei Arten von containerisierter Software bezüglich ihrer Lastaufnahme im IDLE-Zustand verglichen. Es wurde sich an dieser Stelle für den Webserver Nginx und die Datenbank MySQL entschieden, deren Basis-Images auf Docker Hub zu den meist heruntergeladenen zählen (beide mehr als zehn Millionen Mal, vgl. [Docker Inc., 2021g]).
Um die Messungen durchzuführen, wurde Iteration 3 des in Abschnitt 6.2.1 dargestellten Messaufbaus und das Messskript aus Abschnitt 6.2.3 verwendet.
Eine Messrunde dauert eine Minute und wurde in diesem Fall zwanzig Mal wiederholt. Zwischen jeder Messrunde findet wiederum eine einminütige Pause statt.

7.2.2.2 Darstellung der Messergebnisse

Wie in Tabelle 7 zu erkennen, liefert das Messverfahren Ergebnisse mit sehr geringer Standardabweichung von maximal 4%. Es ist gleichermaßen klar erkennbar, dass es keine Unterschiede in der Leistungsaufnahme zwischen der System-Baseline und nginx gibt. Im Gegensatz dazu weist der MySQL-Container im IDLE eine um 1,5% höhere Leistungsaufnahme als die Systembaseline auf. Darüber hinaus lieferten die Messungen Ergebnisse zur CPU- und RAM-Auslastung der Container. Während Nginx eine null-prozentige CPU-Auslastung und eine kaum erkennbare RAM-Auslastung in Höhe von 0,06% hat, verbraucht MySQL im IDLE durchschnittlich bereits 1,8% der CPU und hat eine RAM-Auslastung von 9,68% .

7.2.2.3 Interpretation der Ergebnisse

Wie an den Ergebnissen zu sehen, ist keine Verallgemeinerung über die Leistungsaufnahme von Containern im IDLE möglich. Diese wird nämlich durch die Software im Container bestimmt. Trotzdem entsteht durch die Messungen ein Mehrwert bezüglich des Energieverbrauchs. Es gibt Container, die

	System-Baseline in Watt	IDLE nginx Container in Watt	IDLE MySQL Container in Watt
1	4,4736	4,4578	4,5151
2	4,4568	4,4583	4,5216
3	4,4506	4,4481	4,5222
4	4,4758	4,4578	4,5206
5	4,4553	4,4257	4,5015
6	4,4259	4,4507	4,506
7	4,5200	4,4518	4,516
8	4,4710	4,4418	4,5289
9	4,4518	4,4282	4,5164
10	4,4619	4,459	4,5278
11	4,4594	4,4502	4,5185
12	4,4268	4,4461	4,5074
13	4,4323	4,4568	4,5225
14	4,4436	4,4663	4,525
15	4,4676	4,4647	4,513
16	4,4562	4,4625	4,5339
17	4,4555	4,4453	4,5338
18	4,4609	4,4802	4,5517
19	4,4545	4,4543	4,5387
20	4,4518	4,478	4,5385
Mittelwert	4,457	4,454	4,523
Standardabweichung	0,02	0,014	0,012

Tabelle 7: Darstellung der durchschnittlichen Leistungsaufnahme von Testdurchläufen je 1 Minute

mit der Aussicht auf Wiederverwendung nicht gelöscht werden müssen (so wie ein Nginx Container), da sie im IDLE keine zusätzliche Energie verbrauchen. Da wie in Abschnitt 7.2.4 zu sehen ist, auch das Starten und Löschen von Containern einen erheblichen energetischen Overhead hat, kann auf diese Weise bereits Energie eingespart werden. Im Sinne der Handlungsempfehlungen kann aus diesem Grund bereits festgehalten werden, dass es sich in Abhängigkeit zur zum Einsatz kommenden Software lohnt, frühzeitig zu entscheiden, ob Container weiterverwendet werden und daher am Laufen gehalten werden sollten oder ob diese gestoppt werden können.

7.2.3 Docker Build-Prozesse für Images

Wie im letzten Kapitel gezeigt, können Container bereits im IDLE für eine höhere Leistungsaufnahme des Systems sorgen. Doch der Weg hin zum laufenden Container kann bereits Energie verbrauchen beziehungsweise verschwenden. Der Prozess zur Erzeugung von Images, mit denen wiederum Container erzeugt werden, besitzt eine Vielzahl von Anpassungsmöglichkeiten. Nachfolgend werden exemplarisch sechs Möglichkeiten zur Anpassung untersucht.

7.2.3.1 Messszenario und Messmethodik

Um die Auswirkungen der Anpassungsmöglichkeiten des Build-Prozesses von Images zu untersuchen, wurde folgende Dockerfile definiert, aus der ein Image erzeugt werden soll:

```
FROM golang

WORKDIR /app
COPY server.go .
RUN go build -o server .
EXPOSE 8080

CMD ./server
```

Listing 9: Messzenario beschreibende Dockerfile

Im Dockerfile wird eine Datei *server.go* zum Image, basierend auf dem offiziellen golang-Image, hinzugefügt und danach kompiliert. Dann wird der Port 8080 nach außen freigegeben, um abschließend dort den durch die Go-Datei beschriebenen Server erreichbar zu machen. Der Server selber wird durch den letzten Befehl des Dockerfiles gestartet. Hier wurde bewusst eine Datei gewählt, die während der Erzeugung des Images kompiliert werden soll. Dieses Vorgehen wird häufig angewandt, um Compiler, Linker oder Ahnliches nicht auf dem Host-System installieren zu müssen. Dadurch wird unter anderem auch die Portabilität der Software gesteigert, da alle notwendigen Abhängigkeiten im Container zu finden sind. Die Dockerfile wird nachfolgend drei Mal angepasst: In einem ersten Schritt werden die Auswirkungen auf die Ausführungszeit und die Leistungsaufnahme durch Verwendung eines kleinen Basis-Images untersucht. Hierzu wird die Dockerfile wie folgt angepasst:

```
FROM golang:1.11-alpine3.10

WORKDIR /app
COPY server.go .
RUN go build -o server .
EXPOSE 8080

CMD ./server
```

Listing 10: Messzenario beschreibende Dockerfile

Das auf Alpine Linux basierende Basis-Image ist 312 MB groß, während das zuvor verwendete Image 775 MB groß ist.
Eine zweite Veränderung des Dockerfiles äußerte sich wie folgt:

```
FROM golang:1.11-alpine3.10

WORKDIR /app
COPY server.go .
ARG CGO_ENABLED=0
ARG CACHEBUST=no
RUN go build -o server .
EXPOSE 8080

CMD ./server
```

Listing 11: Messzenario beschreibende Dockerfile

In dieser Anpassung wird ein Argument CACHEBUST hinzugefügt. Dieses wird während des Build-Prozesses willkürlich verändert, um so die Cache-Funktionen von Docker auszuhebeln und ab bestimmten Stellen in der Dockerfile zu deaktivieren. Genaueres dazu wird im nachfolgenden Kapitel dargestellt.
Insgesamt ist es wichtig zu beachten, dass hier sehr einfache Dockerfiles verwendet werden. Steigt die Komplexität einer Dockerfile an, so verlängert sich zum einen die Ausführungszeit, als auch der Energieverbrauch.
Die Messungen fanden mit einem angepassten Messaufbau der 3. Iteration, wie in Abschnitt 6.2.1 beschrieben, statt. Das Messskript aus Abschnitt 6.2.3 wurde leicht angepasst, damit die Dauer der Messung an die Dauer des Build-Prozesses geknüpft wird. Weiter war natürlich keine Lasterzeugung durch einen WG notwendig. Jede Messung wurde zwanzig Mal durchgeführt. Dabei wurde zwischen jeder Messrunde eine dreißigsekündige Pause eingehalten. Zusammenfassend sieht das Messszenario wie folgt aus:

i) Container wird mit Hilfe einer Dockerfile gebaut

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
1	17,82	6,853
2	17,61	6,792
3	17,53	6,815
4	17,65	6,853
5	17,68	6,877
6	17,72	6,862
7	17,76	6,877
8	17,65	6,894
9	17,52	6,888
10	17,54	6,879
11	17,59	6,877
12	17,53	6,883
13	17,68	6,889
14	17,51	6,889
15	17,69	6,889
16	17,60	6,894
17	17,72	6,901
18	17,65	6,902
19	17,72	6,901
20	17,58	6,907
Mittelwert	17,64	6,876
Standardabweichung	0,087	0,03

Tabelle 8: Leistungsaufnahme und Dauer eines default Docker Build-Prozesses ohne Cache

ii) Während des Builds wird ein Go Programm kompiliert

iii) Verschiedene Einstellungen werden dabei durchgetestet (Default, Buildkit, Multistage, Cache/kein Cache)

iv) Dauer des Build-Prozesses und durchschnittliche Leistungsaufnahme werden überprüft

7.2.3.2 Darstellung der Messergebnisse

In Tabelle 8 ist eine Messreihe mit zwanzig Messrunden eines Docker Build-Prozesses zu sehen, in welcher die erste Dockerfile, die im letzten Abschnitt beschrieben wurde, verwendet wurde, um ein neues Image zu erzeugen. Da hier nicht auf den Cache zurückgegriffen werden soll, wird nach jedem Testdurchlauf das neu erzeugte Image gelöscht.

Erneut ist zu erkennen, dass das Messverfahren sehr gut wiederholbar ist, da die Standardabweichungen der Messungen sowohl für die Ausführungszeit als auch für die Leistungsaufnahme geringer als 0,5% sind.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
1	1,993	5,452
2	2,028	5,472
3	2,011	5,483
4	2,009	5,412
5	2,013	5,543
6	1,972	5,552
7	1,991	5,566
8	2,049	5,579
9	2,022	5,596
10	1,989	5,611
Mittelwert	2,008	5,533
Standardabweichung	0,022	0,06

Tabelle 9: Leistungsaufnahme und Dauer eines Docker Build-Prozesses mit der Option –cache-from

Im Folgenden wird aus Platzgründen auf die Darstellung ganzer Messreihen verzichtet. Stattdessen werden wie in Tabelle 11 nur die Ergebnisse der Messungen dargestellt. Die zwei nachfolgenden Tabellen zeigen zur Verdeutlichung der Reproduzierbarkeit der Messungen jeweils zehn Messrunden. In Tabelle 9 ist eine Messreihe dargestellt, in der bereits ein Image existiert, sodass alle weiteren Build-Prozesse auf den Cache zurückgreifen können. Die Darstellung ist hier zur Platzersparnis auf zehn Messungen reduziert.

Es ist zu erkennen, dass die Laufzeit des Build-Prozesses nun nur noch 11,4% des vorherigen Prozesses beträgt. Ebenfalls ist zu erkennen, dass die Leistungsaufnahme im Mittel um 20% gefallen ist. Dies ergibt eine Energieersparnis von durchschnittlich 90% pro Build-Vorgang (es ist zu beachten, dass die Ausführungszeit und der Energiebedarf bei komplexeren Dockerfiles ansteigen). Allerdings ergibt sich für den Nutzer hierbei ein Problem: Die im Dockerfile angegebene Kompilierung der Go-Datei wird im Falle des Caches nicht genutzt. Hier wird lediglich das Image neu erzeugt. Es ist auch zu bedenken, dass die Nutzung des Caches per default angenommen wird. Um nun dafür zu sorgen, dass sowohl der Cache verwendet, als auch die Kompilierung des Go-Skripts neu ausgeführt wird, kann man das Prinzip des Cache-Bustings verwenden, wie in Listing 11 dargestellt. Hier wird durch eine automatisierte Umformulierung der Dockerfile an der gewünschten Stelle (in der Dockerfile markiert durch das Auslesen der Systemvariablen CACHEBUST) der neue Kompilierungs-Prozess angestoßen. Alle anderen Befehle vor dieser Maßnahme kommen weiterhin aus dem Cache. Hierfür ergibt sich der Verbrauch wie in Tabelle 10 zu erkennen.

Durch Cachebusting lässt sich zwar keine hohe Ersparnis wie im Falle der reinen Cache-Nutzung herbeiführen, jedoch behält der Prozess seine volle Funktionalität. Insgesamt kann so trotzdem eine Energieersparnis von circa 10% erreicht werden.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
1	15,973	6,939
2	15,959	6,906
3	16,21	6,917
4	15,98	6,93
5	15,83	6,936
6	16,027	6,934
7	16,153	6,94
8	16,093	6,937
9	16,139	6,936
10	16,03	6,942
Mittelwert	16,04	6,93
Standardabweichung	0,1	0,01

Tabelle 10: Leistungsaufnahme und Dauer eines Docker Build-Prozesses mit explizitem Cachebusting

Zusätzlich zu diesen Messungen wurde der Unterschied zwischen kleinen und größeren Basis-Images bezüglich der Leistungsaufnahme untersucht. Beide Basis-Images müssen jedoch den selben Funktionsumfang bieten. Für diese Untersuchung bieten sich offizielle Images aus Docker Hub an, die auf unterschiedlichen Linux-Kernels basieren. Als besonders schlank gelten dabei Images, die auf Alpine Linux basieren. Wie in Tabelle 11 zu sehen, ist der Unterschied minimal und liegt im getesteten Fall bei circa 3%. Zusätzlich zu den soeben gezeigten Anpassungsmöglichkeiten besteht, wie bereits

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
golang	17,64	6,876
golang:1.11-alpine3.10	18,07	6,89

Tabelle 11: Unterschiede zwischen einem großen und einem kleineren Basis-Image ohne Cache gemäß Listing 9 und Listing 10

in Absatz 3.3.2.2 beschrieben, die Möglichkeit, mehrstufige Build-Prozesse auszuführen, um so beispielsweise Build-Dependencies zu entfernen. Gerade in CI/CD-Pipelines wird ein solches Vorgehen häufig angewendet, da man so eine „reine“ Applikation ausliefert. Die Dockerfile muss dann entsprechend Listing 12 angepasst werden.

```
FROM golang AS builder

WORKDIR /app
COPY server.go .
ARG CGO_ENABLED=0
```

```
RUN go build -o server .

FROM alpine

WORKDIR /app
COPY --from=builder /app/server .
EXPOSE 8080
CMD ./server
```

Listing 12: Messzenario beschreibende Dockerfile

Durch die Verwendung von zwei Images wird mit Hilfe von Image 1 basierend auf golang, nur das Go-Programm kompiliert. Image 2 erhält dann die ausführbare Datei, wodurch das resultierende Image kleiner ist als in oben gezeigten Fällen. Der Nachteil hierbei ist jedoch, dass eben zwei Build-Prozesse angestoßen werden, was sich sowohl in der Laufzeit als auch in der Leistungsaufnahme zeigt. Wie in

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Build gemäß Listing 9	17,64	6,876
Multistage-Build gemäß Listing 12	22,65	6,71

Tabelle 12: Unterschiede zwischen Build gemäß Listing 9 und einem Multistage-Build

Tabelle 12 zu erkennen, verlängert sich im Multistage die Ausführungszeit deutlich. Im Beispiel handelt es sich um circa 28%. Der Energieverbrauch sinkt dabei leicht, da zwischen dem ersten und dem zweiten Build eine kurze Pause entsteht, in der die Leistungsaufnahme deutlich absinkt. Insgesamt kommt es durch die steigende Laufzeit zu einer Erhöhung des Energieverbrauchs. Während im ersten Fall 121 Ws verbraucht werden, steigt der Verbrauch beim Multistage-Build auf 151 Ws und somit um 25%.

Zusätzlich zu den bisher vorgestellten Anpassungen des Build-Vorgangs, die in der Praxis häufig so angewendet werden, kommt mit Docker 18.09 eine weitere Anpassungsmöglichkeit des Build-Prozesses hinzu. Mit Hilfe der Option *DOCKER_BUILDKIT=1* soll laut [Docker Inc., 2019b] eine Verbesserung der Performance des Build-Prozesses herbeigeführt werden. Nutzt man diese Option beim Start des Build-Prozesses, ergeben sich Messwerte entsprechend Tabelle 13.

Wie zu erkennen ist, sinkt die Laufzeit deutlich um 4 Sekunden und damit um circa 26%. Entgegen dessen steigt jedoch der Stromverbrauch leicht um 4% an. Dies ist ein Resultat der besseren CPU-Ausnutzung von Buildkit, was im Gegenzug zur erhöhten Geschwindigkeit führt. Insgesamt ergibt sich trotzdem eine Energieersparnis von circa 23%.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Build gemäß Listing 9	17,64	6,876
Build gemäß Listing 9 mit Buildkit	13,06	7,17

Tabelle 13: Unterschiede zwischen Build gemäß Listing 9 und einem Build mit Buildkit

7.2.3.3 Interpretation der Ergebnisse

Im Allgemeinen kann festgehalten werden, dass Buildkit die Ausführungszeit deutlich verbessert. Multistage-Builds sollten aus energetischer Sicht nur angewendet werden, wenn eine Verhinderung von Build Dependencies wichtig ist und einen Mehrwert für die Stabilität der Applikation mit sich bringt.
Sollten häufig ein und dieselben Build-Prozesse ausgeführt werden, ist die Nutzung des Cache unabdingbar. Muss dabei Software innerhalb eines Container gebaut, komprimiert oder kompiliert werden, sollte stets auf Cachebusting zurückgegriffen werden, um nicht den gesamten Cache ausschalten zu müssen.

7.2.4 Start-, Stopp- und Löschvorgang von Containern

Der Container-Lifecycle umfasst, wie bereits in Abschnitt 3.4.3 gezeigt, alle Stadien eines Containers vom Build des Images über die Erzeugung, das Starten und Stoppen (docker stop) hin zu Updates und final zum Löschen des Containers. Die Zeit zwischen zwei Updates beläuft sich dabei in zwei Drittel aller Fälle auf 7 Tage, sodass 95% aller Images weniger als eine Woche laufen, bevor sie gelöscht und neu aufgesetzt werden [sysdig.com, 2021][datadoghq.com, 2018]. Aber auch zwischen diesen Updatephasen kommt es durch die Integration neuer Features zu häufigem Löschen und Neustart von Containern, wie zum Beispiel im Fehlerfall oder bei temporärer Abschaltung eines Services. Dabei stellt sich die Frage, ob Starten (docker container start: startet einen vorhanden Container) und Stoppen (docker container stop: stoppt einen Container, aber löscht ihn nicht) oder Neustart (docker run: erstellt einen neuen Container mit entsprechenden Eigenschaften und startet ihn) und Löschen (docker container rm: löscht einen gestoppten Container) von Containern effizienter sind (vergleiche Absatz 3.3.2.2). Neben dieser Frage soll in den nachfolgenden Abschnitten geklärt werden, ob der Start/Neustart eines Containers mit *docker run* oder docker-compose energieeffizienter ist (vergleiche hierzu ebenfalls Absatz 3.3.2.2).

7.2.4.1 Messszenario und Messmethodik

Als Vorlage für den Messaufbau dient jener aus Abschnitt 7.2.3. Demnach wird auch hier die Dauer des Prozesses und die durchschnittliche Leistungsaufnahme im entsprechenden Zeitintervall untersucht. Erzeugt werden die Container ebenfalls entsprechend der Images, die in Abschnitt 7.2.3 vor-

gestellt wurden und einen Go-Server beinhalten. In einem ersten Experiment wird ein Container mit folgendem Befehl erzeugt:

```
docker run -d --name go_server -p 8080:8080 go_server
```

Listing 13: Erzeugen und Starten eines Containers

Auf diese Weise werden alle für den Betrieb des Servers notwendigen Eigenschaften des Containers festgelegt. Nachfolgend dazu wird der Container mit dem Stopp-Befehl

```
docker container rm -f go_server
```

Listing 14: Löschen eines gestoppten oder laufenden Containers

gelöscht. In beiden Fällen werden die Befehle an den Docker-Host über einen Remote-Call und die Zeit vom Absetzen des Befehls bis hin zur Antwort des Docker-Host gemessen. Ab diesem Moment steht der Docker Container bereit und kann beispielsweise betreten werden. Dies bedeutet jedoch nicht, dass die Software innerhalb des Containers ebenfalls bereitstehen muss. Zwischen dem Erzeugen und dem Löschen des Containers wird eine Pause von Sekunden durchgeführt. Weiter findet zwischen den Messrunden eine Pause von Sekunden statt. Insgesamt umfasst jedes Experiment 20 Messrunden.
Im zweiten Experiment wird ein bereits erzeugter Container mit Hilfe des Befehls

```
docker container start go_server
```

Listing 15: Starten eines Containers.

gestartet. Der grundsätzliche Aufbau des ersten Experiments wird hierbei beibehalten. Am Ende wird der Container hier jedoch nicht gelöscht, sondern nur gestoppt.
Abschließend wird in einem dritten Experiment bestimmt, ob die Erzeugung und der Start mit Hilfe von Docker-Compose effizienter als die Nutzung von *docker run* ist. Der Aufbau bleibt konsistent zu den vorherigen Experimenten, gemessen wird hierbei jedoch die Ausführdauer und durchschnittliche Leistungsaufnahme des folgenden Befehls:

```
docker-compose up -d
```

Listing 16: Starten eines Containers mit dem Befehl docker-compose

Für die Ausführung von Docker-Compose ist die nachfolgende Datei notwendig:

```
version: '3.7'
services:
  server:
    image: go_server
```

```
    ports:
      - "8080:8080"
```

Listing 17: Aufbau der Compose-Datei

Wie in Listing 17 zu sehen ist, wird das gleiche Image wie in den vorhergehenden zwei Experimenten verwendet, um diese entsprechend vergleichen zu können. Da Docker-Compose per Default ein eigenes Netzwerk für die Container erzeugt, wird dasselbe Experiment noch einmal für die folgende Compose-Datei durchgeführt:

```
version: '3.7'
services:
  server:
    image: go_server
    ports:
      - "8090:8080"
    networks:
      - go-api-own

networks:
  go-api-own:
     external: true
```

Listing 18: Aufbau der Compose-Datei mit externem Netzwerk

Hierbei wird unter der Voraussetzung, dass das Netz *go-api-own* bereits im Vorhinein erzeugt wurde, auf die Erzeugung eines weiteren Netzes verzichtet.
Um einen Vergleich zu einem anderen Messszenario zu haben und auf die Stärken von Compose, mehrere Container auf einmal zu erzeugen, eingehen zu können, wird abschließend das vierte Experiment mit einer größeren Compose-File, die in Listing 19 zu sehen ist, wiederholt.

```
version: '3.7'
services:
  server:
    image: goapi
    ports:
      - "9000:9000"
    depends_on:
      - db

  db:
    image: mysql
    command: --default-authentication-plugin=mysql_native_password
    restart: always
```

```
    environment:
      MYSQL_ROOT_PASSWORD: 123456
```

Listing 19: Aufbau der Compose-Datei mit Server und Datenbank

In diesem Messszenario wird zum einen ein Container für einen Go-Server bereitgestellt, der eine API ausliefert und zum anderen ein MySQL-Container, in welchem die über die API erreichbaren Daten liegen. Nachfolgend an die Messung von Docker-Compose werden beide Container mit zwei direkt hintereinander ausgeführten run-Befehlen erzeugt (siehe Listing 20). Erneut werden die mittlere Dauer der Container-Erzeugung und die durchschnittliche Leistungsaufnahme zwischen Compose und run-Befehl verglichen.

```
docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=123456 -d mysql
docker run -d --name go_api_server -p 9000:9000 goapi
```

Listing 20: Starten eines Servers und einer Datenbank in zwei Containern

7.2.4.2 Darstellung der Messergebnisse

In Tabelle 14 und Tabelle 15 sind die Ergebnisse von *docker run* und *docker rm* dargestellt. Wie zu erkennen ist, dauern beide Prozesse im entsprechenden Messszenario 1,52 Sekunden. Entsprechend dessen verbraucht die Erzeugung des Containers 8,13 Ws und das Löschen 8,2 Ws.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	1,52	5,35
Standardabweichung	0,01	0,1

Tabelle 14: Erzeugung und Start eines Containers mit *docker run*

Aufsummiert ergibt dies 16,5 Ws. Wie weiter unten zu sehen ist, ist die Ausführungsdauer abhängig vom Messszenario.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	1,52	5,4
Standardabweichung	0,02	0,08

Tabelle 15: Löschen eines Containers mit *docker container rm*

Im angegebenen Messszenario dauert der Startprozess, wie in Tabelle 16 zu sehen, 3,9 Sekunden und nimmt durchschnittlich 5,6 W an Leistung auf. Das Stoppen wiederum dauert im Mittel 1,5 Sekunden

mit einer durchschnittlichen Leistungsaufnahme von 5,22 W. Somit verbraucht der Startprozess im angegebenen Fall 22,9 Ws und das Stoppen 7,83 Ws, was summiert einen Verbrauch von 30,7 Ws ausmacht.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	3,9	5,88
Standardabweichung	0,02	0,03

Tabelle 16: Start eines zuvor erzeugten beziehungsweise gestoppten Containers mit *docker container start*

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	1,51	5,22
Standardabweichung	0,02	0,06

Tabelle 17: Stoppen eines laufenden Containers mit *docker container stop*

Zwar stellen *docker run* und *docker start* nicht in Gänze dieselbe Funktionalität zur Verfügung, jedoch beinhaltet der run-Befehl das Starten des Containers. Dementsprechend führt im besprochenen Messszenario die ausschließliche Nutzung von run- um rm-Befehlen zu einer Verringerung des Energieverbrauchs von circa 47%.
Verwendet man anstatt des run-Befehls Docker-Compose, welches sich im Besonderen für den gleichzeitigen Start mehrerer Container eignet, so ergeben sich entsprechend des angelegten Szenarios die Ergebnisse aus Tabelle 18.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	6,6	6,15
Standardabweichung	0,03	0,03

Tabelle 18: Erzeugung und Start eines Containers mit *docker-compose up -d*

Um einen Container wie in Tabelle 14 zu erzeugen (gleiches Image, gleiche Funktionalität), verstreichen bei Docker-Compose im Mittel 6,6 Sekunden mit einer durchschnittlichen Leistungsaufnahme von 6,14 W. Im Vergleich zum run-Befehl wird somit 400% mehr Energie verbraucht.
Um auszuschließen, dass diese Steigerung an der Erzeugung des Netzwerks für die Container liegt, wurde, wie vorher beschrieben, Compose mit einem zuvor angelegten Netzwerk gestartet. Wie in Tabelle 19 zu sehen ist, verringert sich die Dauer der Erzeugung und das Starten des Containers um circa 0,3 Sekunden. Auch bei der Leistungsaufnahme ist eine leichte Verbesserung zu erkennen. Insgesamt

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	6,36	6,07
Standardabweichung	0,05	0,03

Tabelle 19: Erzeugung und Start eines Containers mit *docker-compose up -d* ohne Erzeugung eines Netzwerks

ergibt sich im Vergleich zum standardmäßigen Compose eine Verbesserung von circa 5%. Damit wird immer noch circa 370% mehr Energie verbraucht als beim run-Befehl.
Abschließend sind in Tabelle 20 die Ausführdauer und die Leistungsaufnahme von Compose bezüglich eines API-Servers und einer Datenbank dargestellt, die mit der multiplen Ausführung von run-Befehlen verglichen werden. Es ist zu erkennen, dass zusätzlich zur Dauer auch die Leistungsaufnahme ansteigt, sodass im Mittel 62,5 Ws an Energie verbraucht werden, bevor beide Container zur Verfügung stehen.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	9,47	6,6
Standardabweichung	0,08	0,1

Tabelle 20: Erzeugung und Start eines Go-API-Servers und einer MySQL-Datenbank mit *docker-compose up -d*

Im Fall von zwei hintereinander ausgeführten run-Befehlen (siehe Tabelle 21) werden im Durchschnitt 52,3 Ws an Energie verbraucht. Im Vergleich zu Compose kann somit selbst bei mehreren Containern circa 27% an Energie gespart werden.

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	7,8	6,7
Standardabweichung	0,1	0,04

Tabelle 21: Erzeugung und Start eines Go-API-Servers und einer MySQL-Datenbank mit zwei hintereinander ausgeführten *docker run* Befehlen

7.2.4.3 Interpretation der Ergebnisse

Interpretiert man die Ergebnisse aus den vorhergegangenen Abschnitten, so kann beginnend festgehalten werden, dass die durchaus verbreitete Policy, Container nicht zu stoppen und zu behalten, sondern nach der Nutzung gleich zu löschen und bei Bedarf neu zu starten, einen positiven Einfluss auf den

Energieverbrauch hat. Ebenso ist festzuhalten, dass Compose in den getesteten Szenarien stets länger dauerte als der standardmäßige run-Befehl von Docker, was zu einer geringeren Effizienz von Compose führte. Besonders im Falle eines Containers ist vom Einsatz von Compose abzuraten. Hierbei ist zu erwähnen, dass man in öffentlichen Registries häufig Compose-Dateien für die Erzeugung einzelner Container findet (vergleiche dazu beispielhaft [Docker Inc., 2021f]). Im Falle der Ausführung mehrerer Container lässt sich mit Compose deutlich komfortabler arbeiten. Kann jedoch auf den Komfort einer Compose-Datei verzichtet werden, sollte überlegt werden, die Erzeugung weniger Container händisch über den run-Befehl durchzuführen (zum Beispiel durch ein Bash-Skript), da die Ersparnis immer noch im zweistelligen Bereich liegt.

7.2.5 Auswirkungen der Containergröße auf die Effizienz während der Laufzeit

Wie bereits in Abschnitt 7.2.3 gezeigt, hat die Größe eines des Basis-Images keinen Einfluss auf die Laufzeit und den Energieverbrauch von Build-Prozessen. Da mit großen Images auch häufig eine große Masse an Layern einhergeht, könnten diese jedoch Einfluss auf den Energieverbrauch eines laufenden Containers haben, da Layer beim Zugriff auf Dateien durchlaufen werden müssen. Im folgenden Abschnitt werden daher zwei Container mit unterschiedlich großem Basis-Image aber gleicher Funktion untersucht.

7.2.5.1 Messszenario und Messmethodik

Zur Messung wird die dritte Iteration des Messaufbaus aus Abschnitt 6.2.1 verwendet. Dadurch besteht die Möglichkeit, jede Messung genau dann auszuführen, wenn Arbeitslast erzeugt wird. Um Arbeitslast zu erzeugen, wird im Container ein selbstgeschriebener Server, der eine einfach „Hello World“-Webseite zur Verfügung stellt, in Go ausgeführt. Der Server läuft dabei zum einen in einem Container mit Basis-Image golang, welches insgesamt 788 MB groß ist, und zum anderen in einem Container mit Basis-Image golang:1.11-alpine3.10, welches 325 MB groß und damit um 60% kleiner ist. Abgesehen davon sind alle unnötigen Features von Docker, wie beispielsweise das Logging, ausgeschaltet.
Der Go-Server wird mit der Software Siege [Joe Dog Software, 2021] eine Minute unter Last gesetzt. Danach wird eine Minute pausiert und die Messung startet erneut. Insgesamt wird dieser Ablauf zehn Mal wiederholt. Um das Messszenario vorher ausreichend definieren zu können, ist es notwendig, die Belastungsgrenzen des Servers zu definieren, um so eine geeignete Arbeitslast für eine Untersuchung feststellen zu können. Hierfür werden Nutzer simuliert, die im Abstand von 0,1 Sekunden einen Zugriff auf den Server durchführen. Die Anzahl an Nutzern wird alle dreißig Sekunden um zwei weitere Nutzer erhöht. Stagniert dabei die Zugriffszahl im Vergleich zum vorherigen Messdurchlauf, wird dieser insgesamt abgebrochen.
In den folgenden Experimenten werden diese Messungen zur Platzersparnis nicht in Gänze dargestellt. Wie bereits zu Beginn des Kapitels erwähnt, sind die vollständigen Messreihen jedoch im Anhang der

User	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
2	1153	2,38	0,11	5,01
4	2314	4,96	0,19	5,08
6	3408	6,57	0,28	5,17
8	4575	7,71	0,34	5,25
10	5746	9,17	0,41	5,29
12	6890	10,3	0,47	5,35
14	8044	11,31	0,53	5,42
16	9283	12,35	0,58	5,48
18	10348	13,13	0,64	5,5
20	11440	13,84	0,69	5,59
22	12778	14,63	0,7	5,65
24	13889	15,48	0,75	5,69
26	15080	15,65	0,84	5,73
28	16214	17,12	0,89	5,77
30	17429	17,03	0,9	5,82

Tabelle 22: Messreihe zur Bestimmung der Belastungsobergrenze des Go-Servers (Basis-Image golang:1.11-alpine3.10 (325 MB))

User	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
2	1128	2,84	0,11	4,96
4	2298	5,03	0,19	5,1
6	3422	6,67	0,28	5,18
8	4528	7,99	0,33	5,22
10	5700	9,22	0,39	5,32
12	6826	9,87	0,46	5,37
14	8051	11,38	0,51	5,44
16	9238	12,29	0,58	5,48
18	10395	13,14	0,63	5,52
20	11532	13,95	0,69	5,59
22	12739	14,96	0,73	5,66
24	13913	15,56	0,78	5,71
26	15017	16,41	0,83	5,73
28	16341	17,18	0,85	5,78
30	17497	17,8	0,91	5,84

Tabelle 23: Messreihe zur Bestimmung der Belastungsobergrenze des Go-Servers (Basis-Image golang (788 MB))

Arbeit zu finden. Hier wird nur das Ergebnis aufgeführt. Für weitere Informationen zu den Messreihen kann das im Anhang angegebene Repository eingesehen werden. Wie beim Vergleich von Tabelle 22 und Tabelle 23 zu erkennen, liegt die maximale Anzahl an Zugriffen auf den Server bei circa 17500. Dies liegt zum einen daran, dass der Server aufgrund seiner Einstellung nur einen Zugriff parallel verarbeiten kann, zum anderen wird nur eine kleine, statische Webseite dargestellt. Vergrößert man die Seite, sinkt die Anzahl an Zugriffen deutlich ab. Dies wird später in Abschnitt 7.2.7 zu sehen sein. Zusätzlich ist zu erkennen, dass alle Werte gleichmäßig mit der Erhöhung der Nutzeranzahl und damit der Zugriffe ansteigen. Allerdings wird ebenfalls deutlich, dass im Falle von golang:1.11-alpine3.10 die CPU-Auslastung zwischen 28 und 30 Usern stagniert. Aus diesem Grund werden für das Experiment 28 Nutzer gewählt. Demnach ergibt sich folgendes Vergleichsszenario:

i) Go-Server mit statischer Webseite wird unter Last gesetzt

ii) Go-Server läuft im Container (einmal golang und einmal golang:1.11-alpine3.10)

iii) 28 Nutzer greifen alle 0,1 Sekunden auf den Server zu

iv) 20 Messrunden laufen je eine Minute mit je einminütiger Pause zwischen den Runden

v) Last wird durch Siege erzeugt

vi) Überprüft werden Zugriffsanzahl, CPU-Auslastung, durchschnittliche Leistungsaufnahme in Watt

7.2.5.2 Darstellung der Messergebnisse

Wie in Tabelle 24 und Tabelle 25 sichtbar, existiert nur ein Unterschied von 0,02 W bezüglich der durchschnittlichen Leitungsaufnahme. Dies entspricht einem Unterschied von weniger als 1%. Demnach sind die Unterschiede hier zu vernachlässigen. Auch bei CPU-Auslastung und Anzahl simultaner Zugriffe existieren keinerlei Unterschiede zwischen unterschiedlich großen Containern. Einzig bei der Anzahl an Zugriffen existiert noch ein Unterschied von 0,5%.

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	16200	16,6	0,86	5,79

Tabelle 24: Messergebnisse zum Go-Server; Belastung mit 28 Usern (Basis-Image golang:1.11-alpine3.10 (325 MB))

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	16284	16,6	0,86	5,81

Tabelle 25: Messergebnisse zum Go-Server; Belastung mit 28 Usern (Basis-Image golang (788 MB))

7.2.5.3 Interpretation der Ergebnisse

Wie bereits während des Build-Prozesses existieren keine erkennbaren Effizienzunterschiede zwischen kleineren und größeren Containern über die gesamte Ausführungszeit. Daher kann festgehalten werden, dass die Größe des Basis-Images keinen Einfluss auf die Energieeffizienz eines Containers hat, sofern beide Container denselben Funktionsumfang besitzen.

7.2.6 Auswirkungen der Imagegröße auf die Effizienz der Erzeugung eines Containers

Obwohl zu sehen war, dass die Größe eines Images keinen Einfluss auf den Build-Prozess eines Images hat (vergleiche Abschnitt 7.2.3) und ebenfalls nicht die Effizienz während der Laufzeit eines Containers beeinflusst (vergleiche Abschnitt 7.2.5), wird der Vollständigkeit halber im nachfolgenden Kapitel getestet, ob der Start unterschiedlich großer Container einen Einfluss auf die Effizienz hat.

7.2.6.1 Messszenario und Messmethodik

Zum Test werden die Starts der Container aus Abschnitt 7.2.5 mit Basis-Images golang (788 MB) und golang:1.11-alpine3.10 (325 MB) verwendet. Das Vorgehensschema der Messungen entspricht dem aus Abschnitt 7.2.4. Gestoppt wird demanch die Dauer des Startvorgangs vom Moment der Absetzung des Befehls bis zur Rückkehr auf die Kommandozeile sowie die durchschnittliche Leistungsaufnahme in diesem Zeitintervall. Hiermit wird, wie bereits in Abschnitt 7.2.4 erwähnt, nicht getestet, ob die Software innerhalb des Containers nach dessen Start bereits zur Verfügung steht, denn es geht nur um die Bereitstellung des Containers.

7.2.6.2 Darstellung der Messergebnisse

Vergleicht man die Messergebnisse aus Tabelle 26 und Tabelle 27 besteht, wie schon bezüglich des Build-Prozesses der Images (vergleiche Abschnitt 7.2.3) als auch bezüglich Effizienz während der Laufzeit eines Container (vergleiche Abschnitt 7.2.5) gezeigt, kein Unterschied zwischen unterschiedlich großen Containern mit gleicher Funktionalität. In beiden Fällen stimmen in diesem Experiment die durchschnittliche Dauer der Erzeugung des Containers als auch die mittlere Leistungsaufnahme überein.

7.2.6.3 Interpretation der Ergebnisse

Es kann festgehalten werden, dass die Größe eines Containers keinen Einfluss auf die Effizienz der

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	1,52	5,4
Standardabweichung	0,02	0,08

Tabelle 26: Erzeugung und Start eines Containers mit *docker run* und Basis golang

	Dauer in Sekunden	durchschnittliche Leistungsaufnahme in Watt
Mittelwert	1,52	5,4
Standardabweichung	0,02	0,08

Tabelle 27: Erzeugung und Start eines Containers mit *docker run* und Basis golang:1.11-alpine3.10

Container-Erzeugung mit *docker run* hat. Nach den Ergebnissen aus Abschnitt 7.2.3, Abschnitt 7.2.5 und den im vorherigen Abschnitt gezeigten Resultaten wird final festgehalten, dass weder die Image- noch die Container-Größe einen geeigneten Angriffspunkt für Verbesserungen der Energieeffizienz von Docker bieten.

7.2.7 Auswirkungen des Layersystems auf die Energieeffizienz während der Laufzeit von Containern

Wie in Absatz 3.3.1.3 beschrieben, sind alle Daten eines Containers in Read-Only-Layern gespeichert. Möchte man eine Datei verändern, so wird ein neuer Layer angelegt und alle Änderungen werden hier vermerkt. Somit ist klar, dass alle Container, die auf dem gleichen Image basieren, eine identische Layer-Basis besitzen. Tatsächlich teilen sich Container diese Layer-Basis, was vermuten lässt, dass es effektiver ist, stets die gleichen Basis-Images zu verwenden. Im folgenden Abschnitt soll daher zum einen untersucht werden, wie sich der Stromverbrauch ändert, wenn mehrere Images auf dem gleichen Image basieren und damit dieselben Layer nutzen. Es soll geklärt werden, ob sich der Stromverbrauch während der Skalierung, zum Beispiel von einem auf zwei Container, ändert. Zum anderen soll erkennbar werden, ob es sinnvoll ist, alle Container eines Clusters auf einem Image aufzubauen.

7.2.7.1 Messszenario und Messmethodik

Um zu überprüfen, welchen Einfluss das Layersystem auf die Energieeffizienz hat, muss zuerst untersucht werden, wie sich Container bei der Skalierung bezüglich ihres Energieverbrauchs verhalten. Dafür wird in einem ersten Schritt ein Container mit Nginx-Server unter Last gesetzt (vergleiche hierzu [Kreten et al., 2018]). Der Nginx-Server liefert dabei nur die statische Default-Seite des Webservers aus. Der Container wird mit Hilfe von Siege in zehn Messrunden für eine Minute unter Last gesetzt. Danach findet eine Pause von einer Minute statt. Als generellen Messaufbau wird Iteration 3 aus Ab-

schnitt 6.2.1 verwendet.
Nach Beendigung der Messung wird ein weiterer Container des gleichen Images erzeugt. Beide Container werden nun unter Last gesetzt. Die Gesamtlast ist dabei aber verdoppelt. Dieser Test wird noch einmal mit drei und vier Containern wiederholt.
Danach wird ein erneuter Test mit Containern gleicher Image-Basis durchgeführt. Dieses Mal wird die Last von Siege unter beiden Containern aufgeteilt. Dieser Vorgang wird nachfolgend noch mit 4, 8 und 16 Containern der gleichen Bauart wiederholt. Dabei wird die Last von 2 - 32 Nutzern erzeugt, die parallel auf den Server zugreifen.
In einem zweiten Experiment wird ein ähnlicher Test mit einem Go-Server getestet. Dabei wird dieser Server zum einen in einem Container mit Basis-Image golang und zum anderen in einem Container mit Basis-Image golang:1.11-alpine3.10 ausgeführt und liefern die gleiche statische Webseite aus. Beide Images haben, wie in Abschnitt 7.2.5 beschrieben, den gleichen Funktionsumfang, verwenden jedoch eine unterschiedliche Layer-Basis. Beide Container werden, wie im bereits zuvor erläuterten Messaufbau, unabhängig voneinander unter Last gesetzt. Danach werden beide Container in zweiter Ausführung erzeugt. Nach dem obigen Prinzip wird nun die Last unter beiden Containern gleicher Bauart aufgeteilt und der Stromverbrauch wird untersucht. Abschließend wird diese Messung mit einem Container mit Basis-Image golang und einem Container mit Basis-Image golang:1.11-alpine3.10 durchgeführt. Erneut soll hierzu der Stromverbrauch überprüft werden. Die hier verwendete Last entspricht circa 50% der maximal verarbeitbaren Zugriffe des Go-Servers. Dieser Wert wurde im Vorhinein durch einen Test mit linear ansteigender Last ermittelt, wie beispielhaft in Abschnitt 7.2.5 gezeigt.

7.2.7.2 Darstellung der Messergebnisse

Es ist offensichtlich anzunehmen, dass zwei Container mit gleicher Last die gleiche Leistung benötigen. Wie in Tabelle 28 zu erkennen, ist die Leistungsaufnahme von zwei Containern nicht doppelt so groß wie die Leistungsaufnahme eines Containers, sondern nur 60% größer. Ähnliches gilt für drei und vier Container, bei denen man eine Verdreifachung beziehungsweise Vervierfachung erwarten würde, jedoch nur 100% beziehungsweise 120% höhere Leistungsaufnahme erkennen kann.

	durchschnittliche Leistungsaufnahme in W
1 Container, Belastung durch 16 Nutzer	0,53
2 Container, Belastung durch je 16 Nutzer	0,84
3 Container, Belastung durch je 16 Nutzer	1,1
4 Container, Belastung durch je 16 Nutzer	1,3

Tabelle 28: Vergleich der Leistungsaufnahme (ohne Baseline) bei linear ansteigender Last und Containeranzahl

Dieser Vorteil zeigt sich auch während der Skalierung mit gleichzeitiger Lastverteilung, sofern geringe Lasten wirken. Tabelle 29 zeigt, dass die zusätzliche Verwendung eines Containers im Sinne einer

Skalierung und in den getesteten Szenarien (siehe zum Vergleich von Tabelle 30, Tabelle 31 und Tabelle 38) bei geringer Last zu einer identischen Leistungsaufnahme führt. Bei hoher Last kann die Skalierung zu einer 20% höheren Leistungsaufnahme im getesteten Szenario führen.

	2 Nutzer	4 Nutzer	8 Nutzer	16 Nutzer	32 Nutzer	64 Nutzer
1 Container	0,11 W	0,22 W	0,36 W	0,52 W	0,77 W	1,09 W
2 Container	0,11 W	0,23 W	0,38 W	0,57 W	0,83 W	1,16 W
4 Container	0,15 W	0,23 W	0,4 W	0,6 W	0,86 W	1,25 W
8 Container	0,16 W	0,25 W	0,38 W	0,6 W	0,9 W	1,29 W
16 Container	0,17 W	0,26 W	0,41 W	0,61 W	0,92 W	1,32 W

Tabelle 29: Vergleich der Leistungsaufnahme (ohne Baseline) eines skalierten Webservers mit gleichmäßiger Lastverteilung

Zur Überprüfung dieses Verhaltens mit Containern ungleicher Container-Basis sind Tabelle 30 und Tabelle 31 sowie Tabelle 32 zu vergleichen. Wie Abschnitt 7.2.5 darstellt, besteht zwischen dem Container basierend auf golang und dem deutlichen kleineren Container basierend auf golang:1.11-alpine3.10 kein Perfomanceunterschied. Dies lässt sich wie in Tabelle 30 und Tabelle 31 dargestellt, auch nach der Skalierung beider Container erkennen. In Tabelle 32 ist jedoch abzulesen, dass sich bei der Lastverteilung auf zwei Container unterschiedlicher Image-Basis eine Veränderung der Lastaufnahme einstellt. Während zuvor eine durchschnittliche Lastverteilung von 5,97 W verzeichnet wurde, hat sich diese inTabelle 32 um 0,8% erhöht.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	8005	18,2	0,5	5,97

Tabelle 30: Messergebnisse zum skalierten Go-Server; Belastung mit 28 Usern für 15 Sekunden (Basis-Image golang (788 MB))

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	8022	18,32	0,48	5,97

Tabelle 31: Messergebnisse zum skalierten Go-Server; Belastung mit 28 Usern für 15 Sekunden (Basis-Image golang:1.11-alpine3.10 (325 MB))

Erhöht man im gezeigten Szenario die Last weiter, so ergeben sich größere Veränderungen von bis zu 1,1%. Weiter ist in Tabelle 32 zu erkennen, dass der Start eines zweiten Containers, mit gleicher Last und unterschiedlicher Image- und damit Layer-Basis, nicht zur doppelten Leistungsaufnahme führt.

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	8041	18,74	0,49	6,02

Tabelle 32: Messergebnisse von zwei Go-Servern; Belastung mit 28 Usern für 15 Sekunden (Basis-Image Container 1: golang:1.11-alpine3.10 (325 MB), Basis-Image Container 2: golang (788 MB))

7.2.7.3 Interpretation der Ergebnisse

Wie zu sehen ist, führt der Start eines weiteren Containers mit gleicher Last nicht zu einer doppelt so hohen Leistungsaufnahme. Dies gilt sowohl für Container gleicher Layer-Basis als auch für die Container mit unterschiedlichen Layern. Man kann diesen Effekt auch bei der Skalierung von Containern mit geringen Lasten beobachten, bei denen eine Skalierung nur zu einer kaum veränderten Leistungsaufnahme führt. Es ist jedoch auch zu erkennbar, dass bei hoher Last und entsprechender Skalierung, der Energieverbrauch deutlich ansteigt.
Gleichermaßen lässt sich erkennen, dass die Verwendung gleicher Images, beziehungsweise einer gleichen Layer-Basis, einen geringen, aber erkennbaren Vorteil bei der Leistungsaufnahme und somit beim Energieverbrauch mit sich bringt.

7.2.8 Storage-Treiber

Das Layersystem wird je nach gewähltem Storage-Treiber unterschiedlich aufgebaut und sorgt während der Laufzeit dafür, dass Änderungen am R/W-Layer unterschiedlich abgehandelt werden. Entsprechend dessen, kann dies während der Laufzeit zu Performance-Schwankungen führen. Docker empfiehlt daher ausdrücklich den standardmäßigen Storage-Treiber *overlay2* zu verwenden. Trotzdem ist je nach Betriebssystem die Möglichkeit gegeben, den Storage-Treiber auszutauschen. Im folgenden Experiment werden die zwei Storage-Treiber *overlay2* und *ZFS* bezüglich ihrer Performance verglichen. Dabei wird eine Unterscheidung zwischen rein lesenden und den Layer verändernden Zugriffen gemacht.

7.2.8.1 Messszenario und Messmethodik

Als Grundlage für die Messung wird der Messaufbau aus Abschnitt 7.2.6 verwendet. Dabei werden die Container pro Messrunde dreißig Sekunden mit einer zuvor ermittelten Arbeitslast bespielt. In diesem Fall wird alle 0.1 Sekunden durch zwanzig simultane Nutzer auf die in den Containern laufenden Server zugegriffen. Eine Messung besteht dabei aus 20 Messrunden.
Die Messung wird in zwei Szenarien ausgeführt. Zum einen wird auf einen Nginx-Server mit statischer Webseite zugegriffen. Hierbei findet keine Veränderung der Layer statt, da die Webseite nur gelesen

wird. Zum anderen wird in einem zweiten Szenario auf einen Go-Server zugegriffen, der bei jedem Zugriff eine Datei erzeugt, wodurch getestet werden soll, wie effizient das Layersystem verändert wird. Es ist zu erwähnen, dass beide Storage-Treiber nicht parallel verwendet werden können. Nach der Messung mit *overlay2* muss daher der Docker Daemon neu gestartet werden, wodurch es zu einer geringen Verzögerung zwischen beiden Messungen kommt.

7.2.8.2 Darstellung der Messergebnisse

In Tabelle 33 und Tabelle 34 werden die Ergebnisse bezüglich der Messungen mit einem Nginx-Container und einfachen lesenden Zugriffen dargestellt. Es wird deutlich, dass es bezüglich der Zugriffsanzahl im Testintervall nur leichte Schwankungen gibt. Auch bei den simultan zu verarbeitenden Zugriffen sind kaum Unterschiede zwischen *overlay2* und *ZFS* erkennbar.

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	11545	7,24	0,64	5,68
Standardabweichungen	94	0,12	0,005	0,023

Tabelle 33: Zugriffe auf einen Nginx-Container mit overlay2 Storage-Treiber

Eine leichte Schwankung von circa 0,3% zeigt sich bei der CPU-Auslastung und auch bei der durchschnittlichen Leistungsaufnahme ist eine Verschlechterung von circa 0,9% beim Wechsel von *overlay2* auf *ZFS* erkennbar.

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	11574	7,52	0,63	5,73
Standardabweichungen	56	0,11	0,014	0,03

Tabelle 34: Zugriffe auf einen Nginx-Container mit ZFS Storage-Treiber

Betrachtet man jedoch die Ergebnisse bezüglich denjenigen Zugriffen, die das Layersystem verändern, zeigt sich ein anderes Bild. Vergleicht man Tabelle 35 und Tabelle 36, erkennt man durch die Nutzung von ZFS eine Verbesserung bezüglich der durchschnittlichen Leistungsaufnahme von circa 2,7%, obwohl die CPU-Auslastung im Falle von *ZFS* circa 1% höher ist. An der Performance des Go-Servers ändert sich bei Veränderung des Storage-Treibers nichts. Die Anzahl an verarbeiteten Zugriffen im Testintervall bleibt nahezu gleich.

7.2.8.3 Interpretation der Ergebnisse

Interpretiert man die Ergebnisse aus dem vorherigen Abschnitt, so wird ersichtlich, dass die Nutzung von ZFS bei rein lesenden Zugriffen keinen Vorteil, sondern eher einen Nachteil mit sich bringt.

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	11363	17,73	0,92	6,2
Standardabweichungen	101	0,1	0,06	0,03

Tabelle 35: Zugriffe auf einen Go-Server-Container mit overlay2 Storage-Treiber und dabei erzeugten Layern

	Zugriffe	CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	11321	18,62	1,0	6,03
Standardabweichungen	100	0,14	0,15	0,03

Tabelle 36: Zugriffe auf einen Go-Server-Container mit ZFS Storage-Treiber und dabei erzeugten Layern

Verändert man jedoch das Layersystem, so ergibt sich ein erkennbarer Vorteil bezüglich der durchschnittlichen Leistungsaufnahme. Dies kann man mit der effizienten Ausnutzung zweier Fähigkeiten von ZFS in Verbindung bringen. Wird ein neuer Layer erzeugt, so wird zuerst ein Snapshot des alten Layers erstellt, der wiederum in den neuen Layer geklont wird. Dies führt dazu, dass Lesezugriffe im Allgemeinen ähnlich performant wie bei *overlay2* sein sollten, da der Layer alle Daten enthält. Im Falle von schreibenden Zugriffen hat dies zum Vorteil, dass der gesamte Layer veränderbar ist. Es müssen keine neuen Layer hinzugefügt werden, da der Layer selber eine Kopie ist. Nachteil ist hierbei der große Speicheraufwand. Um ZFS benutzen zu können, werden zum einen SSDs vorausgesetzt und zum anderen wird jeder ZFS-Pool gespiegelt. Ergänzend dazu muss der zusätzliche administrative Aufwand beachtet werden, den die Verwendung von ZFS mit sich bringt. Daher sollte vor der Nutzung von ZFS unbedingt eine ausführliche Analyse des Anwendungsfalls durchgeführt werden.

7.2.9 Effizienz von Containern während der Laufzeit mit und ohne Volumes

In [Felter et al., 2015] wird gezeigt, dass Volumes insgesamt energieeffizienter als das AUFS-Dateisystem sind. Da AUFS nicht mehr der voreingestellte Storage Driver in Docker ist (seit Version 18.07), kann nicht mehr allgemein davon ausgegangen werden, dass Volumes per Default für einen geringeren Energieverbrauch stehen. Aus diesem Grund werden im Folgenden sowohl schreibende als auch lesende Zugriffe auf im Container persistent gespeicherte Daten beziehungsweise im Volume abgelegte Daten hinsichtlich ihrer Leistungsaufnahme überprüft.

7.2.9.1 Messszenario und Messmethodik

Der Messaufbau des Versuchs wird bezüglich der Messmethodik aus Abschnitt 7.2.5 übernommen. Ein Messdurchgang umfasst dabei 20 fünfzehnsekündige Messrunden mit einer Pause von 30 Sekunden zwischen den einzelnen Runden. Im Vorhinein wird für jedes Messszenario eine geeignete

Belastung ermittelt, indem der Service mit einer linear ansteigenden Arbeitslast bespielt wird, bis der Service zusammenbricht. Nachfolgend werden dabei folgende Szenarien auf die Unterschiede zwischen Volumes und persistenter Speicherung überprüft:

i) Nginx-Server liefert dynamische Webseite aus (Zugriffe nur lesend)

ii) Skalierter Nginx-Server liefert dynamische Website aus (Zugriffe nur lesend)

iii) Go-API mit Datenbank (nur lesende Zugriffe)

iv) Go-API mit Datenbank (nur schreibende Zugriffe)

In allen vier Fällen werden die Daten sowohl einmal im Container als auch im Volume bereitgestellt. Im Falle der Go-API werden die Daten der MySQL Datenbank im Volume gelagert.

7.2.9.2 Darstellung der Messergebnisse

Vergleich man Tabelle 37, in der die gemittelten Messergebnisse eines Nginx-Servers mit persistent abgelegten Daten präsentiert werden, und Tabelle 39, in welcher die Messergebnisse bezüglich des Servers mit im Volume abgelegten Daten dargestellt werden, so kann man keinen Unterschied in der Leistungsaufnahme feststellen. Erst bei Skalierung beider Services, lassen sich Unterschiede von circa 0,6% erkennen.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	9637	11,98	3,12	6,2

Tabelle 37: Messung eines Nginx-Servers mit persistent im Container abgelegter dynamischer Webseite

Steigert man die Belastung der Services weiter, steigt dieser Unterschied in den getesteten Szenarien auf maximal 0,8% an.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	10003	13,06	1,42	6,3

Tabelle 38: Messung eines skalierten Nginx-Servers mit persistent im Container abgelegter dynamischer Webseite

Da die Unterschiede an dieser Stelle sehr gering sind, wurde die Messung weitere vier Male wiederholt, wobei das gleiche Ergebnis erzielt wurde.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	9872	11,95	3,06	6,2

Tabelle 39: Messung eines Nginx-Servers mit im Volume abgelegter dynamischer Webseite

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	10172	12,9	1,45	6,26

Tabelle 40: Messung eines skalierten Nginx-Servers mit im Volume abgelegter dynamischer Webseite

Auch beim Szenario der API zeigt sich ein ähnliches Bild. Im Falle der lesenden Zugriffe, dargestellt bezüglich der persistenten Speicherung in Tabelle 41, und bezüglich der Datenspeicherung dargestellt in Tabelle 42, ergeben sich erneut Unterschiede in der Leistungsaufnahme von circa 0,8%.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	2622	33,12	1,2	6,41

Tabelle 41: Messung eines Go-API-Servers mit im Container abgelegter Datenbank und ausschließlich lesenden Zugriffen

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	2623	30,9	1,1	6,36

Tabelle 42: Messung eines Go-API-Servers mit im Volume abgelegter Datenbank und ausschließlich lesenden Zugriffen

Zusätzlich hierzu erkennt man auch bei schreibenden Zugriffen, dargestellt in Tabelle 43 und Tabelle 44, einen minimalen Unterschied von circa 0,2%. Alle Messungen zeigen zudem, dass es ebenfalls nur leichte Unterschiede bezüglich der CPU-Auslastung und der im Messintervall verarbeiteten Zugriffe gibt.

7.2.9.3 Interpretation der Ergebnisse

Da sich die Unterschiede bezüglich der Leistungsaufnahme in einem Bereich von unter einem Prozent bewegen, kann keine allgemeine Aussage darüber getroffen werden, ob Volumes energieeffizienter als die persistente Speicherung der Daten im Container sind. Eine plausible Erklärung für die geringe Verbesserung mit Volumes wären jedoch die Layer des Containers, die bei jedem lesenden Zugriff auf

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	3533	24,6	4,0	6,749

Tabelle 43: Messung eines Go-API-Servers mit im Container abgelegter Datenbank und ausschließlich schreibenden Zugriffen

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
durchschnittliche Werte	3536	24,6	4,0	6,738

Tabelle 44: Messung eines Go-API-Servers mit im Volume abgelegter Datenbank und ausschließlich schreibenden Zugriffen

persistente Daten durchlaufen werden müssen, um die auszuliefernden Daten zu finden, was zu einem zusätzlichen, wenn auch geringen, Overhead führt. Bei schreibenden Zugriffen wäre es die Erstellung eines zusätzlichen Layers, der aus jeder Dateneintragung in der Datenbank resultiert. Festzuhalten ist an dieser Stelle jedoch die wichtige Erkenntnis, dass Volumes im Vergleich zur persistenten Speicherung nicht zu einer Verschlechterung der Leistungsaufnahme und damit der Energieeffizienz führen.

7.2.10 Netzwerk-Treiber

Im nachfolgenden Kapitel wird das standardmäßig verwendete Bridge Network von Docker mit dem Bridge Network von Open vSwitch verglichen. Wie in den Ergebnissen der Umfrage zu sehen, wird nur selten eine Veränderung der Standardtreiber vorgenommen, da diese mit einem nicht zu unterschätzenden Mehraufwand verbunden sind. Die folgenden Ergebnisse sollen zeigen, ob sich dieser Mehraufwand Performance-technisch lohnen kann.

7.2.10.1 Messszenario und Messmethodik

Um zu überprüfen, ob ein Netzwerk-Treiber effizient arbeitet, wird in diesem Test kein in der Praxis verwendbarer Container genutzt. Während die bisherigen Testcontainer Bestandteil einer tatsächlichen Containerumgebung sein könnten, um beispielsweise eine Webseite oder eine API auszuliefern, wird in diesem Messaufbau eine Containerumgebung aus drei Worker- und einem Master-Container verwendet. Diese Container bilden im Verbund einen TCP-Kreis, wobei der Master-Container ein TCP-Paket durch diesen Kreis sendet. Dieser Vorgang wird für eine Minute wiederholt, wobei der Master dabei die Anzahl an vollständigen Runden zählt. Diese Werte werden im Anschluss an den DAE weitergegeben, der diese dann mit Verbrauchswerten bezüglich Leistung und letztlich Energieverbrauch verbindet und speichert. Die Messung wird insgesamt zehn Mal wiederholt.
Der DAE ist während der Messung über den Port 9000 mit dem Master-Container verbunden, während

alle anderen Container einen Server auf Port 8080 bereitstellen. Software-seitig wird hier innerhalb der Container erneut mit Go gearbeitet, da die Channels die Kommunikation zwischen Server-Thread und Client-Thread eines jeden Containers vereinfachen. Auf Seiten des DAE wird mit einer Anpassung des aus Abschnitt 6.2.3 bekannten Python-Scripts gearbeitet, welches hier jedoch keine Last erzeugt, sondern das Experiment via Netcat anstößt und im Testintervall, zwischen Request an den Master-Container und dessen Response, die Daten des PM auswertet. Um die in diesem Fall komplexe Messumgebung zu starten, welche in Abbildung 43 dargestellt ist, wird Docker-Compose verwendet. Die dazu notwendige Compose-Datei wird in Listing 21 präsentiert.

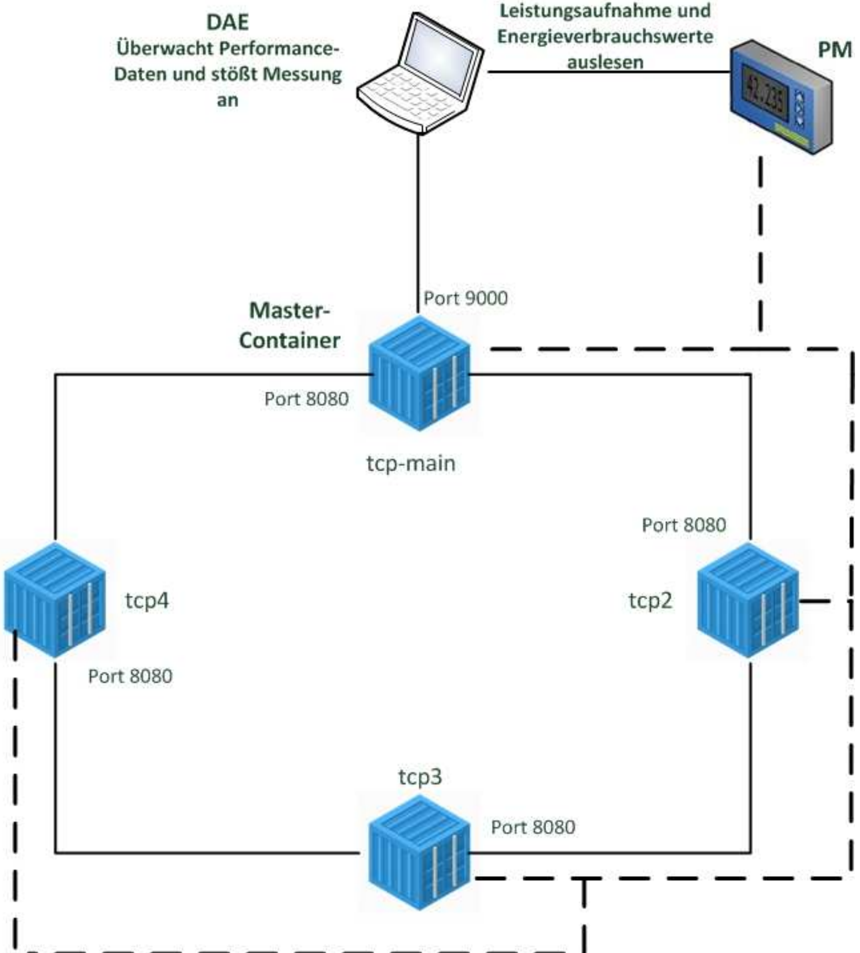

Abbildung 43: Messaufbau zur Überprüfung der Effizienz des Bridge-Treibers (eigene Abbildung)

In diesem Fall werden alle Container erzeugt und durch den angegeben Command wird das Go-Programm innerhalb des Containers gestartet. Das Go-Programm erhält dabei zum Start zwei Parameter. Bei ihnen handelt es ich zum einen um den Port des eigenen Servers und zum anderen um die Zieladresse für den TCP-Client. Ganz unten in der Compose-Datei ist die explizite Angabe eines Netzwerks mit Bridge-Treiber gegeben. Hierdurch wird beim Start ein neues Netz mit dem Namen *go_net* erzeugt, welches auf dem standardmäßigen Bridge-Treiber basiert.

```
version: '3.7'
services:
   tcp1:
      image: golang
      command: ./tcp-test :8080 tcp2:8080
      ports:
           - "9000:9000"
      volumes:
           - ./tcp-main:/tcp-main

      working_dir: /tcp-main
      container_name: tcp-main
      networks:
          - go_net
   tcp2:
      image: golang
      command: ./tcp-answer-test :8080 tcp3:8080
      volumes:
            - ./tcp-sec:/tcp-sec
      working_dir: /tcp-sec
      container_name: tcp2
      networks:
            - go_net
   tcp3:
      image: golang
      command: ./tcp-answer-test :8080 tcp4:8080
      volumes:
            - ./tcp-sec:/tcp-sec
      working_dir: /tcp-sec

      container_name: tcp3
      networks:
            - go_net
   tcp4:
      image: golang
      command: ./tcp-answer-test :8080 tcp1:8080
      volumes:
            - ./tcp-sec:/tcp-sec
      working_dir: /tcp-sec
      container_name: tcp4
      networks:
            - go_net

networks:
```

```
    go_net:
      driver: bridge
```

Listing 21: docker-compose.yaml zum Start der Netzwerk-Testumgebung

Zum Start der gleichen Umgebung auf Basis der Open vSwitch Bridge (OVS-Bridge) sind einige umfangreiche Anpassungen zu tätigen, die den Umgang mit den Linux Net-Tools, Iptables und Open vSwitch selbst erfordern. Gestartet werden die Server hier zu Beginn mit dem Treiber *none*. Danach wird eine neue OVS-Bridge erzeugt, deren Gateway über einen Port mit dem Netzwerk-Interface des Hosts verbunden werden muss, sodass der Docker Host dieses Gateway erreichen kann. Nachfolgend muss mittels Iptables eine Port-Weiterleitung erfolgen, da der über Docker Expose freigegebene Port 9000 bei OVS nicht automatisch einsatzbereit ist, wodurch der DAE nicht auf diesen zugreifen kann. Abschließend wird jeder einzelne Container dem neuen Gateway zugeordnet und erhält eine IP-Adresse. Auch hier ergeben sich Schwierigkeiten, da die von Docker bekannte automatische Hostname-Auflösung nicht funktioniert. Der Start des Go-Programms, wie in Listing 21 gezeigt, muss demnach angepasst werden, sodass die Hostnames durch IP-Adressen zu ersetzen sind.

7.2.10.2 Darstellung der Messergebnisse

Tabelle 45 präsentiert die Ergebnisse des Experiments. Klar wird, dass durch den standardmäßigen Docker Bridge Treiber während des Testintervalls eine um 0,43 Watt niedrigere Leistungsaufnahme erreicht wird, was einer Verringerung von fast 7% entspricht. Im Gegensatz dazu steht jedoch die Anzahl an Runden, die ein Paket während des Testintervalls im TCP-Kreis erreicht.

	Docker Bridge	OVS Bridge
durchschnittliche Leistungsaufnahme in Watt	6,0	6,43
durchschnittlich verbrauchte Energie in Ws	360	385,8
durchschnittliche Anzahl an Runden	63274	71540

Tabelle 45: Ergebnisübersicht der Messung bezüglich des Energieverbrauchs der Netzwerk-Treiber Bridge und OVS Bridge

Hierbei werden durch den OVS Bridge Treiber im Mittel 8000 Runden mehr als beim Docker Bridge Treiber erreicht. Setzt man nun die ebenfalls in Tabelle 45 dargestellte durchschnittlich verbrauchte Energie im Testintervall mit den zurückgelegten Runden in Verbindung, so erhält man eine neue funktionale Einheit „Runden pro Ws“.

	Docker Bridge	OVS Bridge
Runden pro Ws	175,5	185,5

Tabelle 46: Runden pro Ws der Netzwerk-Treiber Bridge und OVS Bridge

In Tabelle 46 wird deutlich, dass dementsprechend bei den Tests mit OVS mehr Arbeit im Testintervall erledigt wird. Um die gleiche Arbeit zu verrichten, müsste das Testintervall für den Docker Bridge Treiber fast acht Sekunden länger sein.

7.2.10.3 Interpretation der Ergebnisse

Auf den ersten Blick scheint der Docker Bridge Treiber einen energetischen Vorteil bezüglich des Energieverbrauchs zu haben. Anders als bei den restlichen Experimenten jedoch, existiert ein Unterschied bezüglich der verrichteten Arbeit im Testintervall. Verwendet man den OVS Bridge Treiber, führt dies zu einer schnelleren Kommunikation zwischen den Containern. Als Konsequenz wird hierdurch eine geringere Lastzeit und damit ein verringerter Energieverbrauch erreicht. Andere Treiber-Plugins wie Weave oder Contiv dienen der Kommunikation zwischen unterschiedlichen Hosts und sind damit mit dem Overlay Treiber von Docker zu vergleichen. Dies wird in dieser Arbeit aufgrund des Administrationsaufwands von virtuellen Netzen jedoch nicht umgesetzt.

7.2.11 Logging-Treiber

Der Logging-Mechanismus und die Logging-Treiber von Docker erlauben die Weiterleitung von im Container geschriebenen Logs auf den Docker Host sowie auf externe Logging-Plattformen. Dabei hat jeder Logging-Treiber andere Funktionalitäten. So puffern manche Logging-Treiber die Logs vor der Weiterleitung, während andere direkt an andere Dienste weiterleiten. Im Folgenden werden sieben Logging-Treiber in ihren Standardeinstellungen bezüglich ihres Energieverbrauchs überprüft. Dabei wurden die gängigsten Logging-Treiber ausgewählt, die ohne die Nutzung einer externen Plattform lauffähig sind. Hierzu zählen, bis auf *gelf*, alle durch die Teilnehmer der Umfrage angegebenen Treiber.

7.2.11.1 Messszenario und Messmethodik

Als grundsätzlicher Messaufbau wird erneut jener aus Abschnitt 7.2.5 verwendet. Dabei besteht ein Messdurchgang erneut aus zwanzig hintereinander ausgeführten, dreißigsekündigen Messrunden, welche mit einer halben Minute voneinander getrennt sind. Getestet wird ein Nginx-Server, der bei jedem Zugriff einen Log-Eintrag unter */var/log/nginx/access.log* verfasst. Die Datei access.log entspricht innerhalb des Containers jedoch nur einem Symoblic Link auf das standardmäßige Ausgabe-Interface stdout. Alle getesteten Treiber, mit Ausnahme des Treibers *none*, der das Logging ausschaltet, verarbeiten alle Ausgaben auf diesem Interface. Im Vorhinein wurde erneut eine mittlere Auslastung des Servers, in Form von parallel auf den Dienst zugreifenden Nutzern, ermittelt. Die Nutzer greifen dabei alle 0,01 Sekunden auf den Server zu. In Abbildung 44 ist zu erkennen, dass alle Treiber, bis auf *journald*, bei 16 beziehungsweise 18 Nutzern die maximale CPU-Auslastung des Dienstes, die bei

25%, als der genauen Auslastung eines CPU-Kerns liegt, erreichen. *journald* erreicht seine maximale Auslastung bereits bei 14 Nutzern.

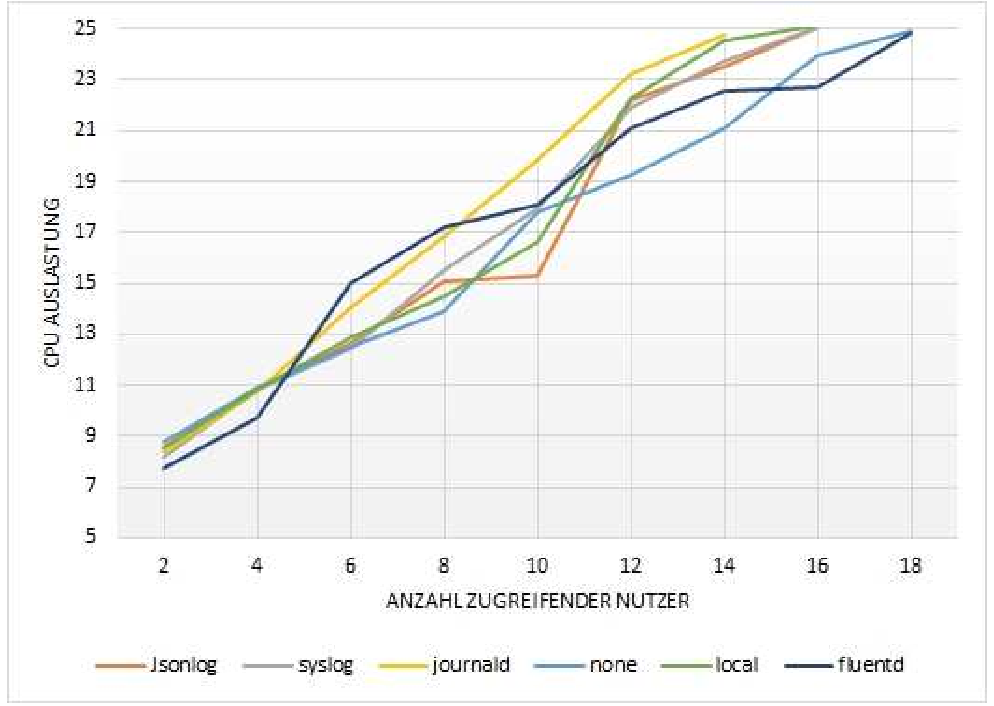

Abbildung 44: Lastmaxima des Nginx-Servers unter Benutzung der verschiedenen Treiber

Entsprechend diesen Ergebnissen ergibt sich als brauchbare Last für den Versuch eine Verwendung von acht beziehungsweise zehn Nutzern. Für die weiteren Tests werden zehn Nutzer verwendet. Aufgrund der geringen Datenmenge, die in jedem Zugriff ausgeliefert wird, ist eine hohe Anzahl an Zugriffen notwendig, um eine entsprechende Last auf dem Server zu erzeugen. Bei zehn Nutzern entspricht dies einer Zugriffszahl von circa 45.000 innerhalb des dreißig Sekunden Testintervalls. Erhöht man die Anzahl und die Größe der Daten, so würde sich diese Zahl deutlich verringern, so wie in Tabelle 39 erkennbar. Da es bei diesem Versuch jedoch um die Arbeit des Logging-Treibers und nicht um die Lese-/Schreibarbeit des Servers oder des Layersystems geht, wird mit einer geringen Datenmenge und dafür mit einer hohen Anzahl an Zugriffen gearbeitet, um Arbeitslast zu erzeugen. Zudem ist zu beachten, dass eine solch hohe Zugriffszahl besonders im Bereich von APIs und M2M Kommunikation erwartet werden kann.

7.2.11.2 Darstellung der Messergebnisse

Wie man in Tabelle 47 erkennen kann, weichen die Ergebnisse bei der Verwendung unterschiedlicher Loggintreiber deutlich voneinander ab. Während der standardmäßig eingestellte Logging-Treiber *json-file* im Versuch eine mittlere Leistungsaufnahme von 7,0 W hatte, weist der, wie auch aus der Umfrage erkennbar, ebenfalls häufig verwendete Treiber *fluentd* eine um circa 6% höhere Leistungsaufnahme auf. Hier ist ebenfalls der Anstieg der CPU-Auslastung um 1% sowie die Erhöhung der

simultan zu verarbeitenden Zugriffe zu beachten. *syslog* und *journald* werden zusätzlich zu den oben genannten Treibern des Öfteren verwendet und weisen im Falle von *journald* eine um 7% geringere Leistungsaufnahme als *json-file* und im Falle von *syslog* eine um circa 0,5% geringere Leistungsaufnahme auf.

Logging-Treiber	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
none	47460	14,05	1,9	6,3
local	46294	14,4	2,0	6,5
json-file	44478	16,4	2,4	7,0
journald	45821	14,4	2,1	6,52
syslog	44796	16,0	2,3	6,97
fluentd	43381	17,3	2,6	7,4

Tabelle 47: Übersicht über die Effizienz der Logging-Treiber *none*, *fluentd*, *local*, *json-file*, *journald* und *syslog*

Ebenfalls wird deutlich, dass der Treiber *local* eine um circa 7% geringere Leistungsaufnahme besitzt und darüber hinaus für eine um 2% geringere CPU-Auslastung sorgt. Zusätzlich dazu verringert sich hier die Anzahl an simultan zu verarbeitenden Zugriffen um 0,4, was somit die Erreichbarkeit des Dienstes verbessert.

Entscheidet man sich im getesteten Szenario gegen das Logging und damit für den Treiber *none*, so kann man sowohl die Leistungsaufnahme als auch die CPU-Auslastung und die Anzahl der simultanen Zugriffe verbessern. Dies äußert sich natürlich ebenfalls in einer höheren Anzahl verarbeiteter Zugriffe. Bezogen auf die Leistungsaufnahme kann im Testszenario von einer Reduktion um circa 10% ausgegangen werden.

Da sich in Abschnitt 7.2.9 zeigte, dass Volumes in jedem Fall nicht zu einer schlechteren Leistungsaufnahme führen, wurde in einem weiteren Versuch für alle Logdateien des Servers ein Volume zur Verfügung gestellt. Außerdem wurde der Logging-Treiber *none* verwendet. Um die Logdateien anschließend auslesen zu können, wurden diese mit Hilfe von *systemd-cat* an das System-Journal (journald) angehangen. Die Ergebnisse bezüglich dieses Szenarios werden in Tabelle 48 dargestellt.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
Mittelwert	48179	14,2	1,9	5,9
Standardabweichung	343,5	0,21	0,05	0,02

Tabelle 48: Übersicht über die Effizienz des Logging-Treiber *none* in Kombination mit Logging im Volume

Im Gegensatz zum Standardtreiber *json-file* steigert die neue Variante basierend auf Volumes die Anzahl der verarbeiteten Zugriffe im Testintervall um fast 4.000 und damit auch die Anzahl der simultan

zu verarbeitenden Zugriffe. Auch verringert sich die Leistungsaufnahme auf 5,9 W und ist damit um 15% geringer als beim Standard.

7.2.11.3 Interpretation der Ergebnisse

Insgesamt ist festzuhalten, dass Logging-Treiber eine erkennbare Möglichkeit zur Verringerung des Energieverbrauchs einer Containerlandschaft bieten. Mit einfachen Mitteln und kleinen Anpassungen, so wie exemplarisch mit der Nutzung eines Volumes gezeigt, lassen sich große Ersparnisse erreichen. Da das Logging in Docker im Allgemeinen als problematisch angesehen wird, würde an dieser Stelle eine weitere Fortführung der Forschung empfohlen werden.
Als Fazit ergibt sich, dass sich besonders *fluentd* als ineffizient bezüglich CPU-Auslastung und Energieverbrauch herausgestellt hat, obwohl dies die Grundlage der in Kubernetes zur Verfügung gestellten, externen Logging-Agents auf Cluster-Level darstellt und damit in einer Vielzahl umfangreicher Containerumgebungen verwendet wird. Kubernetes selber spezifiziert keinen nativen Logging-Agent für Cluster, sondern setzt nur auf jene externe Treiber:

> Kubernetes doesn't specify a logging agent, but two optional logging agents are packaged with the Kubernetes release: Stackdriver Logging for use with Google Cloud Platform, and Elasticsearch. You can find more information and instructions in the dedicated documents. Both use fluentd with custom configuration as an agent on the node. pK8Log

7.2.12 Container-Runtimes *runC*, *kata* und *runq*

In Kapitel 3 wurde die Verbindung zwischen Virtualisierung und Containern beschrieben. Zwar besteht hier eine Verbindung, von einer wirklichen Virtualisierung kann jedoch nicht gesprochen werden. Aus diesem Grund wird häufig darüber diskutiert, ob Container wirklich isoliert sind oder nur eine Virtualisierung dies gewährleisten kann. Aus diesem Grund wurden neben der Runtime *runC* auch andere in Docker verwendbare Container-Runtimes entwickelt, die Container und Virtualisierung verbinden. Die Runtimes *kata* und *runq*, die beispielsweise KVM nutzen, um innerhalb eines Containers eine leichtgewichtige virtuelle Maschine aufzusetzen, versuchen hier eine Verbindung beider Techniken herzustellen, um so die gewünschte Isolierung zu erzeugen. Dabei kommuniziert containerd nicht wie beschrieben mit *runC*, sondern mit einem Hypervisor, um alle Anfragen in die VM umzuleiten. Im Folgenden soll überprüft werden, ob diese alternativen Runtimes bezüglich ihrer Performance und im Speziellen bezüglich ihres Energieverbrauchs mit *runC* mithalten können.

7.2.12.1 Messszenario und Messmethodik

Als Messaufbau wird derjenige aus Abschnitt 7.2.8 verwendet. Hierbei greifen zwanzig simultane Nutzer alle 0,1 Sekunden auf einen Nginx-Server innerhalb eines Containers zu. Eine Messrunde

dauert dabei dreißig Sekunden, gefolgt von einer ebenfalls dreißig Sekunden langen Pause. Die Messrunde wird dabei zwanzig Mal wiederholt. Das Messszenario beschränkt sich auf die Auslieferung einer statischen Webseite.

7.2.12.2 Darstellung der Messergebnisse

Wie deutlich beim Vergleich von Tabelle 49, Tabelle 50 und Tabelle 51 zu erkennen ist, verringert sich die Anzahl an verarbeiteten Zugriffen beim Wechsel von *runC* zu *kata* um fast 8.000 und damit um fast 70%. Gleichzeitig sinkt jedoch die CPU-Auslastung des Containers um 4% ab. Die CPU-Auslastung steigt hierbei auch deutlich um fast 2 Watt und damit um 33% an.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
Mittelwert	11532	7,22	0,63	5,47
Standardabweichung	73	0,19	0,02	0,02

Tabelle 49: Zugriffe auf einen Nginx-Container mit Container-Runtime *runC*

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
Mittelwert	3628	3,1	13,93	7,3
Standardabweichung	75,65	0,15	0,155	0,06

Tabelle 50: Zugriffe auf einen Nginx-Container mit Container-Runtime *kata*

Vergleicht man *runq* und *kata* fällt hier eine erneute Verschlechterung der verarbeiteten Zugriffe um fast die Hälfte auf. Die CPU-Auslastung steigt auf fast 40% an, während die durchschnittliche Leistungsaufnahme erneut um 0,23 Watt steigt.

	Zugriffe	durchschnittliche CPU-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
Mittelwert	1930	39,6	16,69	7,53
Standardabweichung	49,37	0,6	0,1	0,07

Tabelle 51: Zugriffe auf einen Nginx-Container mit Container-Runtime *runq*

7.2.12.3 Interpretation der Ergebnisse

Die Messungen zeigen einen deutlichen Performance-Unterschied zwischen den getesteten Runtimes. Auffällig ist besonders bei *kata* die geringe CPU-Auslastung. Man sieht hierbei, dass die Verbindung zwischen Virtualisierung und Container nur bedingt besteht und die Hauptlast von der VM erzeugt

wird. Im Falle von *runq* wird diese Verbindung deutlicher. Die erhöhte Leistungsaufnahme zeigt sich klar in der stark gestiegenen CPU-Auslastung. Zum aktuellen Zeitpunkt gibt es neben den vorgestellten Runtimes noch weitere, die sich mit der Verbindung zwischen Virtualisierung und Container befassen. Ein Wechsel im Sinne der Isolierung klingt interessant, kann aber aufgrund der Einflüsse auf die Performance der Apps innerhalb des Containers nicht empfohlen werden.

7.3 Zusammenfassung und Fazit aller Messungen

Wie die Experimente gezeigt haben, bietet Docker in seiner Standardeinstellung einige Angriffspunkte bezüglich einer Verbesserung des Energieverbrauchs von Docker Containern. Dabei zeigen sich einige Verbesserungsmöglichkeiten jedoch deutlicher als andere. Die folgenden Potentiale konnten mit Hilfe der Messungen und der entsprechenden Szenarien ermittelt werden:

i) Build-Prozess

   (a) Verwendung des Caches
   Energieersparnis von circa 20%

   (b) Verwendung von Cachebusting
   Energieersparnis von circa 10%

   (c) Verwendung von Multistage-Builds
   keine Ersparnis ermittelbar

   (d) Verwendung von Buildkit
   Energieersparnis von circa 23%

ii) Erzeugen eines Containers
run 370% effizienter als Compose

iii) Container im IDLE
Stromverbrauch abhängig von der Software im Container

iv) Image und Containergröße
keine Unterschiede ermittelbar

v) Auswirkungen des Layersystems auf skalierte Umgebungen
Nutzen gleicher Images führt zu Energieersparnis von circa 1%

vi) Storage Driver
ZFS bei schreibenden Zugriffen um 2,7% bessere Leistungsaufnahme als overlay2
ZFS bei lesenden Zugriffen um 0,9% schlechtere Leistungsaufnahme als overlay2

vii) Volumes
Verwendung von Volumes führt zu 0,6-0,8% besserer Leistungsaufnahme

viii) Logging
Ersparnis abhängig vom verwendeten Treiber, zum Standard 15% geringere Leistungsaufnahme erreicht

ix) Networks
Verringerung der Leistungsaufnahme um 7% erreicht

x) Container Runtimes
keine Ersparnis durch Wechsel der Runtime ermittelbar

In sieben von zehn untersuchten Bestandteilen von Docker konnten Verbesserungen bezüglich der Energieeffizienz ermittelt werden. Dabei waren nur in den Fällen Networks und Storage Treiber erweiterte administrative Kenntnisse notwendig, um die Verbesserungen herbeizuführen. Daher kann festgehalten werden, dass Docker als System zwar im Allgemeinen zur verbesserten Ausschöpfung vorhandener Rechnerressourcen und zur Serverkonsolidierung beiträgt, in seinem Kern jedoch nicht bezüglich der Energieeffizienz optimiert ist. Trotzdem kann Docker ohne weitere Veränderung der Grundkonfiguration einen Großteil aller auf Docker Hub verfügbaren Anwendungen ausführen, weswegen ein genauer Blick auf die Grundkonfigurationen oftmals ausbleibt. Der Out-of-the-Box-Gedanke gleicht dabei die Nachteile in der Performance aus. Aufgrund der immensen Anzahl von Docker Containern besteht daher wie angenommen großes Potential zur Einsparung von Energie und aus wirtschaftlicher Sicht Potential zur Geldeinsparung.

8 Handlungsempfehlungen für die energieeffiziente Nutzung von Docker (Modell-Kernbestandteil II)

Durch die in Kapitel 7 dargestellten Messergebnisse lassen sich nun einige Handlungsempfehlungen für einen energieeffizienteren Umgang mit Docker aufstellen. Diese werden nachfolgend präsentiert und abschließend mit Simulationen empirisch überprüft, indem zwei geeignete, realitätsnahe Szenarien gewählt werden, um die Ergebnisse auf den täglichen Gebrauch von Docker übertragen zu können. Die Handlungsempfehlungen, welche Teil II des in Unterkapitel 6.1 dargestellten Modells repräsentieren, sehen dabei wie folgt aus:

- Optimiere den Build-Prozess von Images stets mit Buildkit und verwende, falls möglich, den Cache oder mindestens Cachebusting
- Verwende kein Docker-Compose, sofern die Komplexität deiner App dies zulässt
- Lösche Container und erzeuge sie neu, anstatt sie zu stoppen und wieder zu starten
- Nutze eine geringe Anzahl unterschiedlicher Basis-Images, um das Layer-System geschickt auszunutzen (erstelle eine eigene Container Registry mit eigenen *trusted Images*)
- Verwende stets Volumes
- Wähle den Logging-Treiber abhängig vom Anwendungsfall
  - Erzeuge, falls möglich, Logs in Volumes
  - Vorsicht bei Third-Party Logging-Treibern
- Besondere Handlungsempfehlungen für Nutzer mit entsprechenden administrativen Kenntnissen:
  - Nutze alternative Netzwerk-Treiber wie OVS, Weave oder Contiv
  - Nutze ZFS, falls Docker auf Ubuntu, CentOS (RHEL) oder Fedora läuft und hauptsächlich schreibende Zugriffe auf die Services vollzogen werden

Um zu zeigen, dass die Einhaltung der zuvor aufgestellten Handlungsempfehlungen im täglichen Gebrauch mit Docker große Vorteile mit sich bringen, werden im folgenden Kapitel, im Sinne einer empirischen Überprüfung, zwei realitätsbezogene Einsatzszenarien aufgestellt, umgesetzt und anschließend durchgemessen. Darüber hinaus finden mit Hilfe dieser Messungen Simulationen statt, welche die Einsatzszenarien auf die Nutzung innerhalb eines ganzen Jahres projiziert. Es werden hier zwei unterschiedliche Szenarien betrachtet, da sich zum einen nicht alle Handlungsempfehlungen für jedes

Szenario eignen und zum anderen, um die Ersparnisse in Abhängigkeit vom Szenario zu verdeutlichen. Gemäß der typischen Szenarien aus [Öggl and Kofler, 2018] und der bei Frage 6 der Umfrage aus Kapitel 5 („typischen Einsatzszenarien“) gegebenen Antworten ergeben sich hierbei REST-APIs und dynamische Webseiten als geeignete Fallbeispiele, die im Folgenden untersucht werden.

8.1 Empirische Überprüfung der Handlungsempfehlungen

Im Folgenden sollen die zuvor aufgestellten Handlungsempfehlungen empirisch überprüft werden. In Form von zwei Simulationen soll dabei gezeigt werden, dass bei Einhaltung der Handlungsempfehlungen eine Verbesserung des Energieverbrauchs von containerisierten Anwendungen erreicht werden kann. Darüber hinaus werden die Handlungsempfehlungen zur Erstellung eines Tools zur Verbesserung von Skalierungspunkten genutzt, welches Teil III des in Unterkapitel 6.1 dargestellten Modells entspricht.

8.1.1 Erste Ersparnis-Simulation

Wie zuvor beschrieben, eignen sich REST-APIs als realitätsnahes Fallbeispiel für eine Simulation. REST-APIs stellen Schnittstellen zwischen Maschinen dar, die heutzutage gerade in der modularen Programmierung immer mehr an Bedeutung gewinnen. Dies ist unter anderem auch bei der Kommunikation zwischen Docker Host und Docker Daemon zu erkennen, die ebenfalls auf einer REST-API basiert.

8.1.1.1 Aufbau des ersten Testszenarios

Eine solche REST-API wird in Verbindung mit den Umfragen der Cloud Monitoring Dienstleister Sysdig [sysdig.com, 2021] und Datadog [datadoghq.com, 2018], in denen Aussagen über die mittleren Lebens- und Updatezyklen von Containern gegeben werden, aufgebaut, sodass sich folgendes **Testszenario** ergibt:

- Eine REST-API mit dahinter liegender Datenbank,
- deren Webserver während einer Lastspitze skaliert wird.
- Der Server wird in der Nacht neu gestartet,
- wofür das Image neu erzeugt wird, um Änderungen zu übernehmen.
- Während der Erzeugung des Images wird die Software neu kompiliert.
- Parallel zum Betrieb wird an der Software entwickelt, was zu 50 Commits der Entwickler am Tag führt.

- Jeder Commit führt zu einem an die CI gebundenen Testbuild des Containers.

Des Weiteren steht die im Testszenario dargestellte REST-API nicht unter permanent gleichbleibender Belastung und soll zur Kommunikation zwischen Maschinen dienen. In der Datenbank werden sich daher Produktionsdaten befinden. **Für die Laufzeit REST-API werden zudem folgende Annahmen gemacht**:

- Die REST-API läuft 24 Stunden und 365 Tage im Jahr.
- Zwischen 8 und 18 Uhr steht die REST-API unter hoher Belastung und wird daher skaliert (zwei Millionen Zugriffe pro Stunde).
- Zwischen 5 und 8 sowie zwischen 18 und 22 Uhr wird eine mittlere Belastung verzeichnet (dreihunderttausend Zugriffe pro Stunde).
- In der Nacht wird nur vereinzelt auf die API zugegriffen (fünfzigtausend Zugriffe pro Stunde).
- Anfragen an die API sind zu gleichen Teilen lesend und schreibend.

Besonders beim letzten Punkt ergeben sich hier Unterschiede bei der Wahl des Storage-Treibers, da sich gezeigt hatte, dass *ZFS* bei schreibenden Zugriffen effizienter als *overlay2* arbeitet. Entsprechend den Anforderungen an das Simulationsszenario ergeben sich die in Tabelle 52 dargestellten Einstellungen bezüglich des Build-Prozesses sowie der Komponenten der Container **mit und ohne Berücksichtigung der Handlungsempfehlungen**.

Komponente	ohne Empfehlungen	mit Empfehlungen
Build-Prozess	klassischer Build	**Buildkit**
Build-Cache	keine Cache-Verwendung	**Cachebusting**
Verwendung von Compose	ja	**nein**
Volumes	ja, für die Datenbank	**für Server und Datenbank**
Storage-Treiber	default (overlay2)	**ZFS**
Netzwerk-Treiber	default (bridge)	default (bridge)
Logging-Treiber	default (json-file)	**Logging im Volume und Weiterleitung an System-Journal**
Images	golang:latest und mysql:latest	golang:latest und mysql:latest

Tabelle 52: Konfiguration der Container in Simulation 1 mit und ohne Handlungsempfehlungen

In dieser Simulation lassen sich zwei Handlungsempfehlungen nur bedingt überprüfen. Hierbei handelt es sich zum einen um die Nutzung einer geringen Anzahl von Basis-Images. Eine solche Anpassung macht nur dann Sinn, wenn mehrere Anwendungen auf einem großen Containercluster laufen, welche die gleiche Image-Basis verwenden, um entsprechende Auswirkungen deutlich erkennen zu

können. Da bei der Rest-API zwei unterschiedliche Container verwendet werden, nämlich ein Server und eine Datenbank, würden die dabei entstehenden Container das Gesamtbild deutlich verändern, da die Anforderungen an Datenbank und Server stark unterschiedlich sind.
Ein weiterer Punkt betrifft die Nutzung von OVS, auf die an dieser Stelle verzichtet wird, da die vorbereitenden Maßnahmen für einen in virtuellen Netzwerken ungeübten Administratoren die Verbesserungen durch die anderen Handlungsempfehlungen überwiegen würden. Es soll gezeigt werden, dass auch mit den einfachsten Anpassungen Vorteile beim Energieverbrauch erzielt werden können.

8.1.1.2 Messaufbau zur ersten Simulation

Der Messaufbau entspricht bezüglich der Bestimmung des Energieverbrauchs bei hoher, mittlerer und niedriger Last dem aus Abschnitt 7.2.8 oder Abschnitt 7.2.9. Hierbei wird der Service sechzig Sekunden unter Last gesetzt, während durchschnittliche Leistungsaufnahme, CPU- und RAM-Auslastung sowie die Anzahl an simultan zu verarbeitenden Zugriffen untersucht werden. Die Ergebnisse werden im Nachgang verwendet, um den Energieverbrauch in einem Jahr zu bestimmen, indem eine Hochrechnung entsprechend des Nutzungsszenarios durchgeführt wird. Zur Bestimmung des Energieverbrauchs bei der Erzeugung des Images und beim Start der Container wird der Messaufbau aus Abschnitt 7.2.3 beziehungsweise Abschnitt 7.2.4 verwendet. Hierbei wird zum einen die durchschnittliche Leistungsaufnahme sowie die Dauer des Prozesses bestimmt. Auch diese Daten werden entsprechend des Nutzungsplans auf ein Jahr hochgerechnet.

8.1.1.3 Messergebnisse bezüglich Build, Container-Start, mittlerer und voller Last des ersten Testszenarios

Im folgenden Abschnitt wird die reine Leistungsaufnahme des Testszenarios dargestellt. Dafür wurde im Vorhinein vor jeder Messung die Watt-Baseline des Systems ermittelt und von den Messwerten bei Ausführung des Testszenarios subtrahiert.
Dargestellt in Tabelle 53 ist die Dauer und die mittlere Leistungsaufnahme des Build-Prozesses. Hier ist eine deutliche Diskrepanz zwischen der Ausführung mit den Handlungsempfehlungen und ohne diese zu erkennen. Zwar führt die Konfiguration mit den Handlungsempfehlungen zu einer im Durchschnitt höheren Leistungsaufnahme, jedoch ist die Dauer des Prozesses um 35 Sekunden und damit fast 60% geringer als ohne die Nutzung der Handlungsempfehlungen (vergleiche Tabelle 53). Vor allem die Ausnutzung von Cachebusting sorgt dafür, dass die bereits vorher geladenen Go-Bibliotheken im Cache bleiben, während die Applikation trotzdem bei jeder Ausführung neu gebaut wird. Beim Start der Container ist zu erkennen, dass die Ausführung mit den Handlungsempfehlungen zum einen länger benötigt und zum anderen eine höhere Leistungsaufnahme mit sich bringt. Hier leidet die Vergleichbarkeit allerdings etwas, da durch den Aufbau der Applikation mit den Handlungsempfehlungen ein zusätzliches Volume für den Server erzeugt beziehungsweise verknüpft wird. Dadurch dauert der Startprozess des Servers länger und führt zu einer höheren Leistungsaufnahme. Da die Applika-

	durchschnittliche Dauer in Sekunden	durchschnittliche Leistungsaufnahme in W
ohne Handlungsempfehlungen	60,4	1,53
mit Handlungsempfehlungen	25,04	1,65

Tabelle 53: Build-Prozess des ersten Testszenarios mit und ohne Handlungsempfehlungen

tion allerdings nur ein Mal pro Tag neu gestartet wird, und die Verwendung des Volumes positive Auswirkungen auf die gesamte Laufzeit hat, hat dies kaum erkennbare Effekte auf den gesamten Energieverbrauch (vergleiche Tabelle 54). Im Bereich der geringen Auslastung ist erkennbar, dass die

	durchschnittliche Dauer in Sekunden	durchschnittliche Leistungsaufnahme in W
ohne Hw	7,8	2,04
Hw	8,3	2,42

Tabelle 54: Start der Container des ersten Testszenarios mit und ohne Handlungsempfehlungen

Handlungsempfehlungen zu einer geringeren CPU-Auslastung des Services führen. Allerdings ist in diesem Fall trotzdem eine um circa 3% höhere Leistungsaufnahme erkennbar (vergleiche Tabelle 55). Dies liegt daran, dass die Arbeit, die durch die schreibenden Zugriffe generiert wird, nicht die Arbeit der lesenden Zugriffe überschreitet. Daher wird durch ZFS eine zusätzliche Leistung generiert, die zu einer geringfügig höheren Leistungsaufnahme führt.

	durchschnittliche CPU-Auslastung in %	durchschnittliche RAM-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
ohne Hw	3,1	0,11	0,1	0,66
mit Hw	2,9	0,11	0,1	0,68

Tabelle 55: Ressourcenverbrauch der Container im ersten Testszenario bei niedriger Auslastung (fünfzigtausend Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen

Dieser Umschwung wird erst bei mittlerer Auslastung des Services erreicht. Man erkennt hier eine um 6% niedrigere CPU-Auslastung und eine um 3% geringere Leistungsaufnahme bei Verwendung der Handlungsempfehlungen (vergleiche Tabelle 56). Es ist ebenfalls auffällig, dass die Anzahl der simultanen Zugriffe ebenfalls um fast 6% verringert wurde. Bei hoher Auslastung zeigt sich eine weitere Steigerung der Ersparnis in Form von 7% weniger CPU-Auslastung und 10% geringerer Leistungsaufnahme (vergleiche Tabelle 57). Erneut sinkt auch die Anzahl an simultanen Zugriffen um 10%. Weiter ist zu erkennen, dass die RAM-Auslastung um 0,2% gestiegen ist. Auch hierbei ist diese Steigerung auf ZFS zurückzuführen, da die zu schreibenden Daten im Testszenario nicht nur einmal, sondern

	durchschnittliche CPU-Auslastung in %	durchschnittliche RAM-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
ohne Hw	11,6	0,11	0,52	1,29
mit Hw	10,9	0,11	0,49	1,25

Tabelle 56: Ressourcenverbrauch der Container im ersten Testszenario bei mittlerer Auslastung (dreihunderttausend Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen

gleich drei Mal geschrieben werden, was an der stets umzusetzenden Spiegelung des ZFS-Pools liegt. Insgesamt ergibt sich damit eine Energieersparnis von 13% in einem Jahr bei Verwendung der Hand-

	durchschnittliche CPU-Auslastung in %	durchschnittliche RAM-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
ohne Hw	38,9	0,12	2,0	2,9
mit Hw	36,3	0,14	1,8	2,6

Tabelle 57: Ressourcenverbrauch der Container im ersten Testszenario bei hoher Auslastung (zwei Millionen Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen

lungsempfehlungen. Wobei hier nicht die zusätzliche Ersparnis bezüglich CPU-Ressourcen und die Verbesserung der Erreichbarkeit des Services mit eingerechnet werden (vergleiche Tabelle 58).

	durchschnittlicher Energieverbrauch pro Tag	durchschnittlicher Energieverbrauch pro Jahr	Ersparnis pro Jahr in %
ohne Hw	48,51 Wh	17,71 kWh	
mit Hw	42,15 Wh	15,38 kWh	-13%

Tabelle 58: Energieverbrauch des ersten Testszenarios mit und ohne Handlungsempfehlungen

8.1.1.4 Interpretation der ersten Simulation

Betrachtet man die Ersparnis von 13%, kann die Umsetzung der Handlungsempfehlungen als Erfolg angesehen werden. Aus rein wirtschaftlicher Perspektive rentieren sich die gesparten 2,5 kWh pro Jahr jedoch kaum. Bedenkt man allerdings, dass entsprechend [sysdig.com, 2021] und [datadoghq.com, 2018] nicht einer, sondern im Durchschnitt mehr als 15 Container pro Host laufen, summiert sich ein solches Ergebnis. Auch muss bedacht werden, dass es sich bei der Applikation um REST-API handelt, die bei einer Anfrage nur wenige Daten ausliefert. Bei Applikationen, die viele Daten ausgeben, können sich wiederum ganz andere Ersparnisse ergeben. Darüber hinaus kann festgehalten werden, dass die Nutzung der Handlungsempfehlungen nicht nur zu einer Verbesserung des Energieverbrauchs, sondern auch zu einer besseren CPU-Auslastung sowie einer besseren Erreichbarkeit des

Services führt. Es lässt sich jedoch darüber diskutieren, ob die Verwendung von ZFS im Testszenario dringend notwendig ist, da sie zum einen die Ersparnisse verringert, da lesende Zugriffe, wie in Abschnitt 7.2.8 beschrieben, zu einer schlechteren Leistungsaufnahme führen und zum anderen auch die RAM-Auslastung steigert. Allerdings bringt ZFS nicht nur eine Verbesserung bei schreibenden Zugriffen mit sich, sondern führt darüber hinaus zu einer gespiegelten, redundanten und damit besser gesicherten Datenbasis. Alles in allem kann damit die Verwendung der Handlungsempfehlungen im ersten Testszenario als erfolgreich angesehen werden.

8.1.2 Zweite Ersparnis-Simulation

Wie bereits bei der ersten Ersparnis-Simulation wird im Folgenden ein Testszenario aufgestellt, welches einen typischen im Container ausgeführten Service darstellt. Anschließend findet eine Messung und eine Hochrechnung des Energieverbrauchs innerhalb eines Jahres statt.

8.1.2.1 Aufbau des zweiten Testszenarios

Das zweite Testszenario soll eine Nachrichtenseite darstellen. Hierfür wird zum einen eine Datenbank mit verschiedenen Meldungen bereitgestellt, die beim Aufruf der Seite geladen werden müssen. Zum anderen muss der Server an einen FPM-Service angehangen werden, sodass PHP-Dateien ausgeführt werden können. Damit spiegelt dieses Testszenario auch die „One Container, one Process“-Policy von Docker wider. Das Testszenario ist dabei wie folgt gestaltet:

- Eine dynamische Webseite (Nginx-Server + PHP-FPM) mit dahinter liegender Datenbank,
- deren Webserver während einer Lastspitze skaliert wird.
- Der Server wird in der Nacht neu gestartet,
- wofür das Image neu erzeugt wird, um Änderungen zu übernehmen.
- Parallel zum Betrieb wird an der Software entwickelt, was zu fünfundzwanzig Commits der Entwickler am Tag führt.
- Jeder Commit führt zu einem an die CI gebundenen Testbuild des Containers.

Für die Nachrichtenseite werden zusätzlich die nachfolgenden Annahmen gemacht, die mit den Nutzerzahlen anderer Newsportale wie Golem.de oder Heise.de abgeglichen wurden (vergleiche [SimilarWeb, 2021]):

- Die Webseite läuft 24 Stunden und 365 Tage im Jahr.

- Zwischen 8:00 und 18:00 Uhr steht die Webseite unter hoher Belastung und wird daher skaliert (40.000 Besuche pro Stunde, wobei jeder Nutzer im Durchschnitt 5 Unterseiten aufruft).
- Zwischen 5:00 und 8:00 sowie zwischen 18:00 und 22:00 Uhr wird eine mittlere Belastung verzeichnet (25.000 Zugriffe pro Stunde, ebenfalls im Durchschnitt fünf Unterseiten besucht).
- In der Nacht wird nur vereinzelt auf die Webseite zugegriffen (circa 100 Zugriffe pro Stunde).
- Anfragen an die API sind ausschließlich lesend. Redaktionelle Änderungen werden nicht betrachtet.

Entsprechend der Anforderungen an das Simulationsszenario ergeben sich die in Tabelle 59 dargestellten Einstellungen bezüglich Build-Prozess und Komponenten der Container **mit und ohne Berücksichtigung der Handlungsempfehlungen**.

Komponente	ohne Empfehlungen	mit Empfehlungen
Build-Prozess	klassischer Build	**Buildkit**
Build-Cache	default Build Cache	**Cachebusting**
Verwendung von Compose	ja	**nein**
Volumes	ja, für die Datenbank	**für Server, PHP-FPM und Datenbank**
Storage-Treiber	default (overlay2)	default (overlay2)
Netzwerk-Treiber	default (bridge)	default (bridge)
Logging-Treiber	default (fluentd)	**Logging im Volume und Weiterleitung an System**
Images	nginx:latest, php:7.3-fpm und mysql:latest	nginx:latest, php:7.3-fpm und mysql:latest

Tabelle 59: Konfiguration der Container in Simulation 2 mit und ohne Handlungsempfehlungen

Als Handlungsempfehlungen werden hier vor allem die Nutzung mehrerer Volumes sowie eines anderen Logging-Treibers empfohlen, weswegen der Logging-Treiber *none* gewählt wird. Alle Logs fließen in Dateien, die ebenfalls in den Volumes gespeichert und nachträglich an das System-Journal übergeben werden. Zusätzlich dazu ergibt sich bei Veränderung der Webseite keine Notwendigkeit eines Rebuilds des genutzten nignx-Images. Da die Daten nicht persistent im Container liegen, kann daher auf die Rebuilds nach den Commits der Entwickler verzichtet werden.

8.1.2.2 Messaufbau zur zweiten Simulation

Der Messaufbau des ersten Testszenarios wird an dieser Stelle übernommen. Auf diese Weise ist ein Vergleich der beiden Ergebnisse möglich, sodass auch eine Aussage über den Effekt der Software im Container auf den Energieverbrauch getätigt werden kann.

8.1.2.3 Messergebnisse bezüglich Build, Container-Start, mittlerer und voller Last des zweiten Testszenarios

Wie bereits in Abschnitt 8.1.2 angedeutet, liegen die Daten der Webseite bei Anwendung der Handlungsempfehlungen nicht persistent im Container. Daher ergibt sich eine hundertprozentige Ersparnis bei den Rebuilds (siehe Tabelle 60). Werden keine Volumes benutzt, so wird ein Dockerfile verwendet, um bei Veränderung der Webseite ein neues Image mit persistenten Daten zu erzeugen.

	durchschnittliche Dauer in Sekunden	durchschnittliche Leistungsaufnahme in W
ohne Handlungsempfehlungen	8,36	1,85
mit Handlungsempfehlungen	kein Build benötigt	kein Build benötigt

Tabelle 60: Build-Prozess des zweiten Testszenarios mit und ohne Handlungsempfehlungen

Beim Start der Container wird bei Anwendung der Handlungsempfehlungen auf Docker-Compose verzichtet. Wie sich in Tabelle 61 darstellt, sinkt die Leistungsaufnahme durch die Verwendung von docker run um circa 11,5%. Im Gegenzug dazu steigt jedoch die Ausführdauer um etwas mehr als eine Sekunde an. Dies ist Resultat der Verwendung zusätzlicher Volumes, die beim Start der Container verknüpft werden müssen. Insgesamt gleichen sich diese Unterschiede beim Energieverbrauch aus.

	durchschnittliche Dauer in Sekunden	durchschnittliche Leistungsaufnahme in W
ohne Hw	9,49	2,53
Hw	10,574	2,24

Tabelle 61: Start der Container des zweiten Testszenarios mit und ohne Handlungsempfehlungen

Steht die Webseite unter mittlerer Auslastung, zeigt sich bei Verwendung der Handlungsempfehlungen eine Verringerung der CPU-Auslastung um 3% (vergleiche Tabelle 62). Als Ergebnis verringert sich auch die Leistungsaufnahme um 5%, während die Anzahl simultaner Zugriffe leicht ansteigt.

	durchschnittliche CPU-Auslastung in %	durchschnittliche RAM-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
ohne Hw	2,63	0,1	0,16	0,98
mit Hw	2,55	0,1	0,19	0,93

Tabelle 62: Ressourcenverbrauch der Container im zweiten Testszenario bei mittlerer Auslastung (25.000 Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen

Bei hoher Auslastung der Webseite werden Server und FPM-Service skaliert. Wie Tabelle 63 zeigt, sinkt die CPU-Auslastung bei Nutzung der Handlungsempfehlungen um 3%. Dadurch ergibt sich

	durchschnittliche CPU-Auslastung in %	durchschnittliche RAM-Auslastung in %	Anzahl simultaner Zugriffe	durchschnittliche Leistungsaufnahme in W
ohne Hw	3,35	0,1	0,11	1,06
mit Hw	3,26	0,1	0,123	1,0

Tabelle 63: Ressourcenverbrauch der Container im zweiten Testszenario bei hoher Auslastung (40.000 Zugriffe pro Stunde) mit und ohne Handlungsempfehlungen

eine Verringerung der Leistungsaufnahme um ebenfalls 5%. Fasst man die Ergebnisse zusammen und errechnet die Verbrauchswerte im Testszenario für ein Jahr, so ergibt sich eine Ersparnis von 0,6 kWh. Dies entspricht einer Ersparnis von 9% bei Verwendung der Handlungsempfehlungen.

	durchschnittlicher Energieverbrauch pro Tag	durchschnittlicher Energieverbrauch pro Jahr	Ersparnis pro Jahr in %
ohne Hw	18,35 Wh	6,7 kWh	
mit Hw	16,67 Wh	6,1 kWh	-9%

Tabelle 64: Energieverbrauch des zweiten Testszenarios mit und ohne Handlungsempfehlungen

8.1.2.4 Interpretation der zweiten Simulation

Auch im zweiten Testszenario kann die Nutzung der Handlungsempfehlungen als Erfolg angesehen werden. Auch wenn nur 0,6 kWh gespart werden können, was in Deutschland circa 15 Cent entspricht, findet eine Reduzierung des Energieverbrauchs statt. Man muss zudem bedenken, dass die im Testszenario verwendete Webseite nur bedingt Inhalte ausliefert, während eine redaktionell gepflegte Nachrichtenseite viele unterschiedliche Medien enthält (exemplarisch Foto, Videos), die bei Aufruf der Seite geladen werden müssen. Insgesamt kann daher davon ausgegangen werden, dass im Realfall eine deutlich höhere Ersparnis zu erwarten ist.

8.1.3 Nutzung der Handlungsempfehlungen mit anderen Container-Systemen

Die in Kapitel 8 vorgestellten Handlungsempfehlungen lassen sich auf alle Systeme übertragen, die Docker oder containerd als Container-System verwenden. Hierzu zählt beispielsweise Kubernetes, das sich aktuell als das am stärksten wachsende Clusterwerkzeug darstellt und dessen Wichtigkeit durch die Reichweite der CNCF stetig zunimmt (vergleiche hierzu Abschnitt 3.4.5). An dieser Stelle können weitere Systeme wie Rancher, RedHat Openshift, Apache Mesos, OpenStack und viele mehr genannt werden, die Docker voraussetzen oder dessen Verwendung unterstützen. Sofern die administrativen Kenntnisse es zulassen, können hier den Handlungsempfehlungen entsprechende Konfigurationen umgesetzt werden. Zudem lassen sich die Handlungsempfehlungen bezüglich der Netzwerke, Logging, Volume, Storage Treiber und Build-Prozess auf reines containerd ohne aufgesetztes Docker

übertragen (vergleiche hierzu [The Linux Foundation, 2019b]), sodass ebenfalls Systeme profitieren die von der OCI gefördert werden und nicht auf Docker setzen. Man muss hier jedoch bedenken, dass auch hier entsprechende Kenntnisse vorausgesetzt werden, da beispielsweise Netzwerk-Treiber nicht out-of-the-box zur Verfügung gestellt werden.
Auf Container-Runtimes und auf Container-Systeme, die beide dem OCI-Standard entsprechen (Standard bezüglich Image-Spezifikation und Runtime), lassen sich die Handlungsempfehlungen sehr einfach übertragen. Die Übertragung auf andere Container-Werkzeuge, die diesen Standards nicht entsprechen, gestaltet sich schwieriger, da zwar jedes Container-Werkzeug im Kern mit den selben Mechaniken arbeitet, aber auf Seiten des Daemons Unterschiede mit sich bringt. So setzt das relativ neue Tool Podman beispielsweise im Gegensatz zu Docker nicht auf eine Client-Server-Architektur, sondern auf eine Fork-Exec-Mechanik (Container ist echter Kindprozess von Podman). Darüber hinaus bringt jedes alternative System auch zusätzliche Komponenten in den Bereichen Netzwerk, Storage und Logging mit sich, die überprüft werden sollten. Hierfür kann die Messmethodik einfach auf diese Systeme angewendet werden. Allerdings hat sich auch gezeigt, dass Alternativen zu Docker, wie zum Beispiel rkt von CoreOs, häufig nur kurz überleben.

8.1.4 Transfer des Modells auf Orchestrierungstools (Modell-Kernbestandteil III)

Die zuvor aufgestellten Handlungsempfehlungen sowie die Messmethodik und die Messumgebung lassen sich auf andere Anwendungsbereiche transferieren. Nimmt man beispielsweise die Messergebnisse der zweiten Simulation sowie die Methodik zur Messung des Energieverbrauchs, so lässt sich, wie im Folgenden gezeigt, ein Skalierungspunkt für jenen Anwendungsfall bestimmen, der sowohl die Anforderungen an die Erreichbarkeit des Services erfüllt, als auch zu einer Energieersparnis führt. Im Folgenden wird daher kurz dargestellt, wie sich die Modellteile I und II auf den Kubernetes Horizontal Pod Auto Scaler transferieren lassen, um so ein Tool, welches Modellteil III repräsentiert, für die Wahl von effizienten Skalierungspunkten zu erstellen.

8.1.4.1 Transfer auf den Kubernetes Horizontal Pod Auto Scalers

Kubernetes ist eine der meistgenutzten Orchestrierungsplattformen für Container und besitzt, wie in Abschnitt 3.4.6 gezeigt, die Fähigkeit zur automatisierten horizontalen sowie vertikalen Skalierung der Container bzw. Containerverbünde (Pods). Im Falle der vertikalen Skalierung kann, wie ebenfalls in Abschnitt 3.4.6 beschrieben, auf verschiedene Metriken zurückgegriffen werden, anhand derer der Service skaliert wird. Hierbei wird ein durch Erfahrungswerte entwickelter Wert angegeben, zum Beispiel 50 CPU-Auslastung, an welchem der Service skalieren soll. Durch die Messungen zur Aufstellung der Handlungsempfehlungen in Kapitel 8 und Kapitel 7 hat sich die Möglichkeit ergeben, den Skalierungspunkt unabhängig von Erfahrungswerten und in Abhängigkeit von Energieverbrauch und Erreichbarkeit des Services zu ermitteln. Erreichbarkeit ist dabei ein sehr wichtiger Faktor, denn laut [sysdig.com, 2021] ist die Antwortzeit eines Services die am meisten überwachte Eigenschaft eines

Dienstes. Genaueres über die Funktionalität des Tools wird in Absatz 8.1.4.2 dargestellt.

8.1.4.2 Funktionaler Aufbau des Skalierungstools

Das hier entwickelte Tool verändert nicht den Skalierer von Kubernetes, wie beispielsweise [Casalicchio, 2019], sondern setzt ein Modul auf das vorhandene System von Kubernetes auf. Dies bedeutet, dass das Tool nicht aktiv auf Kubernetes zugreift, dem Nutzer jedoch vor dem Starten des Services einen Handlungsspielraum mit entsprechenden Vorteilen für Veränderungen bei der Skalierung aufzeigt.
Zu Beginn wird eine Messung durchgeführt, die die Leistung und vor allem Grenzen eines Services bei ansteigender Zugriffszahl aufzeigt. Hierfür wird der Service gestartet und in fünfzehnsekündigen Intervallen mit linear ansteigenden Zugriffszahlen belastet. Dies wird solange durchgeführt, bis entweder die Anzahl verarbeiteter Zugriffe stagniert oder die CPU beziehungsweise der RAM voll ausgelastet ist. Durch diese Messungen werden dann folgende Daten ermittelt:

i) Obere Auslastungsgrenze des Services.

ii) Änderungsverlauf der Erreichbarkeit des Services in Abhängigkeit zur Auslastung, in Form von simultan zu verarbeitenden Zugriffen.

iii) Änderungsverlauf der Leistungsaufnahme des Services in Abhängigkeit zur Auslastung.

Die gleiche Messung wird nun mit dem skalierten Service erneut durchgeführt, um so ebenfalls die Verläufe und die Auslastungsgrenzen zu bestimmen. Es bestehen nun mehrere Möglichkeiten, diese Daten weiterzuverarbeiten. Ausarbeitungen wie [Imdoukh et al., 2019] setzen bei der Auswertung der Daten auf Machine Learning. Dieser Ansatz wird jedoch aus zwei Gründen nicht weiterverfolgt: Der Trainingsprozess beim Machine Learning verbraucht so viel Energie, dass sich eine solche Herangehensweise nur für Großbetreiber anbieten würde, um entsprechende Ergebnisse zu erzielen. Da die Software innerhalb des Containers mit ausschlaggebend für den Energieverbrauch ist, und jeder Service seine eigene Charakteristik mit sich bringt, müsste für jeden neuen Service ein eigener Trainingsprozess angestoßen werden. Zudem soll das Tool mit einer überschaubaren Menge an Daten lauffähig sein.
Aus diesen Gründen werden die Daten mit möglichst einfachen Mitteln verarbeitet und ausgewertet. Das Tool arbeitet dabei in der nachfolgend aufgelisteten Art und Weise. Die Auflistung wird durch die Beispiele in Abbildung 45 - Abbildung 48 grafisch unterstützt:

i) Erhobene Daten hinsichtlich Erreichbarkeit und Leistungsaufnahme, sowohl unskaliert als auch skaliert, normalisieren

ii) Jeweils mit Hilfe von kubischen Splines Daten glätten

iii) Durch Interpolation Polynome der Verlaufskurven bestimmen (siehe Abbildung 45 und Abbildung 46)

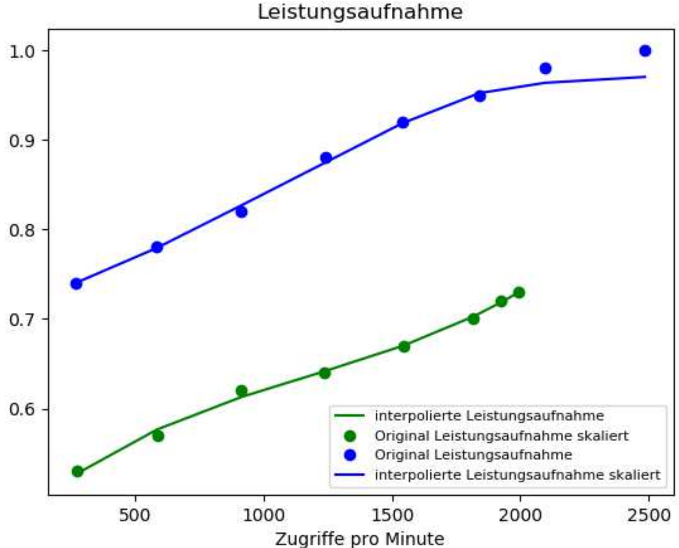

Abbildung 45: Leistungsaufnahme für einen skalierten bzw. unskalierten Service (normalisiert)

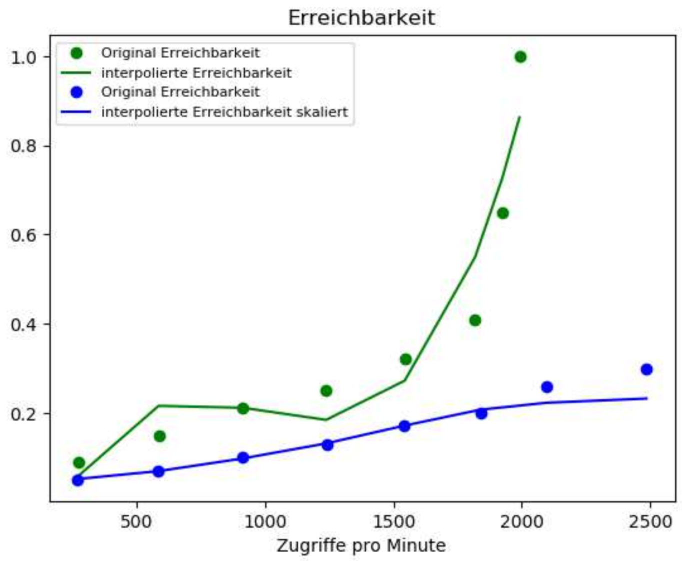

Abbildung 46: Simultane Zugriffe auf einen skalierten bzw. unskalierten Service (normalisiert)

iv) Deltafunktionen der Erreichbarkeit und Leistungsaufnahme zwischen skaliertem und unskaliertem Service berechnen

v) Minimum der Leistungsaufnahme-Deltafunktion und Maximum der Erreichbarkeit-Deltafunktion bestimmen

vi) Maximum der Erreichbarkeit-Deltafunktion halbieren

vii) Mittelwert zwischen Minimum der Leistungsaufnahme-Deltafunktion und Maximum der Erreichbarkeit-Deltafunktion bilden (siehe Abbildung 47)

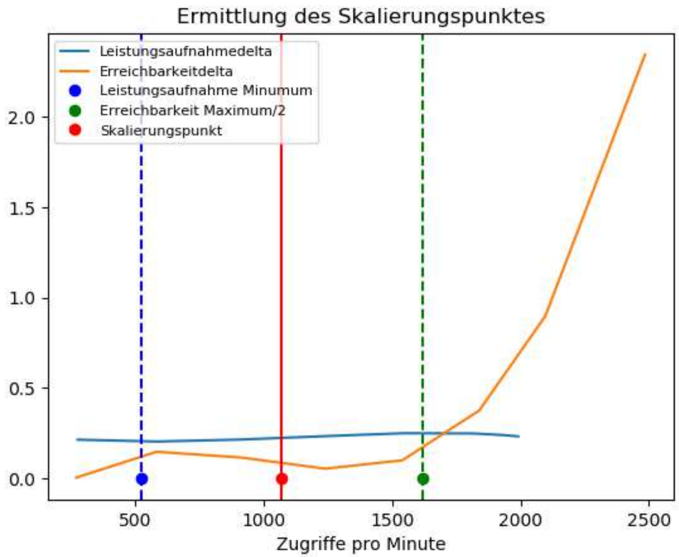

Abbildung 47: Ermittlung des Skalierungspunktes aus den Deltas der Leistungsaufnahme und der simultanen Zugriffe (normalisiert)

viii) Über den erhaltenen Skalierungspunkt die erlaubten Zugriffe pro Minute bestimmen

ix) Mit Hilfe der erlaubten Zugriffe pro Minute den Skalierungspunkt bezüglich der CPU-Auslastung bestimmen (siehe Abbildung 48)

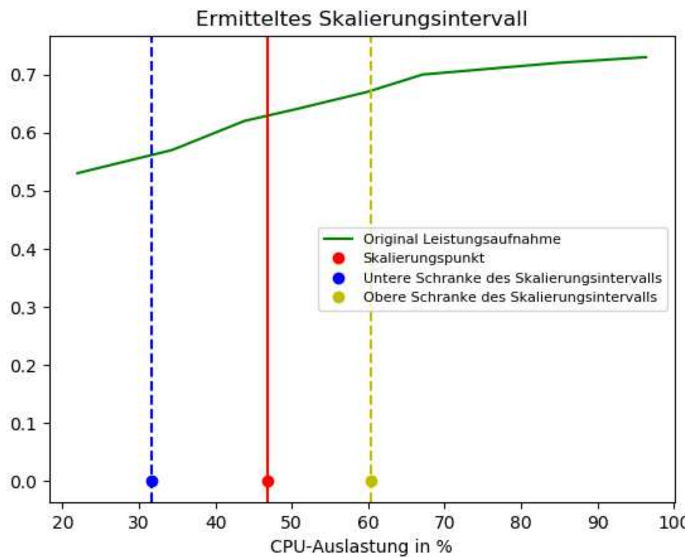

Abbildung 48: Ermitteltes Skalierungsintervall in CPU-Prozent in Relation zu Leistungsaufnahme und simultanen Zugriffen (normalisiert)

Da die Daten bezüglich Leistungsaufnahme und Erreichbarkeit in zwei unterschiedlichen Wertebereichen liegen, findet zu Beginn eine Normalisierung der Daten statt. Weiter werden kubische Splines der Daten gebildet, um somit die Daten zu glätten und abschließend durch Polynominterpolation weiter verwertbare Funktionen zu erzeugen. An dieser Stelle werden zuerst Splines verwendet, da diese deutlich stabiler als Polynome sechsten Grades sind, die sich im Verlauf der Entwicklung am geeignetsten für die Daten gezeigt haben. Wie in Abbildung 45 und Abbildung 46 erkennbar, erhält man auf diese Weise ausreichend genaue Funktionen bezüglich Leistungsaufnahme und Erreichbarkeit.
Weiter werden in einem nächsten Schritt jeweils die Deltas der Leistungsaufnahme- beziehungsweise Erreichbarkeitsfunktionen zu den skalierten Versionen gebildet. Man erkennt am Beispiel in Abbildung 47, dass das Delta im Falle der Leistungsaufnahme nur leicht ansteigend ist, während bei der Erreichbarkeit eine große Diskrepanz bezüglich skaliertem und unskaliertem Service besteht. Aus diesem Grund werden beide Werte im Folgenden unterschiedlich behandelt. Im Falle der Leistungsaufnahme kann davon ausgegangen werden, dass der optimale Skalierungspunkt am rechten Ende der Skala liegt, während bezüglich der Erreichbarkeit bereits eine Skalierung bei Start der Applikation sinnvoll erscheint. Aus diesem Grund werden alternative optimale Punkte für eine Skalierung gesucht. Geht man von der Leistungsaufnahme aus, sollte der Unterschied vor und nach einer Skalierung möglichst gering sein. Daher wird als alternativer optimaler Punkt das **Minimum der Leistungsaufnahme-Deltafunktion** gewählt. Es ist zu erwähnen, dass dieser Punkt in circa 50% aller Testfälle beim Minimum des Definitionsbereichs lag und zu 50% erkennbar größer war, was jeweils in Abhängigkeit zur getesteten Applikation stand. Applikationen, die wenige Daten ausliefern, zeichneten bei der Leistungsaufnahme deutlich konstantere Kurven als beispielsweise dynamische Webseiten. Bezüglich der Erreichbarkeit sollte davon ausgegangen werden können, dass eine Skalierung den Unterschied zwischen skaliertem und unskaliertem Service, bezogen auf simultanen Zugriffen auf einen Service, mindestens halbiert. Aus diesem Grund wird bei einer initialen Ausführung des Scripts **das Maximum der Erreichbarkeit-Deltafunktion bestimmt und halbiert**. Wird das Script erneut ausgeführt, ändert sich dieser Vorfaktor anhand von Kriterien, die weiter unten beschrieben werden.
Abschließend wird der Mittelpunkt der dadurch bestimmten CPU-Auslastungen errechnet, wodurch ein geeigneter Skalierungspunkt in Abhängigkeit zur CPU-Auslastung bestimmt wird, der in gleichen Maßen den Stromverbrauch als auch die Erreichbarkeit eines Services miteinbezieht. Weiter bilden **Minimum der Leistungsaufnahme-Deltafunktion** und **das halbierte Maximum der Erreichbarkeit-Deltafunktion** eine untere beziehungsweise obere Schranke für ein Intervall, in dem der Skalierungspunkt beliebig verschoben werden kann, um so weiterhin ein gesundes Verhältnis zwischen Energieverbrauch und Erreichbarkeit eines Services zu erzielen und trotzdem den Skalierungspunkt an die Anforderungen des Services anpassen zu können. Welche Auswirkung eine Verschiebung des Skalierungspunktes auf den Energieverbrauch und die Erreichbarkeit eines Services haben kann, wird exemplarisch in Absatz 8.1.4.3 gezeigt.
Befindet sich die maximale Auslastung dauerhaft über dem Skalierungsintervall aber unter der Aus-

lastungsgrenze des Services, so wird geraten, eine neue Bestimmung des Skalierungspunktes durchzuführen und hier die maximale Anzahl an Zugriffen als neue obere Auslastungsgrenze des Services anzusehen. Weiter wird hier jedoch der maximale Wert der simultanen Zugriffe nicht halbiert, sondern mit dem Faktor

$$\frac{obereAuslastungsgrenze_{neu}}{obereAuslastungsgrenze_{alt}} = \alpha_{concurrency}$$

multipliziert.

8.1.4.3 Simulation

Nachfolgend wird die Simulation des Autoscaling-Tools beschrieben. Hierfür wird auf ein bereits bekanntes Testszenario zurückgegriffen, welches im nächsten Abschnitt beschrieben wird.

Aufbau des Testszenarios

Als Ausgangsszenario wird das bereits bekannte Testszenario aus Abschnitt 8.1.2 gewählt. Im hier vorgestellten Testszenario läuft eine Newsseite, bestehend aus Webserver, Datenbank und FPM-Container, zehn Stunden am Tag in skaliertem Zustand. Nachfolgend wird präsentiert, welche Auswirkungen die Anwendung das Scaler-Tool auf die Ermittlung des Skalierungspunkt hat und welche Folgen sich für die Leistungsaufnahme ergeben.

Ergebnisse der Simulation

Wendet man das oben beschriebene Skript auf die Messergebnisse bezüglich des zweiten Testszenarios an, so erhält man ein Ergebnis entsprechend Abbildung 49 und Abbildung 50. Demnach liegt der optimale Skalierungspunkt bei gleichmäßiger Gewichtung von Energieeffizienz und Erreichbarkeit bei 64%. Man erkennt hier schon, dass dieser Wert, im Gegensatz zu den üblicherweise gewählten 50% CPU-Auslastung als Skalierungspunkt, schon höher ist.
Als Schranken des Skalierungsintervalls ergeben dabei folgende Werte:

- Unter Schranke bei 28% CPU-Auslastung (entspricht 8000 Zugriffen pro Minute)
- Obere Schranke bei 82% CPU-Auslastung (entspricht 38000 Zugriffen pro Minute)

Der optimale Skalierungspunkt von 64% CPU-Auslastung wird dabei bei circa 21600 Zugriffen pro Minute erreicht. Wird an diesem Punkt skaliert, ergibt sich eine mittlere Antwortzeit des Servers von 2,15 ms. Verschiebt man nun den Skalierungspunkt entsprechend dem Skalierungsintervall nach oben auf beispielsweise 78%, so erhält man als neue mittlere Antwortzeit des Server 4,8 ms.

Insgesamt verbrauchte Zeit für alle Zugriffe im unskalierten Zustand:

$$2,7 \text{ simultane Zugriffe im Durchschnitt } \cdot 60s = 162s$$

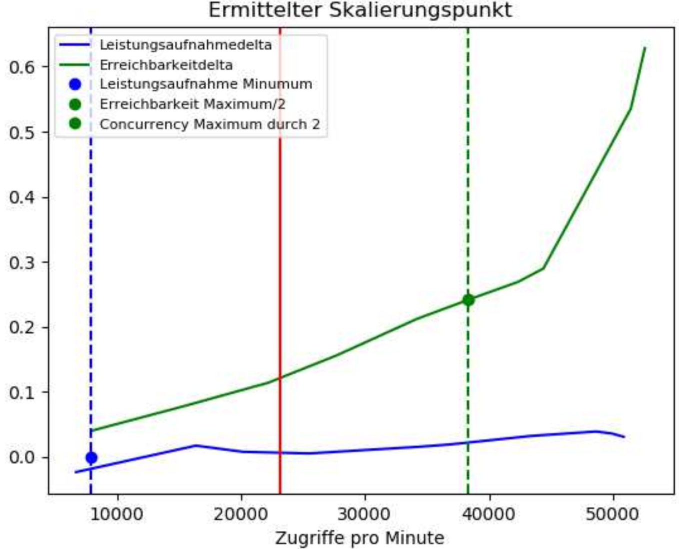

Abbildung 49: Bestimmung des Skalierungspunktes

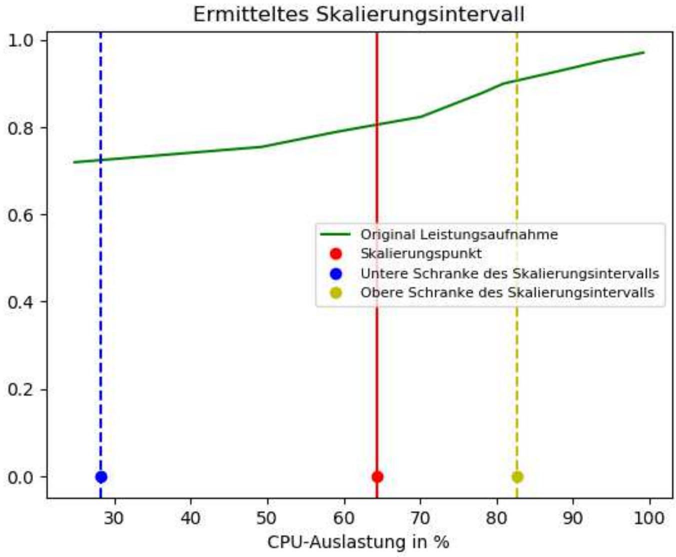

Abbildung 50: Bestimmung des Skalierungsintervalls

$$\Rightarrow \text{Dauer für einen Zugriff: } \frac{162s}{34000} = 0,0048s$$

Insgesamt verbrauchte Zeit für alle Zugriffe im skalierten Zustand:

$$1,22 \text{ simultane Zugriffe im Durchschnitt } \cdot 60s = 73,2s$$

$$\Rightarrow \text{Dauer für einen Zugriff: } \frac{73,2s}{34000} = 0,00215s$$

Im Gegenzug dazu ergibt sich aufgrund der späteren Skalierung eine Ersparnis beim Energieverbrauch, die sich wie folgt äußert: Bei einer Laufzeit von 10 Stunden ohne Skalierung und unter der Annahme, dass es im besten Fall nach Erhöhung des Skalierungspunktes nie zu einer Skalierung kom-

men muss, ergibt sich eine Energieersparnis von 2 Wh pro Tag und 730 Wh im Jahr. Dies entspricht einer Verringerung des Energieverbrauchs von 3%.

8.1.4.4 Fazit und Ausblick

Das vorgestellte Tool zeigt, dass das Modell auf andere und vor allem komplexere Anwendungsfälle transferierbar ist. Weiter wird dargestellt, dass eine geschicktere Auswahl des Skalierungspunktes eines containerisierten Services den Energieverbrauch dauerhaft senken kann. Aktuell ist das Tool nur eine unabhängige Erweiterung zum Kubernetes Horizontal Pod Autoscaler. Da K8 die Möglichkeit von Custom Plugins bietet, wäre es zukünftig möglich, ein solches Plugin zu programmieren, welches die Messung zum Start des Services automatisiert ausführt und dem Nutzer einen Skalierungspunkt vorschlägt. Auf diese Weise könnte der K8-Nutzer selbst entscheiden, ob ihm eine höhere Erreichbarkeit oder ein besserer Energieverbrauch wichtiger ist. Zudem könnten dem Nutzer die Auswirkungen seiner Wahl dargestellt werden.
Weiter wäre es in einem weiteren Schritt möglich, den Energieverbrauch als zusätzliche Custom Metric in K8 einzubinden. Auf diese Weise ließe sich die Berechnung des Skalierungspunkt, im Gegensatz zum aktuellen Vorgehen, vereinfachen. Zusätzlich könnte die Methodik auch auf standardmäßige Autoscaler großer Anbieter wie Azure oder AWS angewendet werden, die ebenfalls nach Metrikschwellenwerten basierend auf Nutzererfahrungen skalieren und den Energieverbrauch nicht betrachten.
Alles in allem ist jedoch festzuhalten, dass bei Anwendung der präsentierten Methode, auf mehrere Apps, der Energieverbrauch großer Container-Cluster verringert werden kann.

8.1.5 Hochrechnung der Ergebnisse auf Rechenzentren

Wie sich in den Simulationen zeigt, führt die Verwendung der Handlungsempfehlungen zu einer Energieersparnis von 13% bei einer REST-API beziehungsweise 9% bei einer Webseite mit dynamischen Inhalten. Umgerechnet auf die Geldersparnis können so einmal 50 beziehungsweise 15 Cent gespart werden. Auch wenn diese Ersparnis relativ betrachtet gering erscheint, muss hier jedoch die absolut gesehen große Masse an Containern bedacht werden, die in Cloud Rechenzentren zum Einsatz kommen.
Aktuell gibt es mehr als 100 Millionen Server, die weltweit verwendet werden [Racksolutions, 2021]. Alleine Google besitzt seit 2016 dabei mehr als 2,5 Millionen Server [datacenterknowledge.com, 2021]. Ein typisches Rechenzentrum hat dabei im Durchschnitt zwischen 50.000 und 80.000 Servern [Johnson, 2021]. Gehen wir nun davon aus, dass ein durchschnittliches Rechenzentrum mit 50.000 Servern circa 400.000 VMs bereitstellt. Auf 20% dieser VMs laufen Container (vergleiche [datadoghq.com, 2018]), was 80.000 VMs entspricht, wobei auf 83% dieser VMs auch Docker läuft (66.400 VMS) (vergleiche [sysdig.com, 2021]). Wie bereits zuvor erwähnt, laufen auf einem normalen Host im Mittel fünfzehn Container (vergleiche [sysdig.com, 2021]). Nehmen wir an, diese fünfzehn Con-

tainer entsprächen drei REST-APIs und zwei Webseiten, die wie in den Testszenarien gestaltet wären. Damit ergibt sich folgenden Rechnung für die Ersparnis in einem Jahr:

$$66.400 \text{ VMs} \cdot 3 \text{ REST-APIs} = 199.200 \text{ REST-APIs}$$
$$66.400 \text{ VMs} \cdot 2 \text{ Webseiten} = 132.800 \text{ Webseiten}$$

Daraus ergibt sich eine Energie-Ersparnis im Jahr von

$$199.200 \cdot 2,5kWh + 132.800 \cdot 0,6kWh = 577.680kWh$$

Dies wiederum entspricht bei einem Strompreis von 0,1 Euro pro kWh einer Ersparnis von **57.768 Euro pro Jahr** für ein mittleres Rechenzentrum. Natürlich muss dieses Ergebnis relativiert werden, da eine Vielzahl unterschiedlicher Services in den Rechenzentren laufen und beide Testszenarien nur exemplarisch sind. Zusätzlich dazu sind jedoch auch nicht die Vorteile mit eingerechnet, die die Auswahl des Skalierungspunktes nach Abschnitt 8.1.4 mit sich bringen würden. Bei Einrechnung würde der Energieverbrauch während der Lastspitzen, wie in Abschnitt 8.1.4 gezeigt, um weitere 3% sinken. Insgesamt lässt sich aber deutlich erkennen, welche Auswirkungen die Anpassungen der Containerlandschaft mit Hilfe der Handlungsempfehlungen haben können.

8.1.6 Zusammenfassung und Fazit

Im letzten Kapitel wurde der Erfolg der aufgestellten Handlungsempfehlungen empirisch untersucht und nachgewiesen, dass unter den untersuchten Rahmenbedingungen Ressourceneinsparungen möglich sind. Abgesehen davon wurde gezeigt, dass sich sowohl die Messmethodik und Messumgebung als auch die Handlungsempfehlungen auf andere Containersysteme sowie auf komplexere Sachverhalte wie den Autoscaler eines Containerorchestrierungstools übertragen lassen. Hiermit wurde das zuvor in Unterkapitel 6.1 aufgestellte Modell finalisiert. Zusätzlich dazu wurden die Auswirkungen einer praktischen Anwendung auf den Energieverbrauch von Cloud-Rechenzentren deutlich gemacht. Auch, wenn der Gesamtenergieverbrauch von Containern unter anderem abhängig vom Anwendungsszenario und der containerisierten Software ist, kann das nun final aufgestellte Modell zu einer Verbesserung des Energieverbrauchs von Containersystemen, und größer gefasst, Cloud-Rechenzentren, führen. Einer praktischen Anwendung steht nur der Zeitaufwand bei der Umsetzung gegenüber. Ziehen Anwender die Nutzung der Handlungsempfehlungen in Betracht, sollte stets überprüft werden, wie viele Container von den Handlungsempfehlungen profitieren können, um einen möglichst hohen Gewinn zu gewährleisten.

9 Fazit und Ausblick

Das moderne Rechenzentrum hat sich in den vergangenen Jahren einer massiven Entwicklung unterzogen. Die Containersoftware Docker hat dabei einen wesentlichen Anteil an der Entwicklung vom klassischen Rechenzentrum hin zur Cloud. Als Resultat dieser Entwicklung zeigt sich, dass Container die Basis einer riesigen Menge an Berechnungseinheiten innerhalb der Cloud sind, deren Umfang selbst durch Firmen wie Docker nicht bestimmt werden kann und die damit erheblich zum durch Software induzierten Stromverbrauch beiträgt. Bereits aktuell sind Rechenzentren für mehr als 1% des globalen Stromverbrauchs verantwortlich, wobei ein Ende des Anstiegs der durch IT verbrauchten Energie nicht in Sicht ist. Daher ist nicht nur aus Perspektive der aktuellen Klimaentwicklungen, sondern auch aus rein wirtschaftlicher Sicht eine Verbesserung des Energieverbrauchs von Containern ein erstrebenswertes Ziel.
Um dieses Ziel zu erreichen, wurde ein Modell für den energieeffizienten Einsatz von Containern, bestehend aus Messmethodik, Messumgebung und Handlungsempfehlungen sowie weiteren Tools postuliert. Hierzu wurden mit Hilfe einer Umfrage sowie zusätzlicher Recherche teils komplexe Standardnutzungsszenarien für Container erarbeitet, die Grundlage für die Erarbeitung einer flexiblen Messmethodik und Messumgebung sind, sodass mit dieser der Stromverbrauch von Containern bestimmt werden kann. Ihre Messskripte lassen sich auf alle Messgeräte mit Ethernet, SNMP-Schnittstelle und MIB-Tabelle anpassen. Wie sich zeigte, wird Docker selbst von Profis aufgrund der Simplizität größtenteils in Standardeinstellungen verwendet. Dies gilt sowohl für Administratoren als auch für Softwareentwickler, die somit immer häufiger Container verwenden. Als Resultat wurden daher die Kernkomponenten von Docker, ausgehend von der Sprache, in welcher die Software programmiert ist, auf ihre Energieeffizienz hin untersucht, sodass hieraus Handlungsempfehlungen für den Einsatz von Docker entwickelt werden konnten. Als Ergebnis war es möglich in 7 von 10 untersuchten Nutzungsszenarien Verbesserungen durch geschickte Konfiguration zu erreichen.
Die aus den Untersuchergebnissen resultierenden Handlungsempfehlungen zeigten sich in einer empirischen Überprüfung als erfolgreich, um dem Problem des steigenden Stromverbrauchs entgegenzuwirken. Hier konnten auf das Jahr gesehen 13% im Falle einer REST-API und 9% im Falle einer dynamischen Webseite eingespart werden. Des Weiteren wurde gezeigt, welche Auswirkungen dies bezüglich der Energiekosten auf Rechenzentren haben kann (siehe Fazit der Simulationen). Ebenfalls war es möglich, die Ergebnisse der Untersuchungen in Form eines Tools zur Bestimmung von Skalierungspunkten auf die Containerorchestrierung zu transferieren, welches das angestrebte Modell vervollständigte. Mit Hilfe dessen konnte ein besseres Verhältnis von Erreichbarkeit und Energieverbrauch gewährleistet werden, was zu weiteren Energieersparnissen führte. Exemplarisch konnten hier zusätzliche 3% eingespart werden.
Alles in allem kann somit das entwickelte Modell als erfolgreich anwendbar angesehen werden, da durch die Untersuchungen Potentiale aufgezeigt wurden, die es im Nachgang an diese Arbeit bei der

professionellen Entwicklung von Containertechnologien zu betrachten gilt. Daher könnte in Zukunft eine neue Container-Runtime entwickelt werden, die alle Vorteile der aktuell verwendeten Software mit sich bringt, jedoch energieeffizientere Module in den Bereichen Netzwerk, Storage und Logging nutzt. Gerade im Bereich des Docker-Loggings sollte eine Neuentwicklung mit einer energieeffizienteren und gleichzeitig auch zuverlässigeren Ausrichtung bedacht werden. Auch die Entwicklung eines Scaling-Tools mit gleichzeitiger Überarbeitung der allgemeinen Scaling-Policies könnte ein Ziel der Weiterentwicklung dieser Arbeit sein. Auf diese Weise hätten Cloud-Anbieter die Möglichkeit, flexiblere Preiskalkulationen zuzulassen. So kostet eine einzige Containerinstanz bei gängigen Cloud-Anbietern aktuell circa 30 Euro pro Monat. Mit einer Entwicklung weg von einer Abrechnung nach Laufzeit, hin zu einer Abrechnung nach Energieverbrauch, könnten diese Kosten deutlich gesenkt werden. Ein weiterer Vorteil, der sich dadurch ergeben würde, wäre die Verbesserung der CO2-Bilanzen von Cloud-Rechenzentren.

Abschließend ist festzuhalten, dass Container eine zukunftssichere Technologie darstellen, deren Potentiale noch nicht völlig ausgeschöpft sind. Die Zukunft der Containertechnologie wurde frühzeitig durch die Gründungen von OCI und CNCF sichergestellt. Auch in der Cloud sind Container eine Macht, denn mit dem immer weiter in den Fokus rückenden Orchestrierungstool Kubernetes steigen vermehrt Unternehmen auf die Cloud und damit Container um. Container im Allgemeinen bilden somit schon jetzt eine fast unverzichtbare Größe im Cloudgeschäft, die in den kommenden Jahren mit PaaS und vor allem FaaS weiter ansteigen wird und mit der Ausführung von Machine Learning as a Service sowie der Verwendung im Bereich IoT neue Geschäftsbereiche findet. Doch auch in der Softwareentwicklung eröffnen Container sowie im Speziellen Docker neue Möglichkeiten, die über die Auslieferung von Updates, Continuous Integration und Continuous Delivery hinausgehen. Im Bereich des Testings ergeben sich neue Optionen, die Entwickler bei der Qualitätssicherung von Software unterstützen. Insgesamt verschnellern Container damit die Entwicklung und Administration immer mehr.

Neben dem Potential bezüglich der Einsetzbarkeit ist auch das Potential für eine energieeffiziente Nutzung von Containern noch nicht ausgeschöpft. Auch wenn diese Technologie nicht den größten Energieverbrauch erzeugt, ergeben sich trotzdem erkennbare Möglichkeiten zur Einsparung enormer Mengen an Energie, die aus der großen Masse an Containern resultieren. Um dieses Potential auszuschöpfen, sind jedoch Administratoren und Entwickler gleichermaßen aufgerufen, ihre Arbeitsweise zu hinterfragen und diese umweltbewusster zu gestalten. Denn nur weil Ressourcen vorhanden sind, heißt es nicht, dass diese erst ausgeschöpft werden sollten, bevor ein Umdenken erfolgt. Zur Sensibilisierung für eine bewusste, langfristig angelegte Planung zum energieeffizienten Einsatz von Technologien, kann diese Arbeit somit einen Beitrag leisten.

Anhang: Messreihen, Code und Weiteres

Im Folgenden werden Inhalte präsentiert, die aufgrund ihres Umfangs nicht in Gänze in der Ausarbeitung dargestellt werden konnten. Hierzu zählt vor allem der verfasste Code und die vollständig erhobenen Messdaten.

Messreihen

Da die Messdaten selber in Form von CSV und XLS-Dateien vorliegen, werden diese in einem Online-Repository zur Verfügung gestellt, erreichbar über den folgenden Link:

`http://doi.org/10.5281/zenodo.3841074`

Code

Nachfolgend werden exemplarisch für den geschriebenen Code das grundlegende Messskript und das Scaler-Skript präsentiert. Weiterer Code, der verfasst wurde, sowie Beispiele für die verwendeten Container in Form von Dockerfiles und Docker-Compose-Files werden in einem Online-Repository zur Verfügung gestellt, das über den folgenden Link erreichbar ist:

`http://doi.org/10.5281/zenodo.3841123`

Messskript

```
#! /usr/bin/env python
#This program is free software: you can redistribute it and/or modify
#it under the terms of the GNU General Public License as published by
#the Free Software Foundation, either version 3 of the License, or
#(at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.
# Author Sandro Kreten
#Python 2.7 optimiert

from datetime import datetime
from subprocess import Popen, PIPE
import time
import sys
import os
import threading
import commands
import docker
import json

#############################################default IPs
    #######################################################################
PDU_IP='192.168.178.72'
DOCKER_HOST_API='192.168.0.2:2375'

#############################################default values
    ####################################################################
#Energy Variables
Volt=0
Ampere=0
PowerFactor=0
#Amount of Users
users=0
#Measurement Variables
number_of_measurements=0
sum_of_all_powers=0
#default delay
```

```
delay=0.2
#default hits
hits=100
#default duration
duration="15"
########################################variables for concurrent use
    ###########################################################
isAlive=0
siege_events=[]
flag=0
####################################################getCPUusage
    ################################################################
concurrency=""
hits=""
threads=[]
total_CPU=[]
total_RAM=[]
measuring_counter=[]
measuring_counter_ram=[]
numberOfContainer=0

############################################Overview over all
    Measurements###########################################################
all_m=[]

##Lese den CPU-Verbrauch des Containers ueber die Docker-API aus
def cpu_perc(d,number):

        if('system_cpu_usage' in d['precpu_stats']):
                cpuDelta=float(d["cpu_stats"]["cpu_usage"]["total_usage"])
                    -float(d["precpu_stats"]["cpu_usage"]["total_usage"])

                systemDelta=float(d["cpu_stats"]["system_cpu_usage"])-
                    float(d["precpu_stats"]["system_cpu_usage"])
                output = cpuDelta / systemDelta * 100
                global total_CPU
                total_CPU[number]=total_CPU[number]+output
                global measuring_counter
                measuring_counter[number]=measuring_counter[number]+1
                return output
        else:
                return 0

##Lese den CPU-Verbrauch des Containers ueber die Docker-API aus
```

```
def ram_perc(d,number):
        if('usage' in d['memory_stats']):
                cpuDelta=float(d["memory_stats"]["usage"])

                systemDelta=float(d["memory_stats"]["limit"])
                output = cpuDelta / systemDelta
                global total_RAM
                total_RAM[number]=total_RAM[number]+output
                global measuring_counter_ram
                measuring_counter_ram[number]=measuring_counter_ram[number
                   ]+1
                return output
        else:
                return 0

##Lese die Stats des Containers aus, bis das Event getriggert wird
def getStatsOfContainer(id,number,event):
        global isCPUAlive
        output=""
        count=0
        flag=0
        for times in client.stats(id,False,True):
                data = json.loads(times)
                output=output+datetime.now().strftime("%Y-%m-%d %H:%M:%S.%
                   f; ")+str(cpu_perc(data,number))+"; "+str(ram_perc(data
                   ,number))+"\n"
                if event.is_set():
                        break

##########################################SNMP GET ENERGY DATA
    #################################################################
def getSNMP(OID, unit):
        global PDU_IP
        process = Popen("snmpget -v1 -c private "+str(PDU_IP)+" "+str(OID)
            ,stdout=PIPE, shell=True)
        while True:
                line=process.stdout.readline().rstrip()

                if not line:

                        break
                line=line.split(":")
                return line[1]
```

```
###############################################SIEGE LOAD
    ######################################################################
def produce_Webserver_load(p, event , u):
        global delay
        global hits
        global concurrency
        global duration
        #print("Siege is started with "+str(users)+" Users and Delay of"+
            str(delay)+"!!!!!")
        #siege command
        command ="siege -c "+str(u)+" -d "+str(delay)+" -t"+duration+"s "+
            p
        #os.system(command)
        status, out=commands.getstatusoutput(command)
        out=out.split('$')
        print(out[1])
        out1=out[1].split(':')
        out1=out1[1].split(' ')
        hits=out1[0]
        print(out[2])
        out2 = out[2].split(':')
        concurrency=out2[1]
        #shows that siege is finished
        event.set()




###############################################START PROGRAMM
    ##################################################################
#get command line data
filename = "default_energy.log"
ports = []
if len(sys.argv)>3:
        filename=str(sys.argv[1])
        users=str(sys.argv[2])
        delay=str(sys.argv[3])

#read all given ports
for i in range(4,len(sys.argv)):
                ports.append(str(sys.argv[i]))

#remote Client
api_ip=ports[0].split(':')
```

```
#print(api_ip)
DOCKER_HOST_API="http:"+api_ip[1]+":2375"

client = docker.APIClient(base_url=DOCKER_HOST_API)
#print client.containers()
#open file and user input
file = open(filename,"a")
name_of_measurement=raw_input("What do you measure? ")
file.write(name_of_measurement+"\n")
##soll mit stets gleich Last gerechnet werden oder soll die Last nach
    jeder Messung linear gesteigert werden?
linear_or_static = int(input("Should the load be linear [1] or static [2]?
     "))
file2 = open("overall.csv","a")
file2.write(name_of_measurement+datetime.now().strftime(" %Y-%m-%d %H:%M:%
    S.%f")+"\n")
time.sleep(0.5)

print("
    ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

###########################################start siege-process for each
    given port (container that uses this port)################
for x in range(2,20,2):

        if linear_or_static==1:
                u=x
        else:
                u=users

        print "User:" + str(u) + " Delay:" + str(delay) + " Duration:" +
            duration
        #######Event to stop Resource-Collection#####################
        event = threading.Event()
        i=0
        ########################Array for all Container-Resources and
            Starting of Threads###########################
        for container in client.containers():
                print(container["Names"])
                total_CPU.append(0)
                total_RAM.append(0)
                measuring_counter.append(0)
                measuring_counter_ram.append(0)
```

```
            threads.append(threading.Thread(target=getStatsOfContainer
                , args=(container["Id"], numberOfContainer, event,)))
            threads[numberOfContainer].start()
            numberOfContainer=numberOfContainer+1
    ########################Start Siege Threats
        ################################
    for member in ports:
            ts = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f;
                startTestrun\n")
            file.write(ts)
            ###############Events to realize if all Siege Threads
                terminated####################################
            siege_events.append(threading.Event())
            threading.Thread(target=produce_Webserver_load,args=(
                member,siege_events[i],u,)).start()
            i=i+1

    while True:
            ##############################Gude
                ##########################################
            # Ampere
            Ampere = getSNMP("1.3.6.1.4.1.28507.43.1.5.1.2.1.5.1", "mA
                ")
            # Volts
            Volt = getSNMP("1.3.6.1.4.1.28507.43.1.5.1.2.1.6.1", "V")
            # PowerFactor
            PowerFactor = getSNMP("1.3.6.1.4.1.28507.43.1.5.1.2.1.8.1"
                , "PF")
            # calculate the active Watts
            activeWatt = float(int(Volt) * int(Ampere) * int(
                PowerFactor)) / (1000 * 1000)

            #print(str(Volt) + " " + str(Ampere) + " " + str(
                PowerFactor) + "\n")

            #####################################count all measurements
            sum_of_all_powers+=activeWatt

            number_of_measurements+=1
            #print(str(activeWatt)+" "+str(number_of_measurements)+"
                "+str(sum_of_all_powers))

            ####################################Wait on all Ports/
                Containers to be no longer under Stress
```

```
                        #########################
                for e in siege_events:
                        if e.is_set():
                                isAlive=isAlive+1
                                print(isAlive)
                if isAlive>=(len(sys.argv)-4):
                        event.set()
                        ts = datetime.now().strftime("%Y-%m-%d %H:%M:%S.%f
                           ;stopTestrun\n")
                        file.write(ts)
                        break
                #time.sleep(0.5)

        ##############################################Print Watts
           #########################################################

        avg=round(sum_of_all_powers/number_of_measurements,2)

        print "\nAverage Watts: "+str(round(sum_of_all_powers/
           number_of_measurements,2))+" Watts"
        all_m.append(avg)
        file2.write("User:"+str(u)+" Delay:"+str(delay)+" Duration:"+
           duration+"s; "+str(avg)+"; "+str(hits)+"; "+str(concurrency)+";
            ")

        ########################################Print Container Resources
           #################################################
        counter = 0
        for container in total_CPU:
                avg_cpu=round(container/measuring_counter[counter],2)
                avg_ram=round(total_RAM[counter]/measuring_counter_ram[
                   counter],2)
                print("Container "+str(counter+1)+" CPU: "+str(avg_cpu)+"
                   RAM: "+str(avg_ram))
                file2.write(str(avg_cpu)+"; "+str(avg_ram)+";")
                counter=counter+1

        file2.write("\n")
        #########################################warte und setze alles
           zurueck#########################################
        time.sleep(1)
        number_of_measurements=0
        sum_of_all_powers=0
        isAlive=0
```

```
        siege_events = []
        threads = []
        total_CPU = []
        total_RAM = []
        measuring_counter = []
        measuring_counter_ram = []
        numberOfContainer = 0
        print("
            ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")
        time.sleep(10)

        if(linear_or_static==0 and x==10):
                break

avg_sum=0
for m in all_m:
        #print(str(m))
        avg_sum=avg_sum+m
print("Average used Power in Watts: "+str(round(avg_sum/len(all_m),2)))
file.close()
file2.close()
```

Scaling-Advisor

```
#Author Sandro Kreten
import scipy.integrate
import numpy as np
import matplotlib.pyplot as plt
import sys
from scipy import interpolate
from scipy import optimize

############ Messung vor der Skalierung
    ####################################

## Messungen Beispieldaten
x = np.array([6688,16360,20136,25496,34500,36784,43368,48648,49900,50876])
     #Zugriffe
y = np.array
    ([0.719,0.754,0.789,0.823,0.878,0.899,0.928,0.952,0.964,0.970]) #Watt
    normalisiert
conc = np.array
    ([0.042,0.121,0.194,0.249,0.376,0.394,0.641,0.803,0.922,1.000]) #
    Concurrency-Faktor normalisiert
cpu = np.array([24.8,49.24,59.2,70.12,78.16,80.88,88.2,94,97.48,99.2])

print("Werte gesetzt")
############ Kubische Splines der Messungen
    #################################
yint = interpolate.UnivariateSpline(x,y , k=3)(x)
yint2 = interpolate.UnivariateSpline(x,conc, k=3)(x)
y_cpu = interpolate.UnivariateSpline(x,cpu, k=3)(x)


#Bestimme Funktionen vor der Skalierung
fwatt = np.polyfit(x,yint,6)
fconc = np.polyfit(x,yint2,6)
fcpu  = np.polyfit(x,y_cpu,6)
pwatt = np.poly1d(fwatt)
pconc = np.poly1d(fconc)
pcpu = np.poly1d(fcpu)
print("Funktionen approximiert")

############### Messungen nach Skalierung
    ####################################

##Messungen Beispieldaten
```

```
xskal = np.array
    ([7904,15368,22192,27696,34200,38760,42340,44400,51452,52584]) #
    Zugriffe
yskal = np.array
    ([0.729,0.764,0.827,0.850,0.899,0.925,0.948,0.968,0.985,1.000]) #Watt
    normalisiert
kskal = np.array
    ([0.024,0.058,0.089,0.118,0.164,0.222,0.275,0.319,0.448,0.550])  #
    Concurrency-Faktor normalisiert

####################Kubische Splines der Messungen
    ############################
yinterp = interpolate.UnivariateSpline(xskal,yskal , k=3)(x)
yinterp2 = interpolate.UnivariateSpline(xskal,kskal, k=3)(x)


################Bestimme Funktionen
fwattskal = np.polyfit(xskal,yinterp,6)
fconcskal = np.polyfit(xskal,yinterp2,6)
pwattskal = np.poly1d(fwattskal)
pconcskal = np.poly1d(fconcskal)

#################Bestimme die Deltafunktionen und dessen Maximum
deltaWatts = pwattskal - pwatt
deltaconc = pconc - pconcskal

####################Bestimme obere und unter Schranke
maximum=0
minimum=sys.maxsize

if(x[len(x)-1]>xskal[len(xskal)-1]):
    maximum=xskal[len(xskal)-1]
else:
    maximum=x[len(x)-1]

if(x[0]<xskal[0]):
    minimum=xskal[0]
else:
    minimum=x[0]

##Bestimme Minimum des Leistungsaufnahmedeltas

min=minimum
t=minimum
```

```
while(t<maximum):
    if(deltaWatts(t)<deltaWatts(min)):
        min=t
    t=t+1

##Bestimme halbiertes Maximum des Nebenlaeufigkeitsdeltas
c2=minimum
newDelta=-0.5
while(newDelta<(deltaconc(maximum)/2)):
    c2=c2+0.5
    newDelta=deltaconc(c2)


ScalingPoint=(c2+min)/2
print(ScalingPoint)
print("Scaling point CPU-Percent:"+str(pcpu(ScalingPoint)))
print("Scaling point (MAX) CPU-Percent:"+str(pcpu(c2)))
print("Scaling point (MIN) CPU-Percent:"+str(pcpu(min)))

####Zeige die Ergebnisse in Form von matplot-Charts
#plt.title("Ermitteltes Skalierungsintervall")
#plt.xlabel('CPU-Auslastung in %')
#plt.plot(x,pwatt(x),'g' ,label = 'interpolierte Leistungsaufnahme')
#plt.plot(cpu,y, 'g', label = 'Original Leistungsaufnahme')
#plt.plot(x,pconc(x) ,label = 'interpolierte Erreichbarkeit')
#plt.plot(x, y, 'go', label = 'Original Leistungsaufnahme skaliert')
#plt.plot(cpu, conc, 'b', label = 'Original Erreichbarkeit')
#plt.plot(xskal, pwattskal(xskal),'b' ,label = 'interpolierte
   Leistungsaufnahme skaliert')
#plt.plot(xskal, pconcskal(xskal), 'b', label = 'interpolierte
   Erreichbarkeit skaliert')
#plt.plot(x, deltaWatts(x), color='b', label = 'Leistungsaufnahmedelta')
#plt.plot(xskal, deltaconc(xskal),color="g", label = 'Erreichbarkeitdelta
   ')
#plt.axvline(min,color='b',linestyle='--')
#plt.plot(min,deltaWatts(min),'bo',label="Leistungsaufnahme Minumum")
#plt.axvline(c2,color='g',linestyle='--')
#plt.plot(c2,deltaconc(c2),'go',label="Erreichbarkeit Maximum/2")
#plt.axvline(ScalingPoint,color='r')
#plt.plot(pcpu(ScalingPoint),0,'ro',label="Skalierungspunkt")
#plt.axvline(pcpu(ScalingPoint),color='r')
#plt.plot(pcpu(min),0,'bo',label="Untere Schranke des
   Skalierungsintervalls")
```

```
#plt.axvline(pcpu(min),color='b',linestyle='--')
#plt.plot(pcpu(c2),0,'yo',label="Obere Schranke des Skalierungsintervalls
    ")
#plt.axvline(pcpu(c2),color='y',linestyle='--')
#plt.plot(maximum,deltaconc(maximum),'yo',label="Concurrency Maximum")
#plt.plot(c2,deltaconc(c2),'go',label="Concurrency Maximum durch 2")
#plt.legend(loc="bottom right",ncol=1,prop={'size':8})
#plt.savefig("C:\\Users\\krete\\scaler\\Skalierungsintervall.png",
    bbox_inches="tight")
#plt.show()
```

Literatur

Eigene Veröffentlichungen

Guldner, A., Kern, E., Kreten, S., and Naumann, S. (2021). *Criteria for Sustainable Software Products: Analysing Software, Informing Users and Politics ind Software Sustainability*. Springer.

Kreten, S. (Deutsche OpenStack Tage 2019). Energy Efficient Docker Usage. `https://youtu.be/kr5mqmMqXpo`. 25.11.2019.

Kreten, S. and Guldner, A. (2017). *Resource Consumption Behavior in Modern Concurrency Models in EnviroInfo 2017 - From Science to Society: The Bridge provided by Environmental Informatics*, pages 213–220. Shaker.

Kreten, S., Guldner, A., and Naumann, S. (2018). An Analysis of the Energy Consumption Behavior of Scaled, Containerized Web Apps. *Sustainability - MDPI*, 10 - 2710.

Mancebo, J., Guldner, A., Kern, E., Kessler, P., Kreten, S., Garcia, F., Calero, C., and Naumann, S. (2019). *Assessing the Sustainability of Software - A Method Comparison in Advances and New Trends in Environmental Informatics*, chapter 1, pages 1–16. Springer.

Fremdveröffentlichungen

Adolfsson, H. (2019). Comparison of auto-scaling policies using docker swarm. Master's thesis, Linköping University, Database and information techniques.

Agarwal, N. (2017). 8 Docker Use Cases. `https://medium.com/@nagarwal/docker-usecases-3b62f4d68bc4`. Letzter Aufruf: 24.01.2021.

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., and Merle, P. (2018). Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447.

Alzahrani, E. J., Tari, Z., Zeephongsekul, P., Lee, Y. C., Alsadie, D., and Zomaya, A. Y. (2016). Sla-aware resource scaling for energy efficiency. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 852–859.

Amazon (2018). Amazon AWS Service Structure - IAAS,PAAS,FAAS. `https://rubygarage.s3.amazonaws.com/uploads/article_image/file/785/iaas-paas-saas-hierarchy-diagram.jpg`. Letzter Aufruf: 24.01.2021.

Amazon (2021). AWS und Nachhaltigkeit. https://aws.amazon.com/de/about-aws/sustainability/. Letzter Aufruf: 24.01.2021.

Andrae, A. (2017). Total consumer power consumption forecast. In *Nordic Digital Business Summit*.

Andrae, A. (2019). Predictions on the way to 2030 of internet's electricity use. In *The Lost Decade? Planning the Future at Ålborg University Copenhapen, 28 February 2019.*

Asnaghi, A., Ferroni, M., and Santambrogio, M. D. (2016). Dockercap: A software-level power capping orchestrator for docker containers. In *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*, pages 90–97.

B1 Systems Gmbh (2021). Deutsche Openstack Tage. https://openstack-tage.de/. Letzter Aufruf: 24.01.2021.

Bundesministerium für Umwelt (2019). Blauer Engel - Ressourcen- und energieeffiziente Softwareprodukte. https://www.blauer-engel.de/de/get/productcategory/171. Letzter Aufruf: 24.04.2020.

Casalicchio, E. (2019). A study on performance measures for auto-scaling cpu-intensive containerized applications. *Cluster Computing*, 22(3):995–1006.

Ceph (2021). Introduction to Ceph. https://ceph.io/ceph-storage/. Letzter Aufruf: 24.01.2021.

Cisco (2018). Cisco Global Cloud Index: Forecast and Methodology, 2016-2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/global-cloud-index-gci/white-paper-c11-738085.html. Letzter Aufruf: 15.07.2019.

CNCF (2019). Cloud Native Computing Foundation . https://cncf.io. Letzter Aufruf: 15.07.2019.

CNCF (2021). CNCF Cloud Native Interactive Landscape. https://landscape.cncf.io/. Letzter Aufruf: 24.01.2021.

Congfeng Jiang, Dongyang Ou, Yumei Wang, Xindong You, Jilin Zhang, Jian Wan, Bing Luo, and Weisong Shi (2016). Energy efficiency comparison of hypervisors. In *2016 Seventh International Green and Sustainable Computing Conference (IGSC)*, pages 1–8.

Crunchbase (2021). Crunchbase Pro - Docker. https://www.crunchbase.com/organization/docker#section-overview. Letzter Aufruf: 24.01.2021.

datacenterknowledge.com (2021). Data Center Knowledge - Google Data Center FAQ. `https://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq`. Letzter Aufruf: 24.01.2021.

datadoghq.com (2018). 8 surprising facts about real Docker adoption. `https://www.datadoghq.com/docker-adoption/`. Letzter Aufruf: 24.10.2019.

Degenmann, K. (2021). JAXenter - Ist Kubernetes 2019 gefragt? `https://jaxenter.de/kubernetes-2019-trend-80573`. Letzter Aufruf: 24.01.2021.

Dirlewanger, W. (2006). *Measurement and Rating of Computer Systems Performance and of Software Efficiency - An Introduction to the ISO / IEC 14756 Method and a Guide to its Application*. Kassel University Press, Kassel, Deutschland.

Docker Inc. (2019a). Docker Documentation - About storage drivers. `https://docs.docker.com/storage/storagedriver/`. Letzter Aufruf: 15.07.2019.

Docker Inc. (2019b). Docker Documentation - Build Enhancements. `https://docs.docker.com/develop/develop-images/build_enhancements/`. Letzter Aufruf: 15.07.2019.

Docker Inc. (2019c). Docker Lifecycle. `https://medium.com/future-vision/docker-lifecycle-tutorial-and-quickstart-guide-c5fd5b987e0d`. Letzter Aufruf: 24.04.2020.

Docker Inc. (2019d). Docker official Images - golang. `https://hub.docker.com/_/golang`. Letzter Aufruf: 15.07.2019.

Docker Inc. (2019e). Docker official Images - ubuntu. `https://hub.docker.com/_/ubuntu`. Letzter Aufruf: 15.07.2019.

Docker Inc. (2019f). Install Docker Desktop for Windows. `https://docs.docker.com/docker-for-windows/install/`. Letzter Aufruf: 15.07.2019.

Docker Inc. (2021a). Our company - Docker by the numbers. `https://www.docker.com/company`. Letzter Aufruf: 13.03.2021.

Docker Inc. (2021b). Docker - Documentation. `https://docs.docker.com/`. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021c). Docker - Documentation - Use macvlan networks. `https://docs.docker.com/network/macvlan/`. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021d). Docker - Dokumentation - Configure logging drivers. https://docs.docker.com/config/containers/logging/configure/. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021e). Docker - The Industry-Leading Container Runtime. https://www.docker.com/products/container-runtime. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021f). Docker Hub - nginx. https://hub.docker.com/_/nginx. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021g). Docker Hub - Official Images. https://hub.docker.com/search?q=&type=image&image_filter=official. Letzter Aufruf: 24.01.2021.

Docker Inc. (2021h). Use the AUFS storage driver. https://docs.docker.com/storage/storagedriver/aufs-driver/. Letzter Aufruf: 24.01.2021.

Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172.

Fluentd Project (2021). Fluentd - Build Your Unified Logging Layer . https://www.fluentd.org/. Letzter Aufruf: 24.01.2021.

Öggl, B. and Kofler, M. (2018). *Docker - Das Praxisbuch für Entwickler und DevOps-Teams*. Rheinwerk Computing, Bonn, Deutschland, 1. Auflage.

Gitlab (2021). Gitlab CI/CD - AutoDevOps and Code-Pipelines. https://docs.gitlab.com/ee/topics/autodevops/. Letzter Aufruf: 24.01.2021.

Goldberg, R. P. (1973). Architecture of virtual machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, New York, NY, USA. ACM.

Google (2019a). Google Search Trends - Suchbegriff Docker . https://trends.google.de/trends/explore?date=all&q=docker. Letzter Aufruf: 15.07.2019.

Google (2019b). Google Trends für Green IT. https://trends.google.de/trends/explore?date=all&q=green%20it. Letzter Aufruf: 24.04.2020.

Gude (2021). Gude Expert Power Control 1202. https://www.gude.info/power-distribution/switched-metered-pdu/expert-power-control-1202-serie.html. Letzter Aufruf: 24.01.2021.

Guldner, A., Garling, M., Morgen, M., Naumann, S., Kern, E., and Hilty, L. M. (2018). Energy Consumption and Hardware Utilization of Standard Software: Methods and Measurements for Software

Sustainability. In Otjacques, B., Hitzelberger, P., Naumann, S., and Wohlgemuth, V., editors, *From Science to Society*, pages 251–261. Springer International Publishing.

Gutzeit, Carol und Schäfer, M. (2020). Vielfältige wolkenwelt - der weg zur cloud-nativen architektur.

Hanafy, W. A., Mohamed, A. E., and Salem, S. A. (2019). A new infrastructure elasticity control algorithm for containerized cloud. *IEEE Access*, 7:39731–39741.

Hankel, A., Heimeriks, G., and Lago, P. (2018). A systematic literature review of the factors of influence on the environmental impact of ict. In *MDPI Technologies*, volume 6.

heise Verlag, d. V. (2021). Container Conf. `https://www.containerconf.de/`. Letzter Aufruf: 24.01.2021.

Helm (2021). Helm - The package manager for Kubernetes. `https://helm.sh/`. Letzter Aufruf: 20.03.2021.

Hilty, L. M. and Aebischer, B. (2015). Ict for sustainability: An emerging research field. In Hilty, L. M. and Aebischer, B., editors, *ICT Innovations for Sustainability*, pages 3–36, Cham. Springer International Publishing.

Hintemann, R. and Hinterholzer, S. (2019). Energy consumption of data centers worldwide. In *Proceedings of Workshop at ICT for Sustainability 2019*, volume 2382.

Imdoukh, M., Ahmad, I., and Alfailakawi, M. G. (2019). Machine learning-based auto-scaling for containerized applications. *Neural Computing and Applications*.

James, A. and Schien, D. (2019). A low carbon kubernetes scheduler. In *Proceedings of the 6th International Conference on ICT for Sustainability, ICT4S 2019, Lappeenranta, Finland, June 10-14, 2019*.

Janitza (2021). UMG 604. `https://www.janitza.de/files/download/Datenblaetter/Janitza-Datenblatt-Kapitel02-UMG-604-de.pdf`. Letzter Aufruf: 24.01.2021.

Joe Dog Software (2021). Proudly serving the Internets since 1999 - Siege Home. `https://www.joedog.org/siege-home/`. Letzter Aufruf: 24.01.2021.

Johnson, P. (2021). Forbes - With The Public Clouds Of Amazon, Microsoft And Google, Big Data Is The Proverbial Big Deal. `https://t1p.de/4zbk`. Letzter Aufruf: 24.01.2021.

Kern, E., Hilty, L., Guldner, A., Filler, A., Naumann, S., and Maksimov, Y. (2018). Sustainable Software Products - Towards assesment cirteria for resource and energy efficiency. *Future Generation Computer Systems*, 86:199–210.

Kubernetes (2021a). Kubernetes Documentation - Kubernetes Basics . https://kubernetes.io/de/docs/tutorials/kubernetes-basics/explore/explore-intro/. Letzter Aufruf: 24.01.2021.

Kubernetes (2021b). Kubernetes Documentation - Understanding Pods . https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/. Letzter Aufruf: 24.01.2021.

Kubernetes (2021c). Kubernetes Documentation - Container runtimes. https://kubernetes.io/docs/setup/production-environment/container-runtimes/. Letzter Aufruf: 24.01.2021.

Kubernetes (2021d). Kubernetes Documentation - Horizontal Pod Autoscaler. https://kubernetes.io/de/docs/tasks/run-application/horizontal-pod-autoscale/. Letzter Aufruf: 24.01.2021.

Kubernetes (2021e). Kubernetes Documentation - Logging Architecture - Using a node logging agent. https://kubernetes.io/docs/concepts/cluster-administration/logging/. Letzter Aufruf: 24.01.2021.

Kubernetes (2021f). Kubernetes Repository - Cluster Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler. Letzter Aufruf: 24.01.2021.

Kubernetes (2021g). Kubernetes Repository - Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler. Letzter Aufruf: 24.01.2021.

Kurose, J. F. and Ross, K. W. (2012). *Computernetzwerke - Der Top-Down-Ansatz*. Pearson Deutschland GmbH, 5. Auflage.

Liebel, O. (2017). *Skalierbare Container-Infrastrukturen - Handbuch für Administratoren*. Rheinwerk Computing, Bonn, 1. Auflage.

Liu, B., Li, P., Lin, W., Shu, N., Li, Y., and Chang, V. (2018). A new container scheduling algorithm based on multi-objective optimization. *Soft Computing*, 22(23):7741–7752.

Lorido-Botran, T., Miguel-Alonso, J., and Lozano, J. A. (2014). A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592.

Loschwitz, Martin Gerhard (2018). Andockmanöver - Netzwerke in Docker. In *iX - Magazin für professionell Informationstechnik*.

man7.org (2021). Linux Programmer's Manual - cgroups . http://man7.org/linux/man-pages/man7/cgroups.7.html. Letzter Aufruf: 24.01.2021.

man7.org (2021a). Linux Programmer's Manual - chroot . http://man7.org/linux/man-pages/man1/chroot.1.html. Letzter Aufruf: 24.01.2021.

man7.org (2021b). Linux Programmer's Manual - namespaces . http://man7.org/linux/man-pages/man7/namespaces.7.html. Letzter Aufruf: 15.07.2019.

Mao, Y., Oak, J., Pompili, A., Beer, D., Han, T., and Hu, P. (2017). Draps: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster. In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–8.

Microsoft (2019). Then vs. now: How cloud storage has changed the way we work. https://www.microsoft.com/en-us/microsoft-365/growth-center/resources/how-cloud-storage-has-changed-the-way-we-work. Letzter Aufruf: 05.04.2020.

Miell, I. and Hobson Sayers, A. (2019). *Docker In Practice*. Manning Publications, Shelter Island, USA , 2. Auflage.

Mohammad Ali, H. M., El-Gorashi, T. E. H., Lawey, A. Q., and Elmirghani, J. M. H. (2017). Future energy efficient data centers with disaggregated servers. *Journal of Lightwave Technology*, 35(24):5361–5380.

Morabito, R. (2015). Power consumption of virtualization technologies: An empirical investigation. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 522–527.

Open Container Initiative (2016). Open Container Initiative. https://www.opencontainers.org/. Letzter Aufruf: 15.07.2019.

Open Container Initiative (2019). runC - Repository and Docs. https://github.com/opencontainers/runc. Letzter Aufruf: 15.07.2019.

Parbel, M. (2021a). Programmiersprache Go profitiert vom Cloud-Hype. https://www.heise.de/developer/meldung/Programmiersprache-Go-profitiert-vom-Cloud-Hype-3889377.html. Letzter Aufruf: 24.01.2021.

Parbel, M. (2021b). Umfrage: Web-Developer und DevOps bauen auf Go. https://www.heise.de/developer/meldung/Umfrage-Web-Developer-und-DevOps-bauen-auf-Go-4356198.html. Letzter Aufruf: 24.01.2021.

Patterson, M. K. (2008). The effect of data center temperature on energy efficiency. In *2008 11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems*, pages 1167–1174.

Penzenstadler, B., Bauer, V., Calero, C., and Franch, X. (2012). Sustainability in software engineering: A systematic literature review. In *Proceedings of the 16th International Conference on Evaluation and Assessment in Software Engineering (EASE 2012), Ciudad Real, Spain*.

Piraghaj, S. F., Dastjerdi, A. V., Calheiros, R. N., and Buyya, R. (2015). A framework and algorithm for energy efficient container consolidation in cloud data centers. In *Proceedings of the 2015 IEEE International Conference on Data Science and Data Intensive Systems (DSDIS)*, DSDIS '15, pages 368–375, Washington, DC, USA. IEEE Computer Society.

Popek, G. J. and Goldberg, R. P. (1974). Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421.

Pöschl, A. (2019). 4 Jahre, 40000 vCPUs, 400000 VMs pro Monat - OpenStack bei BMW (Deutsche OpenStack Tage. `https://youtu.be/kr5mqmMqXpo`. 25.11.2019.

Racksolutions (2021). 400 Million New Servers Might Be Needed by 2020. `https://www.racksolutions.com/news/data-center-trends/400-million-new-servers-might-be-needed-by-2020/`. Letzter Aufruf: 24.01.2021.

Rehmann, K.-T. and Folkerts, E. (2018). Performance of containerized database management systems. In *Proceedings of the Workshop on Testing Database Systems*, DBTest'18, pages 5:1–5:6, New York, NY, USA. ACM.

Santos, E. A., McLean, C., Solinas, C., and Hindle, A. (2018). How does docker affect energy consumption? evaluating workloads in and out of docker containers. *Journal of Systems and Software*, 146:14 – 25.

SimilarWeb (2021). Website Traffic Statistics and Market Intelligence. `https://www.similarweb.com/`. Letzter Aufruf: 24.01.2021.

Stack Overflow (2019). Stack Overflow Insights 2019 - Trends - Suchbegriffe Docker und Kubernetes. `https://stackoverflow.com/tags`. Letzter Aufruf: 24.01.2021.

Stack Overflow (2020). Stack Overlfow Insights 2020 - Technology. `https://insights.stackoverflow.com/survey/2020#technology-databases-all-respondents4`. Letzter Aufruf: 24.01.2021.

Statista (2011). Statistik und Prognose zur Anzahl vernetzter Geräte weltweit in den Jahren 2003 bis 2020. `https://de.statista.com/statistik/daten/studie/479023/umfrage/prognose-zur-anzahl-der-vernetzten-geraete-weltweit/`. Letzter Aufruf: 15.07.2019.

Stine, M. (2015). *Migrating to Cloud-native Application Architectures*. O'Reilly Media.

Suleman, A. (2019). 8 Proven Real-World Ways to Use Docker. https://www.airpair.com/docker/posts/8-proven-real-world-ways-to-use-docker. Letzter Aufruf: 24.01.2021.

Suárez-Albela, M., Fernández-Caramés, T., Fraga-Lamas, P., and Castedo, L. (2017). A practical evaluation of a high-security energy-efficient gateway for iot fog computing applications. *Sensors*, 19(7).

sysdig.com (2021). 2018 Docker usage report. https://sysdig.com/blog/2018-docker-usage-report/. Letzter Aufruf: 24.01.2021.

Tadesse, S. S., Malandrino, F., and Chiasserini, C. (2017). Energy consumption measurements in docker. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 2, pages 272–273.

The Linux Foundation (2019a). containerd. https://containerd.io/. Letzter Aufruf: 15.07.2019.

The Linux Foundation (2019b). containerd - Repository and Doc. https://github.com/containerd/containerd. Letzter Aufruf: 15.07.2019.

The Shift Project (2019). The Carbon Transition Think Tank. Climate Crisis: The unsustainable Use of Online Video. Paris, France.

TIOBE (2021). Programming Language Popularity. https://www.tiobe.com/tiobe-index/. Letzter Aufruf: 24.01.2021.

vSwitch, O. (2021). Production Quality, Multilayer Open Virtual Switch. https://www.openvswitch.org/. Letzter Aufruf: 24.01.2021.

Weaveworks (2021). Weave Net - Network containers across any environment. https://www.weave.works/oss/net/. Letzter Aufruf: 24.01.2021.

Wittek, K. (2019). Auf dem Prüfstand - Testen mit Docker und Testcontainers. In *iX - Magazin für professionell Informationstechnik*.