Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Konzeption und Realisierung einer Cloud-Plattform zur Konfiguration und dem Deployment von Services

Vincent Link

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr. Stefan Wagner

Betreuer/in: Dr. Asim Abduhlkhaleq, ISTE
Timur Tasci, M.Sc., ISW

Beginn am: 10. Juli 2017

Beendet am: 10. Januar 2018

Kurzfassung

Die Industrie befindet sich im Rahmen der Industrie 4.0 seit wenigen Jahren durch eine digitale Transformation im Wandel. Durch moderne Informations- und Kommunikationstechnologien werden Maschinenanlagen miteinander vernetzt, sodass intelligente Prozesse gebildet werden können. Dadurch soll nicht nur eine Automatisierung der einzelnen Teilschritte stattfinden, sondern auch zusammenhängende Prozesse modernisiert werden, um eine individuellere und effizientere Produktion zu ermöglichen. Um diese Prozesse abbilden und verwalten zu können, wird hierfür eine Plattform konzipiert.

Die Plattform soll allgemein in der Lage sein, die Daten von Werkzeugmaschinen in Services in einer Cloud zu nutzen. Über mehrere Cloud-Anbieter hinweg soll nach dem Multi-Cloud-Szenario eine anbieterunabhängige und flexible Nutzung möglich sein. Entwickler stellen verschiedenste Funktionalitäten in Services in einem Marktplatz bereit. Die Services sollen hierbei dynamisch in einer Komposition miteinander verknüpft und deployt werden können, um somit die einzelnen Schritte eines Produktionsprozesses in einer Cloud abzubilden. Services sollen aus dieser Plattform heraus auch ohne technische Kenntnisse an eine Werkzeugmaschine angebunden und allgemein zuverlässig und ausfallsicher betrieben werden. Das Konzept für die obige Plattform wird anhand einer prototypischen Implementierung validiert, um daraufhin abschließend einen Ausblick mit möglichen Erweiterungen zu geben.

Inhaltsverzeichnis

1 Einleitung **9**
1.1 Zielsetzung . . . 10

2 Grundlagen **13**
2.1 Cloud-Computing . . . 13
2.2 Container . . . 16
2.3 Messaging . . . 20
2.4 Deployment . . . 23
2.5 OPC Unified Architecture . . . 24
2.6 Software-Qualitäten . . . 26

3 Verwandte Arbeiten **27**
3.1 Service-Definition . . . 27
3.2 Container-Management . . . 29
3.3 Cloud-Anwendungs-Management . . . 31
3.4 Zuverlässigkeit & Skalierbarkeit . . . 36
3.5 Abgrenzung . . . 38

4 Konzept – Plattform **41**
4.1 Anforderungen . . . 41
4.2 Services . . . 45
4.3 Service-Kommunikation . . . 46
4.4 Back-End . . . 52
4.5 Deployment . . . 52
4.6 Benutzeroberfläche . . . 55

5 Konzept – Maschinenanbindung **59**
5.1 Anforderungen . . . 59
5.2 Generischer Maschinenservice . . . 59
5.3 Integration in Service-Plattform . . . 60
5.4 Benutzeroberfläche zur Generierung . . . 61

6 Konzept – Zuverlässigkeit & Skalierbarkeit **63**
6.1 Notwendigkeit bei Cloud-Computing . . . 63
6.2 Hosts . . . 64
6.3 Services . . . 65
6.4 Kommunikation . . . 65
6.5 Plattform . . . 66

7 Implementierung **69**
7.1 Architektur . . . 69
7.2 Services . . . 71
7.3 Kommunikation . . . 72
7.4 Deployment . . . 72
7.5 Typsicherheit . . . 72
7.6 Generischer Maschinen-Service . . . 73

8 Validierung **75**
8.1 Validierung mit Demo-Services . . . 75
8.2 Validierung an exemplarischem Fließbandaufbau . . . 77
8.3 Validierung der Maschinen-Services . . . 78

9 Zusammenfassung und Ausblick **81**
9.1 Ausblick . . . 82

Literaturverzeichnis **85**

Abbildungsverzeichnis

2.1 Cloud-Service-Modelle mit Einordnung im Application-Stack 14
2.2 Vergleich von virtuellen Maschinen mit Docker-Containern 16
2.3 Übersicht über das Ökosystem von Docker 19
2.4 Vergleich der synchronen mit der asynchronen Kommunikation 21
2.5 Aufbau einer Message-Queue 22
2.6 Kommunikationsmodelle im Vergleich 23
2.7 Aufbau einer OPC-UA-Anwendung 25
2.8 Schichtenmodell von OPC-UA 25

3.1 Aufbau eines TOSCA-Service-Templates 28
3.2 Komponenten der Rancher-Plattform 31
3.3 Architektur von SeaClouds und der verwendeten Technologien 34
3.4 Werkzeugmaschine mit Cloud-Anbindung 35
3.5 Skalierungsmöglichkeiten in der Cloud 37

4.1 Workflow mit Cloud-Services 42
4.2 Stakeholder der Service-Plattform 44
4.3 Service-Kommunikation zwischen zwei Services 47
4.4 Service-Kommunikation – Versand an gleichwertige Services 48
4.5 Service-Kommunikation – Versand an verschiedene Services 48
4.6 Service-Kommunikation – Empfang von gleichwertigen Services 49
4.7 Szenario mit mehrfach versendeten Nachrichten 49
4.8 Verteilung der Message-oriented-Middleware 53
4.9 Beispiel für Constraints bei Containern 55
4.10 Mock-Up für die Stammdatenverwaltung 56
4.11 Mock-Up für die Visualisierung der Orchestrierung 57
4.12 Mock-Up für das Monitoring 58

5.1 Verwendung eines Maschinenservices 59
5.2 Mock-Up der Maschinenservice-Benutzeroberfläche 61

6.1 Beispielrechnung nach der Knoten-basierten Verfügbarkeit 63
6.2 Aufbau eines redundanten Services 65
6.3 Consumer-Groups als logische Empfänger 66

7.1 Architektur der Service-Plattform 69
7.2 Benutzeroberfläche der Orchestrierung 70
7.3 Architektur des Back-Ends 71

8.1 Weboberfläche des Display-Services 76

8.2 Aufbau der zur Validierung verwendeten Services 76
8.3 Aufbau eines exemplarischen Workflows 78
8.4 Konvertierung generischer Werte des Maschinen-Services 79

Tabellenverzeichnis

3.1 Vergleich von Service-Definitionen 38
3.2 Vergleich von Container-Management-Plattformen 39
3.3 Vergleich von Cloud-Deployment-Tools 39
3.4 Vergleich von Industrie-Cloud-Plattformen 40

Verzeichnis der Listings

4.1 Service-Definition für eine Grenzwertüberwachung 46

5.1 Konfiguration von OPC-UA-Endpunkten 61

7.1 Schema-Definition für Apache Avro 73

1 Einleitung

Im Zeitalter der digitalen Transformation befindet sich auch die Industrie im stetigen Wandel. Maschinenanlagen werden mit moderner Informations- und Kommunikationstechnik vernetzt, um die industriellen Prozesse intelligenter zu machen. Mit Hilfe solch intelligenter Systeme sollen nicht nur einzelne Teilaufgaben, sondern auch zusammenhängende Prozesse automatisiert und modernisiert werden, um insgesamt flexibler, individueller und effizienter produzieren zu können. In sogenannten „Smart Factories“ soll die vierte industrielle Revolution eingeleitet werden. Hierfür werden jedoch Plattformen benötigt, die die modernen Produktionsprozesse in vollem Umfang abbilden und unterstützen können. Solche Plattformen müssen in der Lage sein, einzelne Produktionsschritte abzubilden und miteinander zu verknüpfen, um ein individuelles Produkt über dessen gesamten Herstellungsprozess steuern und verfolgen zu können und somit eine Wertschöpfungskette zu bilden. Zu den Aufgaben einer intelligenten Plattform gehört außer der eigentlichen Produktion auch die Überwachung der Prozesse und Maschinen, um durch Datenanalysen beispielsweise Vorhersagen über die Produktqualität oder den Zustand der Maschinen erhalten zu können. Die notwendigen Daten sollen hierbei aus den unzähligen Datenquellen von Sensoren, Aktoren und Steuerungen in Werkzeugmaschinen gesammelt werden und zur Optimierung der Prozesse beitragen[1].

Im Forschungsprojekt *MultiCloud*[2], gefördert vom Bundesministeriums für Bildung und Forschung, soll aus diesen Daten ein Mehrwert für die Industrie gebildet werden. Im Rahmen des Projekts werden sowohl in Forschungsarbeiten als auch in Studienarbeiten und -projekten Methoden und Technologien entwickelt, um solche Daten zu erfassen und serviceorientiert zu verarbeiten. Solche Services sollen über standardisierte Schnittstellen Daten abrufen und auch an andere Services weitergeben können. Um das volle Potential einer serviceorientierten Architektur nutzen zu können, sollen die Services in einer Cloud-Plattform betrieben werden, die eine annähernd Ressourcen-unbeschränkte Nutzung ermöglicht.

Hierfür wird eine Plattform benötigt, die an den Datenquellen ansetzt und bei Services in einer Cloud-Plattform endet. Verschiedene Datenquellen und Schnittstellenformate müssen in solchen Services einfach genutzt werden können. Die Services selbst sollen wiederum auch einfach entwickelt und leicht in Produktionsumgebungen integriert werden können, sodass beispielsweise mit Sharing-Lösungen Services in mehreren Umgebungen nutzbar sind. Der Betrieb soll hierbei die Ansprüche an Zuverlässigkeit, Datensicherheit und Ausfallsicherheit bisheriger Produktionsprozesse erfüllen, jedoch gleichzeitig dem Anwender der Plattform die dahinter liegende Komplexität abnehmen.

[1] Dossier: Digitale Transformation in der Industrie, https://www.bmwi.de/Redaktion/DE/Dossier/industrie-40.html
[2] BMBF MultiCloud, https://www.bmbf-multicloud.de/

1.1 Zielsetzung

In dieser Masterarbeit soll als Teil des Forschungsprojekts MultiCloud eine Plattform konzipiert werden, die an unterster Ebene bei den Maschinen ansetzt und beim Betrieb von Services endet, sodass Maschinen gesteuert und deren Daten genutzt werden können. Es sollen Konzepte für wiederverwendbare und flexible Services, sichere Kommunikation zwischen diesen Services und eine für unerfahrene Nutzer konzipierte Plattform beschrieben werden, welche allerdings auch vorhandene Normen und Standards berücksichtigt, insofern diese geeignet sind. Ziel soll es sein, die zahlreichen Sensoren durch einfach auffindbare Services nutzen zu können und dem Anwender einen einfachen Einstieg zu geben, um eine Wertschöpfungskette zu optimieren.

Die Vision ist, dass ein Anwender die Plattform über eine Nutzeroberfläche ansehen und steuern kann. In dieser Oberfläche sollen in einer Art zentralem Marktplatz verschiedenste Services veröffentlicht und abgerufen werden können, sodass auch andere Nutzer einen Zugriff darauf haben. Service-Entwickler können Services mit beliebigen Technologien entwickeln, um eine geringe Einstiegshürde zu schaffen und eventuell auch das vollste Potential durch Verwenden von Nischentechnologien nutzen zu können. Services müssen allerdings gegen einen Standard entwickelt sein, um wiederverwendbar und mit anderen Services kombinierbar zu bleiben. Plattformnutzer können Services aus dem Marktplatz den Schnittstellen entsprechend miteinander verknüpfen und somit einen Prozess oder ein komplexes Szenario in einer Orchestrierung abbilden. Solch eine Orchestrierung kann daraufhin im eigenen Rechenzentrum und auch auf Servern von Cloud-Providern betrieben werden, während der Nutzer über Statusaktualisierungen und eine Übersicht auf dem Laufenden bleiben kann. Werden von einem Service mehr Ressourcen benötigt oder möchte der Plattformnutzer den Durchsatz erhöhen, soll die Leistung automatisch oder manuell skaliert werden können, während eventuelle Anforderungen an die Ausfallsicherheit erfüllt bleiben.

Aufgrund der hohen Komplexität wird diese Arbeit in enger Zusammenarbeit mit einer parallel laufenden, ergänzenden Masterarbeit durchgeführt. Der Fokus dieser Arbeit liegt deshalb hauptsächlich bei den folgenden Punkten:

- Benutzbarkeit der Plattform
- Zuverlässige Kommunikation von Services
- Deployment und Betrieb der Services auf verschiedenen Zielplattformen
- Einfache Anbindung an Maschinen
- Zuverlässigkeit und Skalierbarkeit der Plattform-Komponenten

Gliederung

Die Arbeit ist hierbei in folgender Weise gegliedert:

Kapitel 2 – Grundlagen leitet den Leser in die Grundlagen dieser Masterarbeit ein.

Kapitel 3 – Verwandte Arbeiten gibt einen Überblick über den aktuellen Stand der Wissenschaft und Technik, um mit einem Vergleich die Notwendigkeit dieser Arbeit zu zeigen.

Kapitel 4 – Konzept – Plattform beschreibt das Konzept einer Plattform, die die zentralen Teile der obigen Zielsetzung erfüllt. Hierbei werden die grundlegenden Anforderungen im Detail erörtert.

Kapitel 5 – Konzept – Maschinenanbindung erweitert das Plattform-Konzept um eine einfache Möglichkeit, mit der Datenquellen von Maschinen genutzt werden können.

Kapitel 6 – Konzept – Zuverlässigkeit & Skalierbarkeit erweitert das Plattform-Konzept um Ansätze, mit denen eine höhere Zuverlässigkeit und allgemein eine Skalierbarkeit der Komponenten erreicht werden kann.

Kapitel 7 – Implementierung gibt einen Überblick über die Umsetzung der Konzepte.

Kapitel 8 – Validierung prüft die zuvor beschriebenen Konzepte anhand der Implementierung mit verschiedenen Szenarien.

Kapitel 9 – Zusammenfassung und Ausblick bereitet die Ergebnisse der Arbeit noch einmal auf und stellt mögliche Erweiterungen vor.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Konzepte und Technologien erläutert, auf denen die darauffolgenden Kapitel basieren. Hierbei wird zuerst allgemein auf die Themen *Cloud-Computing* und *Containerisierung von Anwendungen* eingegangen, um mit spezielleren Themen wie *OPC Unified Architecture* für die Kommunikation mit Industriemaschinen abzuschließen.

2.1 Cloud-Computing

Neben dem starken Bezug zur Industrie in dieser Arbeit ist das Themengebiet *Cloud-Computing* auch von großer Relevanz. Cloud-Computing beschreibt nicht nur Berechnungen in Rechnernetzen, wie der Name zuerst vermuten lässt, sondern umfasst, wie Fehling et al. im Buch *Cloud Computing Patterns* [FLR+14] darlegen, deutlich mehr. Cloud-Computing beschreibt allgemein das Angebot und die Nutzung von Ressourcen, die in einer Cloud — also einem Rechnernetzwerk — bereitgestellt werden. Diese Ressourcen umfassen reine Rechenleistung für Berechnungen mit Computer- oder grafischen Prozessoren, aber auch Datenspeicher oder Netzwerktechnologien für die Kommunikation zwischen Anwendungen.

Eine Cloud-Plattform, die solche Ressourcen anbietet, muss hierbei gewisse Anforderungen erfüllen. So müssen die Cloud-Ressourcen dem Nutzer auf Abruf und über ein Netzwerk mit hoher Bandbreite und Verfügbarkeit zur Verfügung stehen. Um flexibel je nach Anforderung Ressourcen nutzen oder freigeben zu können, muss die Cloud-Plattform diese in Pools bereithalten und schnell freigeben oder wieder aufnehmen können (=elastisch sein) [MG11].

2.1.1 Eigenschaften von Cloud-Anwendungen

Eine Anwendung, die das Potential solch einer Cloud-Plattform im besten Umfang nutzen können soll, muss ebenfalls bestimmte Anforderungen erfüllen — die *IDEAL*-Eigenschaften [FLR+14].

Isolated State Die Anwendung sollte hierbei zustandslos sein, sodass möglichst keine Sitzungen von Nutzern oder im laufenden Betrieb genutzte Daten behalten werden. Ist eine Anwendung zustandsbehaftet, so beeinflusst das die Skalierbarkeit enorm, wenn bei jeder Operation der Zustand mit anderen skalierten Instanzen synchronisiert werden muss. Deshalb empfiehlt es sich, den Zustand isoliert zentral zu halten — wie beispielsweise in einer Datenbank.

Distribution Die Ressourcen einer Cloud-Plattform sind üblicherweise verteilt und in großem Umfang vorhanden. Die Anwendung bzw. die Komponenten einer Anwendung sind auf diesen Ressourcen teilweise sogar in mehreren Rechenzentren auch global verteilt und müssen mit höheren Kommunikationszeiten oder Unterbrechungen bei Netzwerkverbindungen umgehen können.

Elasticity Die Mächtigkeit einer Cloud-Plattform ergibt sich üblicherweise dadurch, dass eine große Menge gleicher Ressourcen vorhanden ist, die sonst nicht an einem Ort zur Verfügung steht. Da diese Ressourcen homogen sind, ist normalerweise nur eine horizontale Skalierung der Anzahl möglich, mit welcher die Anwendung dynamisch umgehen können muss.

Automated Management Die Verfügbarkeit einzelner Cloud-Ressourcen ist üblicherweise nicht garantiert, dafür allerdings die Verfügbarkeit von Ressourcen des gleichen Typs. Durch Überwachen der Auslastung oder Antwortzeiten sollte die Anwendung selbst in der Lage sein, solche Ausfälle einzelner Ressourcen zu erkennen und neue selbst zu provisionieren.

Loose Coupling Eine niedrige Kopplung der Komponenten einer Anwendung erleichtert die Nutzung mehrerer Ressourcen. Durch wenige Abhängigkeiten der Komponenten untereinander und entkoppelte Kommunikationswege (vgl. Abschnitt 2.3) können Auswirkungen von Ausfällen reduziert und eine bessere Skalierbarkeit erreicht werden.

2.1.2 Service-Modelle

Cloud-Plattformen können ihre Dienste auf unterschiedlichen Ebenen anbieten, sodass unterschiedliche Arten von Anwendungen und Nutzern darauf aufbauen können. Die Aufteilung kann hierbei über den Anwendungs-Stack, wie in Abbildung 2.1 dargestellt, stattfinden.

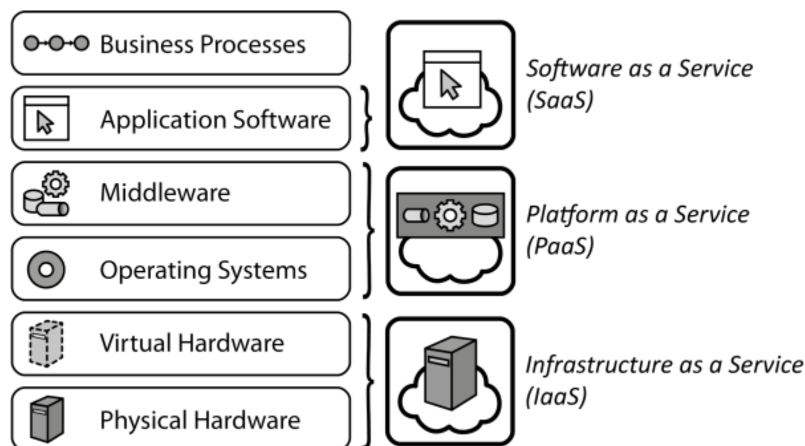


Abbildung 2.1: Die verschiedenen Cloud-Service-Modelle werden auf die jeweils bereitgestellten Ressourcen des Application-Stacks abgebildet (Abbildung angepasst) [FLR+14].

Infrastructure-as-a-Service Auf unterster Ebene gibt der Cloud-Anbieter direkten Zugriff auf echte oder häufiger virtualisierte Hardware. Die Nutzer können somit mit größtmöglicher Flexibilität diese Hardware nutzen und flexibel provisionieren, ohne selbst elementare Verwaltungsaufgaben für die Infrastruktur übernehmen zu müssen. Der Anbieter organisiert die Stromversorgung, den Standort und die Anbindung und leitet die Kosten als Service an die Nutzer weiter. IaaS-Plattformen sind beispielsweise auch die *Elastic Compute Cloud (EC2)*[1] von Amazon oder das Open-Source-Projekt *OpenStack*[2].

[1] Amazon EC2, https://aws.amazon.com/de/ec2/
[2] OpenStack, https://www/openstack.org/

Plattform-as-a-Service Als Abstraktion über dem IaaS-Modell werden beim Platform-as-a-Service-Modell bereits das Betriebssystem und die Laufzeitumgebung für eine Anwendung als Middleware bereitgestellt. Die Nutzer dieser Plattformen benötigen somit kein tiefgehendes Wissen über den Betrieb der Hardware oder die Einrichtung der Middleware, sondern können sich auf den Betrieb konzentrieren, wie es beispielsweise von *Heroku*[3] angeboten wird.

Software-as-a-Service Auf oberster Ebene wird direkt Software als Service angeboten, sodass Anwender diese nur in ihre Geschäftsprozesse integrieren müssen. Die Entwicklung und der Betrieb der Software werden hierbei vom Anbieter übernommen. Bekannte Beispiele sind *Microsoft Office Online*[4] oder CRM-Systeme von *Salesforce*.

2.1.3 Deployment-Modelle

Unabhängig vom Service-Modell können Clouds im Betrieb hinsichtlich des Anbieters und dem Umfang der Nutzerzugriffe unterschieden werden [MG11].

Public-Cloud Eine Public-Cloud bietet ihre Ressourcen öffentlich für eine große Anzahl an Nutzern an. Die große Anzahl womöglich sogar globaler Nutzer sorgt für eine gleichmäßige Nutzung der Ressourcen, sodass sich Anforderungen einzelner weniger stark auf die Gesamtlast auswirken. Durch Virtualisierungstechnologien werden die Nutzerräume sicherheitstechnisch voneinander getrennt, wie es beispielsweise auch bei Angeboten von Amazon oder Google der Fall ist.

Private-Cloud Im Gegenteil zur Public-Cloud werden die Ressourcen einer Private-Cloud nicht öffentlich angeboten und häufig nur von einer relativ kleinen Nutzergruppe verwendet. Starke Schwankungen der Auslastung und höhere Kosten werden trotzdem in Kauf genommen, da die zusätzlich gewonnene Sicherheit der Private-Cloud häufig von Bedeutung und in öffentlichen virtualisierten Umgebungen nicht unbedingt gegeben ist[5].

Community-Cloud Eine Community-Cloud beschreibt eine Mischung aus einer Public- und einer Private-Cloud. Sie steht nur einer kleinen Gruppe von Nutzern zur Verfügung, die sich gegenseitig vertrauen. Somit können Anforderungen an die Sicherheit, den Datenschutz und die Privatsphäre eher gewährleistet werden, während die Vorteile einer Cloud erhalten bleiben.

Hybrid-Cloud Wenn Clouds mit unterschiedlichen Deployment-Modellen zusammen genutzt werden müssen, können diese gemeinsam in einer Hybrid-Cloud integriert werden. Diese Anforderung ergibt sich, wenn beispielsweise firmeninterne Clouds mit öffentlichen Clouds kombiniert werden sollen. Die Clouds können über eine einheitliche Middleware zur Verfügung gestellt werden, während sichere Kommunikationskanäle und Authentifizierung im Hintergrund organisiert werden.

[3] Heroku, https://www.heroku.com/
[4] Microsoft Office Online, https://www.office.com/
[5] Sicherheitslücken in virtuellen Maschinen, https://www.heise.de/newsticker/meldung/Hacker-brechen-aus-virtueller-Maschine-aus-3658416.html oder https://www.heise.de/security/meldung/Ausbruch-aus-VM-moeglich-VMware-schliesst-kritische-Sicherheitsluecken-3894067.html

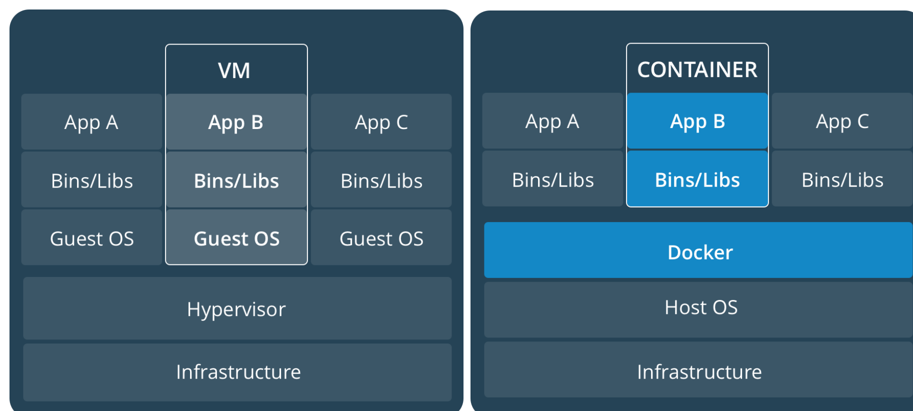
2.2 Container

Elastische Clouds wurden bis vor einigen Jahren hauptsächlich durch Virtualisierung mit virtuellen Maschinen (VMs) erreicht. Die Motivation hierbei ist, Prozesse als kontrollierbare und abgegrenzte Einheiten isoliert voneinander betreiben zu können, während die System-Ressourcen durch Sicherheitsrichtlinien aufteilbar sind. So soll es beispielsweise möglich sein, mehrere Anwendungen, die verschiedene Versionen einer Bibliothek benötigen, gleichzeitig auf einem Server zu betreiben. Virtuelle Maschinen ermöglichen die Virtualisierung eines gesamten Betriebssystems, sodass einzelne physische Server mehrere Betriebssysteme und dadurch effektiv mehrere getrennte Umgebungen bereitstellen können. Eine allerdings immer populärer werdende Virtualisierungstechnik sind sogenannte *Container*.

Virtuelle Maschinen und Container setzen auf unterschiedlichen Cloud-Service-Modellen auf: Während erstere auf einer Infrastructure-as-a-Service-Plattform aufbauen können, erwarten Container bereits eine bestimmte Container-Laufzeitumgebung, wie sie beim Platform-as-a-Service-Modell gegeben ist. Die Grundideen des Container-Konzepts sind eine leichtgewichtige Laufzeitumgebung, das Nutzen einer großer Anzahl von Servern und die Möglichkeit, Container miteinander zu verbinden [Pah15].

2.2.1 Funktionsweise

Virtuelle Maschinen bringen zahlreiche Einschränkungen mit sich, die sich mit den Grundkonzepten der Container kreuzen. Der Hypervisor eines Servers (vgl. Abbildung 2.2) stellt die grundlegend notwendige Funktionalität zur Virtualisierung bereit. Auf dieser Basis können mehrere virtuelle Maschinen aufbauen. Die Speichernutzung einer VM ist vergleichsweise groß, da sie ein gesamtes Betriebssystem zusätzlich zur Anwendung und den notwendigen Programmbibliotheken enthalten muss, wodurch einfache Operationen bereits deutlich mehr Zeit benötigen.



(a) Vergleich von virtuellen Maschinen … **(b)** … mit Docker-Containern.

Abbildung 2.2: Der Application-Stack mehrerer Anwendungen auf einem Server, links mit virtuellen Maschinen, rechts mit Containern, basierend auf der Docker-Container-Implementierung. [Doc17]

Um Container auf einem Server zu betreiben, wird kein Hypervisor benötigt, sondern ein Betriebssystem (meistens Linux), das von der Container-Laufzeitumgebung unterstützt wird. Einzelne Container können wie im abgebildeten Beispiel der Docker-Container-Implementierung auf dieser Umgebung aufsetzen. Die Container enthalten nur die Anwendung selbst und die hierfür benötigten Programmbibliotheken. Eine in einen Container gekapselte Anwendung benötigt deshalb prinzipiell nur den minimal notwendigen Speicher, im Gegensatz zu einer virtuellen Maschine, welche zusätzlich das gesamte Betriebssystem enthält [Mer14; Pah15].

Der Linux-Kernel bietet bereits einige Konzepte, die die Virtualisierung von Ressourcen ermöglichen. Zu diesen Ressourcen gehören die Schnittstellen von LXC (LinuX Containers), cgroups und ein Copy-on-Write-Dateisystem.

Namespaces

Mit Hilfe von Linux Namespaces[6] können System-Ressourcen für mehrere Prozesse virtualisiert und von ihnen isoliert werden. So lassen sich mit den Namespaces Prozess-Ids für Prozesse, Netzwerkgeräte zur Kommunikation, Mount-Points für den Datenspeicher, Nutzer und Nutzergruppen oder auch der verwendete Linux-Kernel virtualisieren und damit vom eigentlichen Host-System entkoppeln. Änderungen an den globalen Ressourcen sind nur für Prozesse innerhalb von einem Namespace sichtbar und von anderen Namespaces isoliert [Mer14].

Control Groups (cgroups)

Ergänzend zu den mit Namespaces isolierten Ressourcen können mit den sogenannten cgroups die Zugriffe und der Umfang der genutzten Ressourcen limitiert werden. So lassen sich Arbeitsspeicher oder die Festplattengröße beschränken, während durch die von cgroups bereitgestellten Metriken noch einmal feiner auf Prozessebene der Ressourcenverbauch kontrolliert werden kann [Mer14; Pah15].

Copy-on-Write-Dateisystem

Das geschichtete Copy-on-Write-Dateisystem ermöglicht eine speicherarme und flexible Verwaltung der Daten beim Einsatz von Containern. Durch Aufteilen des Dateisystems von Containern in Schichten lassen sich einzelne Schichten und somit Container wiederverwenden. Das Dateisystem einer einfachen Anwendung kann so aufgebaut sein, dass die unterste schreibgeschützte Ebene aus einem einfachen Linux besteht, welches auch von anderen Containern genutzt werden kann und somit den Speicherbedarf auf dem Server reduziert. Die Anwendung selbst liegt nun in einer eigenen, disjunkten Schicht darüber, wobei nur geänderte Dateien aus der darunterliegenden Linux-Schicht enthalten sind [Mer14; Pah15].

[6] Linux Namespaces, http://man7.org/linux/man-pages/man7/namespaces.7.html

2.2.2 Implementierungen

Containerisierung unter Linux-Systemen ist bei weitem nichts Neues, wie an den seit Jahren existierenden Linux-Kernel-eigenen Werkzeugen hierfür erkennbar ist. Erst durch den wachsenden Trend und einen einfachen Einstieg — vor allem ermöglicht durch Docker — rücken weitere Tools in den Fokus.

Docker

Die wohl bekannteste und inzwischen am weitesten verbreitete Implementierung des Container-Konzepts ist Docker[7]. Hinter dem Begriff *Docker* verbirgt sich nicht nur die reine Implementierung, sondern die dahinterstehende Firma *Docker Inc.* und gewissermaßen eine Produktkette rund um die Implementierung. Die von Docker Inc. entwickelte Open-Source-Implementierung wurde Anfang 2017 in das Projekt *Moby*[8] zur Wiederverwendbarkeit und Abgrenzung vom Produkt *Docker* umbenannt, auf welchem Docker nun basiert. Die hinter Docker stehende Firma stellt das Produkt in zwei Ausführungen bereit: Eine freie *Community Edition* und eine kommerzielle *Enterprise Edition*. Zusätzlich zur Implementierung der Container-Laufzeitumgebung stellt die Enterprise-Edition weitere Produkte und Support rund um Docker bereit.

Docker funktioniert prinzipiell wie andere Container-Implementierungen wie in Abbildung 2.2 dargestellt. Mit Hilfe der Linux-Kernel-Werkzeuge *cgroups* und *Namespaces* und einem *Copy-on-Write-Dateisystem* realisiert Docker seine eigene Container-Implementierung. Der eigentliche Durchbruch der Container gelang Docker aber nicht durch die reine Implementierung, sondern wegen der Werkzeuge um diese herum. Obwohl Docker Linux-Werkzeuge nutzt, kann es auch von Nutzern auf *Microsoft Windows* und *macOS* genutzt werden. So wird beispielsweise mit virtuellen Maschinen mit Linux-Systemen und einer direkten Anbindung der Docker-Steuerungswerkzeuge eine gleichwertige Container-Schnittstelle wie direkt unter Linux-Systemen geboten. Durch die breite Verfügbarkeit auf allen üblichen Betriebssystemen konnte Docker großen Anklang in einer breit aufgestellten Community finden [Mer14].

Docker Image Docker-Container können aus sogenannten Docker-Images instanziiert werden, die ein Abbild eines Containers sind und dem Abbild einer virtuellen Maschine ähneln. Alle Abbilder einer containerisierten Anwendung werden in einem eindeutig benannten Docker-Repository gruppiert, das alle Versionen der Images dieser Anwendung enthält. Sollen neue Container einer Anwendung gestartet werden, wird Docker angewiesen, ein Image einer bestimmten Version aus dem Repository der Anwendung zu verwenden. Ein Docker-Image besteht aus einer oder mehreren übereinanderliegenden Schichten eines Copy-on-Write-Dateisystems und wird über eine Image-Id eindeutig identifiziert. Um ein neues Image zu erstellen, wird zuerst in einem *Dockerfile* ein existierendes Image referenziert, auf welchem das neue aufbauen soll (bspw. die Linux-Distribution *Ubuntu*). Anschließend können mit verschiedenen Befehlen die im Basis-Image vorhandenen Anwendungen genutzt (bspw. um Java zu installieren), Anwendungsdateien zum Image hinzugefügt (bspw. die Java-Anwendung) und die Startparameter für die Anwendung festgelegt werden, die später beim Start eines Containers aufgerufen werden [Mer14].

[7] Docker, https://docker.com
[8] Moby, https://github.com/moby/moby

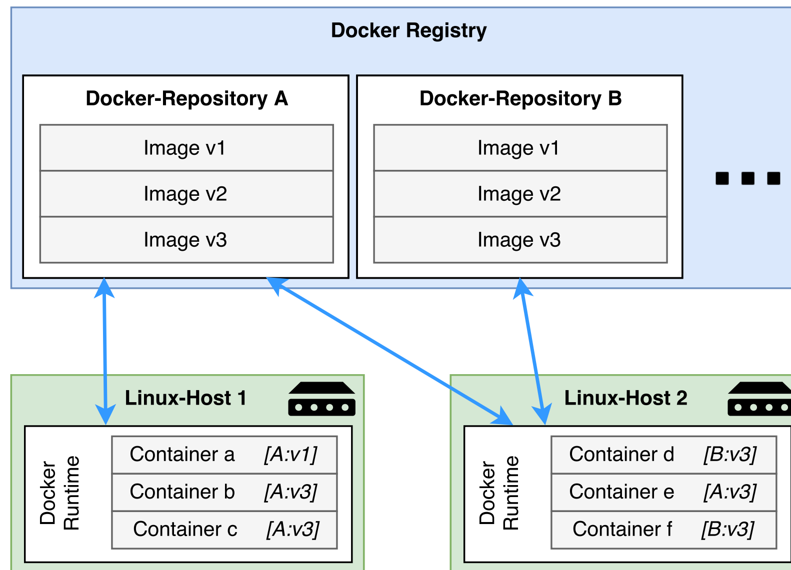


Abbildung 2.3: Eine Übersicht über das Zusammenspiel von Docker-Images, Containern, einer Registry und der Runtime.

Docker Registry Eine *Docker Registry* ist ein freies Produkt von Docker, das Docker-Images zentral versioniert bereitstellen kann. In einer Docker-Registry können verschiedene Nutzer mehrere Docker-Repositories mitsamt den zugehörigen Images verwalten und Images dorthin senden oder abrufen. Diese Abbilder können entweder direkt gestartet oder als Basis für ein darauf basierendes, neues Image verwendet werden. Die standardmäßig konfigurierte Registry bei Docker ist der öffentliche Docker Hub[9], welcher von Docker Inc. betrieben wird. Diese öffentliche Registry ermöglichte eine rasante Verbreitung von Docker durch einfaches Wiederverwenden von Container-Abbildern anderen Nutzern einer globalen Community.

Docker, Docker-Compose, Docker-Machine Zur Container-Implementierung gehört das Kommandozeilentool `docker`. Mit `docker` kann Docker vollständig konfiguriert und gesteuert werden. So lassen sich Container starten, Abbilder verwalten und diese zu Docker Registries schicken bzw. aus ihnen herunterladen. Weiterhin lassen sich auf tieferer Ebene Statistiken darstellen oder virtuelle Netzwerke für die Kommunikation zwischen Containern steuern. Mit `docker-compose` können einzelne Container und Kompositionen als zusammenhängende Applikation verknüpft und konfiguriert werden, sodass Docker gewährleistet, dass der Verbund wie angegeben gestartet wird. `docker-machine` ermöglicht die Verwaltung von virtuellen Maschinen, wie es auf Linux-fremden Betriebssystemen notwendig ist, und sorgt für die Integration `docker` und den virtualisierten Umgebungen.

[9] Docker Hub, https://hub.docker.com

rkt

rkt[10] von CoreOS bietet wie Docker eine Container-Laufzeitumgebung, die allerdings einen größeren Fokus auf sicherheitstechnische Aspekte und offene Standards, wie bspw. die *App Container Specification* `appc` legt. rkt gleicht Docker in vielen Aspekten, unterscheidet sich jedoch vor allem auch im grundlegenden Ansatz bei der Integration in Linux. Docker nutzt einen sogenannten Dämon-Prozess zum Steuern der einzelnen Container, welcher sich allerdings schlecht mit dem init-System kombinieren lässt, das für den Start der Anwendungen und des Systems bei Linux verantwortlich ist. rkt funktioniert an dieser Stelle den Unix-Konventionen entsprechend und setzt außerdem einen größeren Fokus auf Sicherheit und einen modularen Aufbau [Cor].

LXC & LXD

Im Gegensatz zu den beiden vorigen Implementierungen ist LXC eine System-Container-Laufzeitumgebung, die selbst auch Container aus LXC-spezifischen Repositories herunterladen kann. LXC ist allerdings nicht dafür geeignet, containerisierte Anwendungen zu starten und ähnelt damit eher einer Laufzeitumgebung für virtuelle Maschinen. LXD hingegen ist eine REST-Schnittstelle, welche die Container-Runtime über einen entkoppelten Prozess steuert und somit mehr Ausfallsicherheit bietet [Cor; Mer14].

Weitere

Abgesehen von den zuvor genannten Implementierungen existieren auch noch weitere, spezifischere wie *Tupperware* [11] von Facebook, welche allerdings aufgrund ihres Nischendaseins nicht in dieser Arbeit berücksichtigt werden.

2.3 Messaging

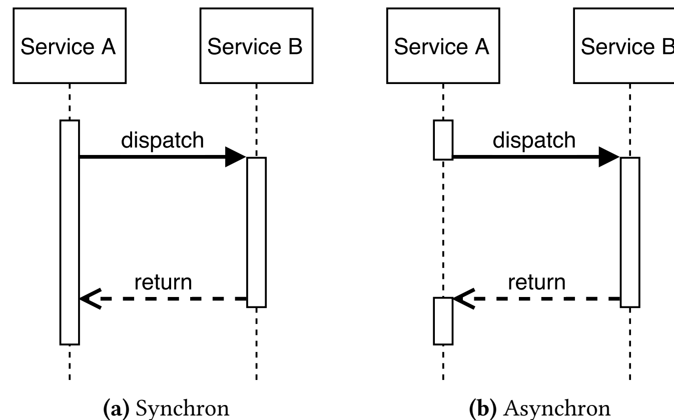
Bei verteilt arbeitenden Anwendungen, wie es vor allem in einem containerisierten Szenario in einer Cloud der Fall ist, ist ein Nachrichtenaustausch über das Dateisystem oder Remote-Procedure-Calls nicht mehr praktikabel [Cur04]. Anforderungen an die Skalierbarkeit, Zuverlässigkeit oder den Durchsatz, welche allesamt eine hohe Kopplung bei oben genannten Kommunikationstechniken implizieren, können hierbei nicht mehr erfüllt werden. Mit einer Message-oriented-Middleware (MOM) als Zwischenschicht zur Entkopplung der Kommunikation und als zentralem Baustein in verteilten Systemen, können verschiedene Anwendungen Nachrichten untereinander austauschen, während die MOM die Einhaltung etwaiger Anforderungen gewährleistet [Cur04].

[10] CoreOS rkt, https://coreos.com/rkt/

[11] Facebook Tupperware, https://www.golem.de/news/statt-docker-und-kubernetes-facebook-braucht-tupperware-fuer-seine-container-1710-130782.html

2.3.1 Interaktionsmodelle

Messaging ermöglicht allgemein die Kommunikation zwischen mehreren Anwendungen auf unterschiedliche Art und Weise: synchron und asynchron.



(a) Synchron **(b)** Asynchron

Abbildung 2.4: Synchron kommunizierende Services blockieren, bis eine Antwort eintrifft, während ein asynchron kommunizierender Service seine Arbeit fortsetzt und durch einen Callback auf die Antwort reagiert.

Beim synchronem Messaging blockiert und wartet der Aufrufer so lange, bis eine Antwort vom Aufgerufenen zurückkommt. Hierbei wird die Kontrolle über die Fortsetzung der Programmlogik in die Hand des Aufgerufenen gegeben, da dieser bestimmen kann, wann er eine Antwort sendet und somit den Aufrufer fortfahren lässt. Wartet der Aufrufer auf mehrere Antworten, verstärkt sich dieser Effekt.

Im Gegensatz dazu behalten alle Beteiligten beim asynchronen Interaktionsmodell die Kontrolle über den Ablauf des eigenen Prozesses, da nicht auf eine Antwort gewartet werden muss. Wird nun allerdings direkt nach dem asynchronen Aufruf mit der eigenen Prozesslogik fortgefahren, kann nicht angenommen werden, dass die aufgerufene Logik bereits ausgeführt worden ist. Das asynchrone Modell erfordert eine Message-oriented-Middleware, die den eigentlichen Nachrichtenaustausch übernimmt und bei Eintreffen von Antworten je nach Anforderung dies dem Aufrufer signalisiert [Cur04].

2.3.2 Message-oriented-Middleware

Eine Message-oriented-Middleware bildet eine Zwischenschicht zwischen allen Kommunikationspartnern, die die Vorteile der asynchronen Kommunikation in vollem Umfang nutzen kann. Die Kommunikation findet hierbei nach einem Service-orientierten Ansatz statt.

Kopplung Die Kopplung zwischen den einzelnen Kommunikationsendpunkten wird stark reduziert, da alle ausschließlich mit der MOM kommunizieren. Änderungen an Schnittstellen der Kommunikationspartner verhindern nicht automatisch die Möglichkeit zu kommunizieren, da in der Middleware beispielsweise Umwandlungen durchgeführt werden können.

Zuverlässigkeit Eine Message-oriented-Middleware kann durch den sogenannten *store-and-forward*-Mechanismus einen zuverlässigen Nachrichtenaustausch, je nach Anforderung an den Grad der Zuverlässigkeit, gewährleisten. Dieser Grad kann beispielsweise angeben, dass eine Nachricht mindestens von einem Empfänger exakt einmal empfangen werden soll.

Skalierbarkeit Die Entkopplung der Kommunikationspartner durch die Zwischenschicht ermöglicht ebenfalls eine Entkopplung der einzelnen Teilsysteme und lässt diese unabhängig voneinander skalieren. Sie kann unerwartet große Mengen an Nachrichten auffangen und an die skalierten Empfängersysteme weitergeben, ohne diese zu überlasten, wenn auf einen Schlag alle Nachrichten auf einmal übergeben würden.

Verfügbarkeit Im Gegensatz zur direkten Kommunikation ohne eine MOM wird nicht mehr erwartet, dass alle Kommunikationspartner gleichzeitig verfügbar sind. Ausfälle einzelner Teilsysteme der Anwendung sowie der Kommunikationszwischenschicht werden erwartet und durch die Entkopplung des Gesamtsystems ohne einen Totalausfall überwunden.

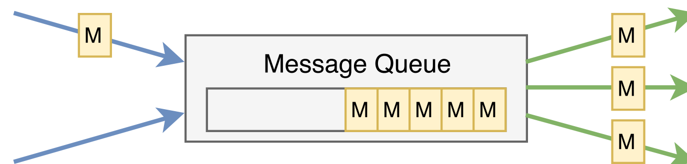


Abbildung 2.5: Schematische Darstellung einer Message-Queue (angelehnt an [Cur04]).

Message-Queues sind hierbei als Speicher der Nachrichten in einer Message-oriented-Middleware ein zentraler Bestandteil. Mehrere Sender (Producer) können Nachrichten wie in Abbildung 2.5 in eine Message-Queue senden, welche mehrere Nachrichten speichern kann und diese an mehrere Empfänger (Consumer) weitergeben kann. Nachrichten können vor der Weitergabe an die Empfänger sortiert und gefiltert werden, sodass beispielsweise nach der Sendezeit sortiert wird und veraltete Nachrichten gelöscht werden. Durch die Message-Queue als Puffer wird außerdem eine Verfügbarkeit bei fehlenden bzw. ausgefallenen Empfängern oder Nachrichtenüberläufen ermöglicht [Cur04].

2.3.3 Kommunikationsmodelle

Die zwei weitest verbreiteten Modelle, um Nachrichten einzelnen oder mehreren Empfängern zugänglich zu machen, sind das Point-to-Point- und das Publish-Subscribe-Modell.

Point-to-Point Mit dem Point-to-Point-Modell werden Nachrichten von mehreren Sendern an genau einen Empfänger über einen Kanal gesendet. Prinzipiell können zwar mehrere Empfänger Nachrichten empfangen, allerdings kann eine Nachricht nicht an mehrere Empfänger zugestellt werden, sondern nur an einen einzigen. Die Empfänger sind untereinander gleichgestellt: Nachrichten können deshalb generell an jeden verfügbaren Empfänger zugestellt werden. Sollen Nachrichten an unterschiedliche Empfänger gesendet werden, sind mehrere Point-to-Point-Kanäle notwendig [Cur04].

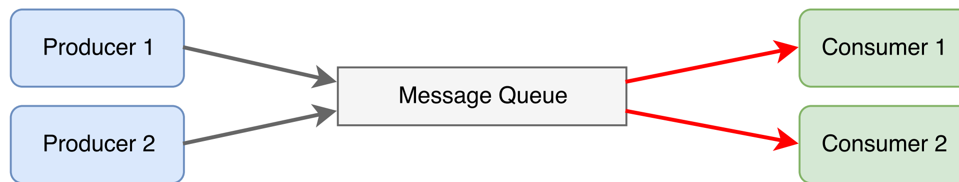


Abbildung 2.6: Die Kommunikationsmodelle unterscheiden sich hauptsächlich in der Anzahl der Empfänger (rote Pfeile). Beim Point-to-Point-Modell erhält nur ein Empfänger die Nachricht, bei Publish-Subscribe alle Empfänger.

Publish-Subscribe Damit Nachrichten in einem entkoppelten System von sich gegenseitig unbekannten Sendern und Empfängern trotzdem kategorisiert empfangen werden können, bietet sich das Publish-Subscribe-Modell an. Sender schicken Nachrichten in der MOM an ein sogenanntes Topic, welches einer Kategorie entspricht. Empfänger können einen oder mehrere Topics abonnieren und empfangen Nachrichten, die an diese gesendet werden. Wird ein Topic von mehreren Empfängern abonniert, werden alle zugehörigen Nachrichten an all jene verteilt. Topics können hierarchisch aufgebaut sein, sodass manche Empfänger nur Nachrichten einer bestimmten Unterkategorie empfangen (bspw. *Grenzwertüberschreitung*), während andere alle Nachrichten empfangen (bspw. *Benachrichtigungen*) [Cur04].

2.4 Deployment

Der Begriff *Deployment* bezeichnet die Prozesse, die nach der Entwicklung für den Einsatz einer Software in Zielsystemen benötigt werden und stellt hiermit einen zentralen Bestandteil der Verwaltung einer IT-Infrastruktur dar. Zu diesen Prozessen gehört die Installation, die Aktivierung, Aktualisierungen der Software sowie die Deaktivierung und das Entfernen der Software aus dem System [CFH+98]. Diese einzelnen Unterprozesse werden fundamental von der gewählten Art des Deployments beeinflusst, also wie und in welcher Form die Software deployt werden soll, und dem Ziel des Deployments, was beispielsweise eine Cloud-Plattform sein kann. Ein vollständig manuelles Deployment wird hierbei außer Acht gelassen, da es für diese Arbeit irrelevant ist. Die *Provisionierung*, also Bereitstellung der notwendigen Ressourcen, wird hierbei nicht als Teil des Deployment-Prozesses gesehen.

2.4.1 Skripte

Das Deployment bzw. Undeployment kann über Skripte, also kleine ausführbare Programme, durchgeführt werden. In diesen Skripten werden die einzelnen Schritte der Unterprozesse programmatisch beschrieben — beginnend bei der Konfiguration und Aktivierung bis zur Deaktivierung und deren Löschung. Diese Skripte können in verschiedenen Programmiersprachen geschrieben sein und werden entweder manuell oder von anderen Skripten aufgerufen. Mit Hilfe von Frameworks können diese Skripte standardisiert implementiert und aufgerufen werden. Typische Beispiele hierfür sind Chef[12] oder Ansible[13].

[12] Chef, https://www.chef.io/
[13] Ansible, https://www.ansible.com/

2.4.2 Virtuelle Maschine

Um die Software isoliert von äußeren Einflüssen auch leicht auf mehreren Systemen deployen zu können, kann sie als Ergebnis der Entwicklung in einer virtuellen Maschine als Grundlage für das Deployment vorliegen. Nach dem Starten der virtuellen Maschine im festgelegten, virtualisierten Betriebssystem mit einer definierten Umgebung kann die Anwendung bereits vollständig nutzbar sein. Im einfachsten Fall kann zur Deaktivierung die virtuelle Maschine heruntergefahren oder gelöscht werden. Das Ausführen einzelner Skripte ist somit nicht mehr notwendig; es genügt die virtuelle Maschine mit Tools wie Oracle VirtualBox[14] oder vRealize Orchestrator[15] zu steuern.

2.4.3 Container

Das Deployment von Containern verhält sich ähnlich zu dem von virtuellen Maschinen, da sie hierbei als leichtgewichtige virtuelle Maschinen betrachtet werden können. Aufgrund der deutlich größeren Kompaktheit einer containerisierten Anwendung im Vergleich zu einer virtuellen Maschine gestalten sich Prozesse zur Verwaltung von Containern leichter und schneller. Zum Deployen und Steuern von Containern bietet sich beispielsweise Docker selbst oder auch Rancher[16] als Verwaltungsplattform an.

2.5 OPC Unified Architecture

Mit den in der Spezifikation der *OPC Unified Architecture (OPC-UA)* definierten Standards beschreibt die OPC Foundation, wie eine Interoperabilität zur Kommunikation mit Industriemaschinen erreicht werden kann. Durch einen Service-orientierten Ansatz können Maschinen aktuelle und historische Daten plattform- und herstellerunabhängig austauschen sowie auf Ereignisse reagieren. OPC-UA spezifiziert auch, wie die Sicherheit des Datenaustauschs bspw. mit SSL oder AES gewährleistet werden kann und wie die Daten aufgebaut sein können, um auch von unterschiedlichen Maschinen gleich interpretierbar zu sein. Durch konfigurierbare Timeouts und automatische Fehlerbehebungsmechanismen kann OPC-UA abgebrochene Verbindungen ohne Datenverlust wiederherstellen [LM06; OPC16].

Wie in Abbildung 2.7 dargestellt, gibt OPC-UA ein Client-Server-Modell mit Programmbibliotheken für die Endpunkte vor. Eine Anwendung verwendet die Implementierung eines OPC-UA-Clients, um hiermit über eine wählbare Kommunikationstechnologie Daten auszutauschen. Der Client abonniert ähnlich zum Publish-Subscribe-Modell beim Messaging (vgl. Abschnitt 2.3.3) beispielsweise Messdaten des Servers, der diese daraufhin an den Client schickt. Je nach Konfiguration ist es ebenfalls möglich, dass Clients Werte an den Server senden und somit Parameter festlegen können. Die einzelnen Werte haben hierbei definierte Attribute wie eine *NodeId*, einen Namen oder auch einen Datentyp, welcher durch ein XML-Schema die Struktur beschreiben kann [LM06; OPC16].

[14] Oracle VirtualBox, https://www.virtualbox.org/
[15] vRealize Orchestrator, https://www.vmware.com/de/products/vrealize-orchestrator.html
[16] Rancher, http://rancher.com/rancher/

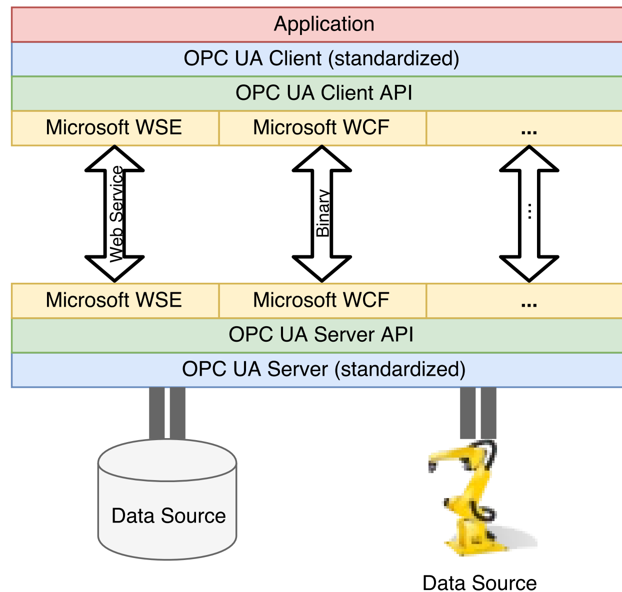


Abbildung 2.7: Der Aufbau einer Anwendung, die OPC-UA für die Kommunikation mit Werkzeugmaschinen nutzt (angelehnt an [LM06]).

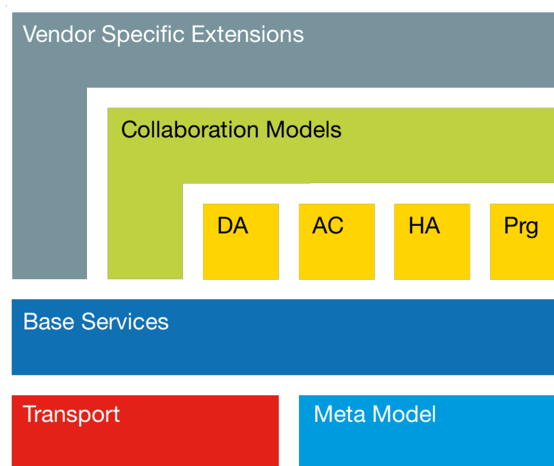


Abbildung 2.8: Der Aufbau von OPC-UA mit den Grundbestandteilen und darauf aufbauenden Erweiterungen [OPC16].

Auf unterster Ebene (vgl. Abbildung 2.8) unterstützt OPC-UA ein binäres Transportprotokoll für eine Kommunikation mit Schwerpunkt auf Geschwindigkeit und Durchsatz sowie ein auf SOAP und HTTP basierendes Protokoll. Mit einem Meta-Modell können vorhandene Basis-Typen als Grundlage für die Modellierung eigener Datenstrukturen genutzt werden. Die Basis-Services können bieten eine Schnittstelle zu den verschiedenen Funktionen des OPC-UA-Servers an, um bspw. die Strukturen des Servers zu durchsuchen oder Werte von Knoten abzurufen und zu ändern. Informationsmodelle für den Zugriff auf Live- (DA) sowie historische Daten (HA), Monitoring und Events (AC) oder zur Definition von komplexen Zustandsmaschinen (Prg) gehören zu den grundlegenden OPC-UA-Funktionen und können durch weitere Modelle ergänzt werden.

2.6 Software-Qualitäten

Im ISO-Standard 25010 für System- und Softwarequalitätsmodelle [ISO10] wird als Nachfolger von ISO 9126 mit dem *Product Quality Model* ein Qualitätenmodell aufgestellt, welches unter anderem die Zuverlässigkeit definiert und auch die Skalierbarkeit aufführt.

Zuverlässigkeit Die Zuverlässigkeit beschreibt, inwiefern eine Software (bzw. ein Produkt oder eine Komponente) die geforderte Funktionalität in einer definierten Umgebung für eine bestimmte Dauer erfüllt. Sie lässt sich mit den folgenden vier Charakteristiken beschreiben:

Reife Die Reife beschreibt, inwiefern ein Produkt im Normalbetrieb zuverlässig ist.

Verfügbarkeit Eine hohe Verfügbarkeit wird erreicht, wenn das Produkt funktionsfähig und erreichbar ist.

Fehlertoleranz Nach Fehlern in Hard- oder Software verhält sich ein fehlertolerantes Produkt wie beabsichtigt.

Wiederherstellbarkeit Der Normalbetrieb wird bei einer hohen Wiederherstellbarkeit nach einem (Teil-)Ausfall zusammen mit beschädigten Daten schnell wiederhergestellt.

Skalierbarkeit Die Skalierbarkeit wird nur als Teil der Anpassbarkeit im ISO-Standard erwähnt, welcher eine Charakteristik der Portabilität einer Anwendung ist. Sie beschreibt, inwiefern der Durchsatz eines Produkts oder einzelner Komponenten verändert werden kann, um einen anderen Durchsatz zu leisten. Hierbei kann zwischen einer horizontalen und einer vertikalen Skalierung unterschieden werden. Bei einer vertikalen Skalierung werden die Eigenschaften einer Ressource geändert, sodass eine virtuelle Maschine bspw. mehr Recheneinheiten erhält. Die horizontale Skalierung hat zur Folge, dass sich die Anzahl an Instanzen ändert, also mehrere Maschinen verfügbar wären. Die Effektivität einer Skalierung sagt dabei aus, ob der Ressourcenverbrauch bzw. der Datendurchsatz proportional zu der Änderung der Instanzen ist [Neu94; VRB11].

3 Verwandte Arbeiten

Im Folgenden werden verwandte und thematisch ähnliche Arbeiten beschrieben und zu dieser Arbeit abgegrenzt. Die einzelnen wissenschaftlichen und technischen Arbeiten sind hierbei nach ihrem jeweiligen Themengebiet gruppiert, wobei allerdings auch leichte Überschneidungen bestehen.

3.1 Service-Definition

Die Services der in dieser Arbeit konzipierten Cloud-Plattform sind ein zentraler Bestandteil, da sie mit allen Teilbereichen in Berührung stehen. Eine einheitliche Definition kann für eine Wiederverwendung und Weiterverbreitung von Services nicht außer Acht gelassen werden.

3.1.1 Web Services Description Language

Mit der *Web Services Description Language (WSDL)* hat das *World Wide Web Consortium* eine Beschreibungssprache für Web Services, also netzwerkbasierte Anwendungen, aufgestellt, welche momentan in Version 2.0 verfügbar ist. WSDL ermöglicht es, Web-Services und deren Schnittstellen im XML-Format zu beschreiben. Die Definition wird hierbei auf zwei verschiedenen Ebenen vorgenommen: abstrakt und konkret. Das Format der Nachrichten wird üblicherweise als XML-Schema im abstrakten Teil beschrieben und mit einem *Message Exchange Pattern* in einer Funktionsbeschreibung (*Operation*) zusammengefasst. Das Message-Exchange-Pattern beschreibt die Anzahl der gesendeten Nachrichten einer Schnittstelle und gibt die Richtung der Nachrichtenkommunikation an. Alle Funktionen werden mit den jeweiligen Datentypen in einer gemeinsamen Schnittstelle (*Interface*) unabhängig von Kommunikationsprotokollen oder ähnlichem zusammengefasst. Im konkreten Teil wird angegeben, wie genau ein anderer Service auf die oben beschriebenen Schnittstellen zugreifen kann und wie die Kommunikation stattfinden soll. In *Bindings* werden die abstrakt definierten Funktionen mit Kommunikationsprotokollen verknüpft, auf welche aus sogenannten *Endpoints*, welche die Zieladresse eines Dienstes enthalten, verwiesen wird. WSDL 2 wurde 2007 veröffentlicht und hat fast alle Konzepte aus der ersten Version übernommen und erweitert. Zusammengefasst lassen sich mit der WSDL nur die äußeren Merkmale eines Services, die zur Kommunikation benötigt werden, beschreiben und keine Informationen zum Aufbau des Services selbst [CCMW01; CMRW07].

3.1.2 OASIS TOSCA

Mit dem Standard *TOSCA (Topology and Orchestration Specification for Cloud Applications)* stellt die *OASIS Standards Group* mit Unterstützung von großen Cloud-Anbietern wie IBM, HP oder

Google eine Notation zur Verfügung, mit der die Topologie von Software sowie Prozesse, die zum Betrieb dieser Software gehören, beschrieben werden können. TOSCA kann sowohl den vollständigen Anwendungsstack und die Komposition von beliebigen Anwendungen in einem Paket abbilden als auch die einzelnen Schritte, die zur Konfiguration und dem Betrieb notwendig sind. Hiermit sollen die Portabilität und das Management von Cloud-Anwendungen und -Services vereinfacht werden, indem der Vendor-Lock-in bei einem Anbieter abgewendet und die Nutzung mehrerer Cloud-Anbieter ermöglicht wird.

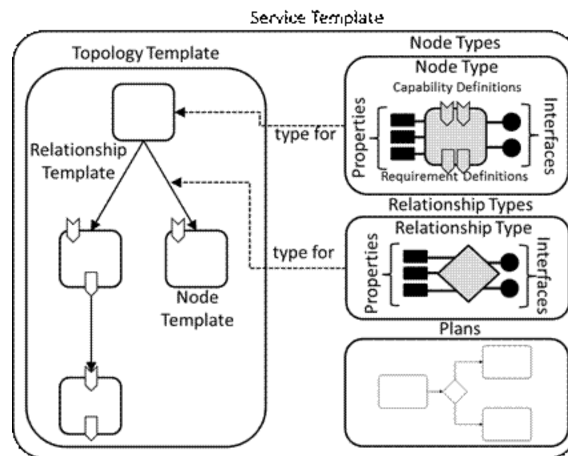


Abbildung 3.1: Eine schematische Darstellung eines TOSCA-Service-Templates und der enthaltenen Komponenten [OAS13].

Die Prozesse (*Plans*) und die Topologie (*Topology Template*) einer Anwendung werden, wie in Abbildung 3.1 gezeigt, in einem sogenannten *Service Template* zusammengefasst. Dieses enthält die grundlegenden Informationen zu den einzelnen Software-Komponenten (*Node Type*) und Konnektoren (*Relation Type*) zwischen ihnen. Das Topology-Template verwendet eben diese Bausteine, um die gesamte Anwendung aufzubauen und im Endeffekt als Graph zu beschreiben, wobei Konnektoren und Software-Komponenten nun auf die jeweiligen Typen verweisen. Die Prozesse können hierbei mit Modellierungssprachen wie BPMN oder BPEL beschreibbare sowohl manuelle als auch automatisierte Prozesse umfassen. Die notwendigen Softwareartefakte und andere Ressourcen werden mit dem Service-Template in einem sogenannten *Cloud-Service-Archive (CSAR)* ausgeliefert. Als Resultat erhält man einen vollständigen Bauplan eines Services bzw. einer Service-Komposition, welcher von einer TOSCA-Runtime ausgeführt werden kann, um eine lauffähige Anwendung zu erhalten. Seit Mitte 2016 kann statt in einem XML-Dokument auch ein kompakteres und besser lesbares YAML-Dokument für die Definition des Service-Templates verwendet werden [OAS13; OAS16].

Implementierungen

Da TOSCA selbst nur eine reine Spezifikation ist, die keine Pläne interpretieren und ausführen kann, ist eine separate Laufzeitumgebung hierfür notwendig.

OpenTOSCA ist eine Laufzeitumgebung für TOSCA, welche als Forschungsprojekt in einem Open-Source-Projekt[1] an der Universität Stuttgart entwickelt wird. Das Projekt bezeichnet sich selbst als Prototypen, kann allerdings Archive im CSAR-Format interpretieren und das Service-Template und Pläne ausführen, wie in mehreren Beispielen auf der Projekt-Webseite gezeigt wird. Zusammen mit dem inzwischen zu Eclipse gehörenden *Winery*[2], einem graphischen Modellierungstool für TOSCA-Templates und -Plans sowie *Vinothek* als Oberfläche zum Steuern von OpenTOSCA ergibt sich ein rundes Gesamtpaket [BBH+13; BBKL14].

Cloudify[3] bietet ebenfalls als Open-Source-Software bereits eine deutlich ausgereiftere Implementierung der TOSCA-Spezifikation im Vergleich zu OpenTOSCA. Außer der reinen TOSCA-Laufzeitumgebung bietet Cloudify einen automatisierten Workflow über den gesamten Anwendungszyklus und Continuous-Deployment-Prozesse hinweg, während auch eine Anbindung an andere Tools und Orchestrierungen vorhanden ist.

Mit *Alien 4 Cloud*[4] als weiterer verbreiteter TOSCA-Implementierung, die ebenfalls Open-Source verfügbar ist, existiert noch eine Möglichkeit, wie sowohl eine containerisierte als auch eine mit herkömmlichen Deployment-Methoden verbreitete Software standardisiert verwaltet werden kann. Die Orchestrierungsfähigkeiten von Cloudify können hierbei in Alien 4 Cloud verwendet werden.

3.2 Container-Management

Werden einzelne Anwendungen oder Kompositionen mit einer Servicedefinition wie TOSCA beschrieben, ist es nötig, die in Abschnitt 2.4 über das Deployment beschriebenen Prozesse steuern zu können. Darüber hinaus gehören noch Aufgaben, um solche Deployments koordinieren und miteinander arrangieren zu können. In den folgenden Abschnitten werden Anwendungen mit Orchestrierungsfähigkeiten für mit Docker containerisierte Anwendungen beschrieben, da Docker-Container das zur Zeit der Arbeit meist genutzte und unterstützte Container-Format sind.

3.2.1 Docker Swarm

Docker Swarm[5] heißt die native Orchestrierung von Docker. Bis zur Version 1.12 war Docker Swarm eine separate Anwendung, ist allerdings seitdem neu als *Swarm Mode* in Docker selbst integriert. Docker Swarm bietet ein direkt in die Docker-Engine integriertes Cluster-Management, wobei solch ein Cluster *Swarm* genannt wird. Knoten in diesem Cluster werden hierbei in zwei Gruppen unterteilt: *Manager* und *Worker*. Manager-Knoten übernehmen Wartungsaufgaben, nehmen Service-Definitionen entgegen und übersetzen diese in Aufgaben, die von den Worker-Knoten — zu welchen Manager-Knoten standardmäßig auch gehören — übernommen werden. Docker spannt zwischen allen Knoten ein verschlüsseltes virtuelles Netzwerk auf, welches die

[1] OpenTOSCA auf Github, https://github.com/OpenTOSCA
[2] Winery, https://github.com/eclipse/winery
[3] Cloudify, http://cloudify.co/
[4] Alien 4 Cloud, https://alien4cloud.github.io/
[5] Docker Swarm, https://docs.docker.com/engine/swarm/

Kommunikation standardmäßig sichert, und verteilt anschließend die Services je nach Anforderungen auf allen Worker-Knoten. Services sprechen sich gegenseitig über Domainnamen an, die von einem DNS-Server im virtuellen Netzwerk aufgelöst werden. Werden Services skaliert oder rollierend aktualisiert, übernimmt der DNS-Server die Lastverteilung zu den neuen Services.

Die Orchestrierung von Docker-Swarm kann über die Docker-eigene API angesprochen werden und ist somit auch aus verschiedenen Programmiersprachen über SDKs (Software-Development-Kits) nutzbar, bietet allerdings außer der Kommandozeileneingabe keine andere Benutzerschnittstelle. Über die offizielle Docker-API können somit auch alle weiteren relevanten Container-Informationen abgerufen werden.

3.2.2 Kubernetes

Kubernetes[6] ist eine von Google Open-Source entwickelte Container-Orchestrierung. Kubernetes baut auf Googles jahrelanger Erfahrung auf und ist auch auf enorm große Deployments ausgelegt. Wie auch Docker Swarm wird in einem Cluster von Servern zwischen Master- und Worker-Knoten unterschieden. Master-Knoten übernehmen Skalierungs-, Replikations- und Verwaltungsaufgaben, während Worker-Knoten die Container verwalten. Einzelne Anwendungen oder Kompositionen aus ihnen werden in sogenannten Pods gruppiert und bestehen aus mehreren Containern. Services machen Pods nach außen verfügbar und stellen ebenfalls Load-Balancer-Funktionalitäten bereit. Mit Kubernetes werden Container automatisch so verteilt, dass die Ressourcen der zugrundeliegenden Server bestmöglich genutzt werden, und ggf. ausfallende Container bzw. Cluster-Knoten automatisch kompensiert [Chi17].

3.2.3 Apache Mesos & Marathon

Apache Mesos[7] ist ein verteilt arbeitender Systemkernel, der durch Abstraktion von CPU, Arbeitsspeicher und weiteren Ressourcen physische und virtuelle Server in einer virtuellen Sicht für Anwendungen verbindet. Mesos ermöglicht Anwendungen und Frameworks eine flexible Nutzung unabhängig von den zugrundeliegenden Ressourcen und ist damit nicht auf die Nutzung von Docker-Containern beschränkt. Der reine Cluster-Manager lässt sich allerdings mit dem Framework *Apache Marathon* um eine Container-Orchestrierung erweitern. Hierfür wird vom Mesos-Master-Knoten über Marathon die auf den Mesos-Slave-Knoten laufenden Docker-Exekutoren angesteuert. Mesos und Marathon erlauben wie auch Kubernetes die automatische Skalierung von Services und die Lastverteilung über skalierte Services hinweg, sowie virtuelle Netzwerke, die Speicherverwaltung der Container und weitere. Mesos ist durch die Erweiterung durch Frameworks hochflexibel und außerdem über eine REST-API ansprechbar.

[6] Kubernetes, https://kubernetes.io/
[7] Apache Mesos, http://mesos.apache.org/

3.2.4 Rancher

Rancher[8] bezeichnet sich als die vollständigste Container-Management-Plattform, die es gibt. Wie auch bei den zuvor genannten Plattformen kann mit Rancher Container modular eine Server-Infrastruktur betrieben werden, während Ausfälle automatisch kompensiert werden können und auf eine hohe Benutzbarkeit Wert gelegt wird. Rancher bringt wie die anderen Lösungen zwar auch eine eigene Container-Orchestrierung mit sich, kann allerdings alle der oben genannten Orchestrierungen verwenden.

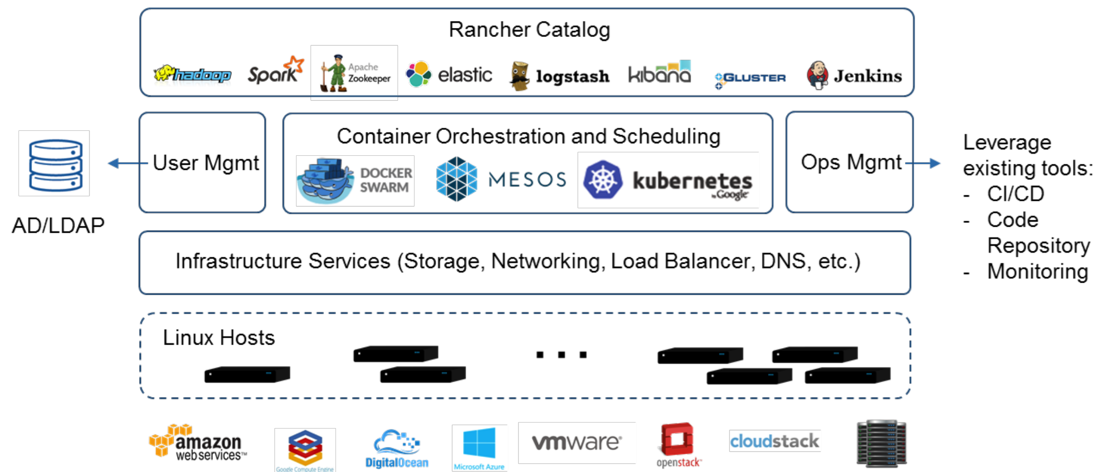


Abbildung 3.2: Die Hauptkomponenten von Rancher beginnend bei den Linux-Hosts der Cloud-Anbieter bis zu den Cloud-Services [Ran].

Rancher ist anders als Apache Mesos explizit für die Verwaltung und Orchestrierung von Containern entwickelt worden. Anwender können in Rancher entweder zahlreiche Anwendungen aus einem vordefinierten Katalog oder beliebige Docker-Images abrufen und als Container deployen. Über ein vordefiniertes Format können außerdem Kompositionen von Docker-Services eingespielt und vollständig konfiguriert werden. Rancher verwendet hierbei eine Rancher-Server-Komponente, um auf beliebigen Linux-Docker-Hosts laufende Rancher-Agents zu steuern. Anwendungen werden in sogenannten Stacks gruppiert und können aus mehreren einzelnen Services bestehen. Der Orchestrator verteilt diese Services auf allen Hosts, insofern sie etwaigen Anforderungen entsprechen. Rancher ermöglicht außerdem die direkte Anbindung von Knoten aus bekannten Cloud-Plattformen wie der Google Cloud, den Amazon Web Services oder Microsoft Azure.

3.3 Cloud-Anwendungs-Management

Cloud-Anwendungen können auf verschiedenen Ebenen bereits von PaaS-Plattformen oder Anwendungen gesteuert werden. Das Ziel, einen Vendor-Lock-in durch Unterstützung von mehreren Cloud-Anbietern zu vermeiden und gleichzeitig die Vorteile der gewonnenen Vielfalt zu nutzen,

[8] Rancher, https://rancher.com/rancher/

wurde bereits in einigen weiteren Projekten angestrebt. Manche Projekte nutzen teilweise bereits vorgestellte Konzepte wie TOSCA zur Service-Definition, während andere komplett eigene Ansätze verfolgen.

3.3.1 Cloud-Deployment-Tools

Bei einer Vielzahl verschiedener Cloud-Anbieter mit jeweils eigenen Konzepten, Schnittstellen und Prozessen ist es erforderlich, jeweils eine separate Programmierschnittstelle zu nutzen, um bspw. die erforderlichen Deployment-Prozesse zu steuern. Cloud-Anbieter bieten nicht nur reine Ressourcen für Berechnungen, sondern bieten auch verschiedene Speicherdienste wie virtuelle Datenträger, einfache Schlüssel-Werte-Speicher oder auch relationale Datenbanken.

Wie zuvor bereits erwähnt, unterstützen teilweise Tools wie Ansible oder Rancher das Deployment auf verschiedene Cloud-Plattformen, allerdings nur, wenn die Funktionalität der restlichen Plattform genutzt wird. Die Bibliothek *Apache jclouds*[9] abstrahiert solche Funktionen von Cloud-Anbietern (*Providern*) in verschiedenen Ansichten (*Views*). Gleichzeitig können dennoch Schnittstellen von einzelnen oder Gruppierungen von Providern genutzt werden, um spezifische Funktionen anzusprechen. Soll eine Anwendung um eine Multi-Cloud-Unterstützung erweitert werden, kann mit solch einer Programmbibliothek portabel eine Anbindung an mehrere Cloud-Anbieter angebunden werden, während gleichzeitig die Komplexität bei der Entwicklung reduziert und dennoch ein direkter Zugriff auf die Schnittstellen erhalten bleibt. Das inzwischen eingestellte Projekt *Apache Whirr*[10] hat Nutzen von Apache jclouds gemacht und konnte das Deployment von verteilten Anwendungen organisieren.

3.3.2 Cloud-Management

Verschiedene Projekte haben sich bereits zum Ziel genommen, das Management von Cloud-Anwendungen bei mehreren Cloud-Anbietern zu vereinheitlichen.

OASIS CAMP

CAMP — kurz für *Cloud Application Management for Platforms* — ist wie TOSCA ein Standard von OASIS mit dem Zweck, die Interoperabilität zwischen verschiedenen PaaS-Plattformen zu ermöglichen. Der Standard beschreibt die Anforderungen an PaaS-Anbieter, die die Entwicklung von Anwendungen und Services ermöglichen, mit denen PaaS-Dienste unabhängig vom Anbieter und der zugrundeliegenden Infrastruktur genutzt werden können. Es werden Schnittstellen, Prozesse und Protokolle beschrieben die für eine einheitliche Self-Service-Schnittstelle notwendig sind. Für die Spezifikation von CAMP existieren wie bei TOSCA exemplarische Implementierungen [BIS+14; OAS14]. So bietet bspw. das OpenStack-Projekt Solum[11] eine unvollständige Integration von CAMP, während das im folgenden Abschnitt beschriebene Apache Brooklyn deutlich ausgereifter ist.

[9] Apache jclouds, https://jclouds.apache.org/
[10] Apache Whirr, http://whirr.apache.org/
[11] Solum, https://wiki.openstack.org/wiki/Solum

Apache Brooklyn

Apache Brooklyn[12] ist eine Anwendung, um Cloud-Applikationen mit Hilfe von Definitionsdateien entweder über eine Nutzeroberfläche oder eine REST-API zu modellieren, deployen und zu verwalten. Es bietet eine ausgereifte Unterstützung von CAMP und TOSCA zusammen mit eigenen Bauplänen (*Blueprints*) für Kompositionen von Services und einer Multi-Cloud-Unterstützung. Die eigenen Blueprints werden hierbei im YAML-Format spezifiziert, welche ähnlich zu denen von CAMP und TOSCA sind, und ermöglichen es Anwendungen mit eigenen Deployment-Skripten direkt in die Anwendung zu integrieren. Brooklyn wandelt die Blueprints in äquivalente Deployments um und sammelt von diesen Metriken, um den Status des Deployments zu verwalten. So wird bspw. automatisch die Latenz von Abfragen an deployte Services gemessen und mit Richtwerten verglichen, um anschließend zu entscheiden, ob ein Service wegen eines Fehlers neu gestartet oder wegen einer hohen Auslastung skaliert werden muss. Die Zielplattformen beschränken sich hierbei nicht nur auf Cloud-Anbieter, da mithilfe von Apache jclouds auch eine generische Anbindung an nicht-Cloud-Umgebungen gegeben ist.

SeaClouds

Das EU-Forschungsprojekt *SeaClouds*[13] strebte eine anpassbare Multi-Cloud-Verwaltung von komplexen Anwendungen durch Unterstützen von Deployment-, Migrations- und Monitoring-Prozessen in PaaS- und IaaS-Umgebungen an, ist allerdings seit März 2016[14] nicht mehr aktiv. SeaClouds plante eine Service-Orchestrierung für verschiedene Cloud-Anbieter, Monitoring und Management von diesen Services während gängige Standards genutzt werden sollen. SeaClouds verwendet das ebenfalls eingestellte Forschungsprojekt Cloud4SOA, um interoperabel über verschiedene Cloud-Plattformen hinweg die Anwendungen deployen und monitoren zu können.

Wie in Abbildung 3.3 gezeigt, nutzt SeaClouds einige der bereits vorgestellten Forschungsprojekte und Tools. Mit TOSCA beschriebene Anwendungen können von SeaClouds orchestriert werden - was allerdings eine TOSCA-Laufzeitumgebung auf allen Zielplattformen erfordert. Für IaaS-Plattformen wird Apache Brooklyn mit Whirr und jClouds für das Deployment und zum Monitoring genutzt. Auf PaaS-Plattformen wird hierfür Cloud4SOA genutzt und auch direkt CAMP-kompatible Schnittstellen aus SeaClouds angesprochen [BIS+14].

3.3.3 Industrie-Cloud-Plattformen

In den letzten Jahren wurde das Potential für Cloud-Services im Industriebereich im Rahmen von Industrie 4.0 bereits erkannt, wodurch sich mehrere Forschungsprojekte und Produkte in diesem Bereich ergeben haben.

[12] Apache Brooklyn, https://brooklyn.apache.org/
[13] SeaClouds, http://www.seaclouds-project.eu/
[14] SeaClouds Projektinformationen, http://www.seaclouds-project.eu/node/258

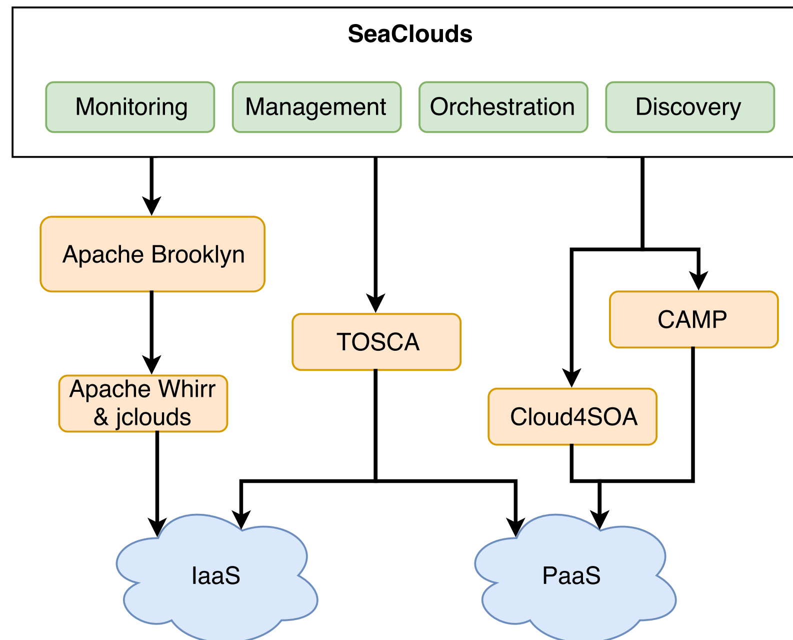


Abbildung 3.3: Der Kontext von SeaClouds und die Verbindung zu den verwendeten Technologien (angelehnt an [BIS+14]).

ADAMOS IIoT-Plattform

Im September 2017[15] gründeten wenige namhafte Maschinenbauer und IT-Konzerne wie ZEISS oder die Software AG das Joint Venture *ADAMOS*[16] *(ADAptive Manufacturing Open Solutions)*. ADAMOS hat das Ziel, die Kompetenzen aus Maschinenbau, Produktion und Informationstechnik gebündelt in einer Service-Plattform zu nutzen. Hierfür werden eine Umgebung für die Entwicklung von solchen Services sowie eine IIoT-Plattform (Industrial Internet of Things) aufgebaut. Diese Plattform soll zum einen als digitaler Marktplatz und zum anderen zentral für die Produktion und dabei aufkommende Daten und Verarbeitungsprozesse dienen. ADAMOS wirbt mit dem Vermeiden eines Vendor-Lock-ins und dem Erhalt der Datenhoheit bei gleichzeitiger Analyse bspw. mit Machine-Learning-Technologien der Daten, Workflow-Automation und der Integration in Cloud-Dienste. Zum Ende dieser Arbeit wurden allerdings noch keine Ergebnisse veröffentlicht.

AXOOM IoT Plattform

AXOOM SOLUTIONS bietet mit der *AXOOM IoT Plattform*[17] eine ähnliche Lösung zu der oben genannten Plattform von ADAMOS. Diese Plattform soll Industriefirmen ebenfalls dabei helfen, mit einer Anbindung an verschiedenste Sensoren und Maschinen durch Echtzeit-Analyse der

[15] Pressemitteilung zur Gründung von ADAMOS, http://www.softwareag.com/de/company/press/news/20170905_adamos
[16] ADAMOS, https://de.adamos.com/
[17] AXOOM IoT Plattform, http://www.axoom-solutions.com/axoom-iot-plattform/

Daten bspw. im Gebiet der Predictive-Maintenance — also einer „vorausschauenden Wartung“ — voranzuschreiten. Besitzer von Industriemaschinen sollen einen Mehrwert durch die Aufbereitung der Maschinendaten und durch Apps erhalten, während sich die Hersteller der Maschinen durch einen Support-Zugang zu den Maschinen verbinden können. Detaillierte Informationen zur Plattform, deren Funktionsweise und Cloud-nähe sind wie bei der ADAMOS IIoT-Plattform nicht öffentlich verfügbar.

Project pICASSO

Das Forschungsprojekt pICASSO[18], hinter welchem sich das rekursive Akronym *pICASSO – Industrielle CloudbASierte SteuerungsplattfOrm für eine Produktion mit cyber-physischen Systemen* verbirgt, soll auch genau das eben Beschriebene erforscht werden. Der bisherige Datenfluss einer Steuerungseinheit geht heutzutage üblicherweise aus Ist-Werten und Berechnungen von Algorithmen zu Soll-Werten, die an die Maschinen gesendet werden. Zukünftig soll die Steuerungstechnik über eine Cloud bereitgestellt werden, um hierdurch eine modulare und skalierbare serviceorientierte Softwarearchitektur zu erhalten sowie Mechanismen von Cloud-Computing nutzen zu können. Trotz der Verlagerung der Maschinen-Module in die Cloud (vgl. Abbildung 3.4) sollen Anforderungen an eine Echtzeitverarbeitung oder auch die Sicherheit über eine Netzwerkbrücke zwischen Cloud und Industriemaschine gewährleistet werden. Die Netzwerkbrücke verbindet hierbei die Werkzeugmaschine mit den in der Cloud laufenden Steuerungsmodulen für die Kommunikation (COM), Benutzerschnittstellen (HMI) oder programmierbare Logik-Controller (PLC). Durch die Cloud-Umgebung der Module können bei höheren Leistungsanforderungen den jeweiligen Modulen mehr Leistung zugewiesen werden.

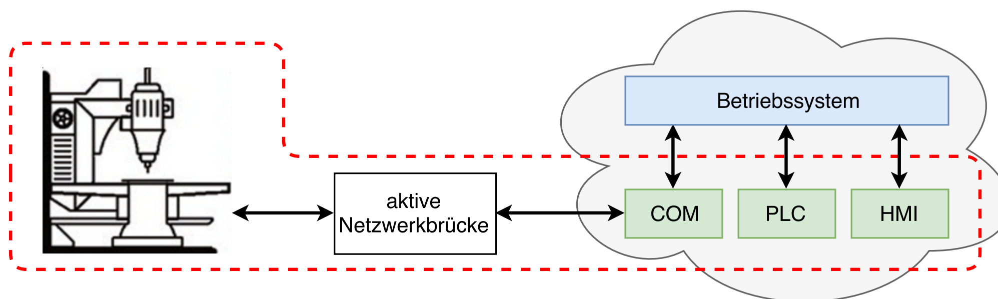


Abbildung 3.4: Der Aufbau einer Werkzeugmaschine mit in die Cloud verlagerter Steuerung, sodass eine cloudbasierte Werkzeugmaschine (rote Linie) entsteht (angelehnt an [LS17]).

Nach den Veröffentlichungen des Projekts und zugehörigen studentischen Arbeiten scheint keine vollständige Plattform entstanden zu sein. Weitere Ergebnisse werden nicht erwartet, da das Projekt 2016 ausgelaufen ist [LS17].

[18]pICASSO, https://www.projekt-picasso.de/

Virtual Fort Knox

Als Pilotprojekt des Wirtschaftsministeriums des Landes Baden-Württemberg steht auch das *Virtual Fort Knox*[19] *(VFK)* vom Fraunhofer Institut für Produktionstechnik und Automatisierung im Raum. Bei VFK stehen nicht unbedingt nur globale Player als Zielgruppe im Vordergrund, sondern auch kleinere und mittelständische Unternehmen, die von der Vernetzung mit Hilfe einer Cloud-Plattform profitieren sollen. Virtual Fort Knox hat dabei die folgenden drei Hauptziele:

- „Horizontale Integration über Wertschöpfungsnetzwerke“ [Die14]
- Konsequente Digitalisierung der Wertschöpfungskette
- „Vertikale Integration und vernetzte Produktionssysteme“ [Die14]

Datenquellen wie intelligente Lager, Werkzeuge und Maschinen, Robotersysteme oder auch Mitarbeiter sollen selbst über Unternehmensgrenzen hinaus in einer Plattform zusammengeführt werden, wobei das geistige Eigentum sichergestellt bleiben soll. Daten sollen nur in dem Maß ausgetauscht werden, in dem sie für kooperative Aktivitäten bspw. zwischen Firmen notwendig sind. Einzelne Services sowie Workflows aus mehreren sollen durch eine Aggregation von Services über ein Marktplatz-Portal aufgesetzt werden können. Nachdem Anwender Services auf dem Marktplatz gekauft haben, können diese mit den technischen Komponenten verknüpft werden. Über einen sogenannten *Manufacturing Service Bus (MSB)* als zentralen Kommunikationsbestandteil können hierbei betriebswirtschaftliche und produktionsnahe Systeme angebunden werden. Virtual Fort Knox legt hierbei einen großen Wert auf die Sicherheit der Plattform und der Daten, da das Vertrauen der teilnehmenden Firmen eine enorme Rolle spielt [Die14].

3.4 Zuverlässigkeit & Skalierbarkeit

Allgemein hängt die Zuverlässigkeit und die Skalierbarkeit einer Cloud-Anwendung stark davon ab, wie bestimmte Technologien in eine speziellen Konstellation mit bestimmten Anforderungen verwendet werden. Im Folgenden werden deshalb allgemeine Konzepte vorgestellt, die dem Erreichen dieser Anforderungen als Grundlage dienen sollen.

3.4.1 Zuverlässigkeit

Dai et al. stellen in [DYDZ09] ein Modell zur Zuverlässigkeit von Cloud-Anwendungen auf, da bisherige Modelle traditioneller Anwendungen aufgrund der großen Unterschiede zum Cloud-Computing nicht anwendbar sind. Im Gegensatz zu gewöhnlichen Anwendungen werden Cloud-Anwendungen in einem Multi-Cloud-Szenario auf Cloud-Plattformen mit unterschiedlichen Umgebungen und großen Netzwerken betrieben und häufig von einer Vielzahl an anderen Anwendungen verwendet. Die Zuverlässigkeit eines Cloud-Services wird hierbei von vielen Faktoren beeinflusst. Bei der Kommunikation mit anderen Services kann eine Reihe generischer Fehler auftreten, wie ein Überlauf aufgrund einer zu großen Menge an Anfragen oder ein Timeout, wenn Nachrichten nicht rechtzeitig bearbeitet wurden. Bei Ausfällen von Cloud-Ressourcen zur

[19] Virtual Fort Knox, https://www.virtualfortknox.de/de/

Datenspeicherung oder für Berechnungen können bei einer veralteten Konfiguration Anfragen an diese — inzwischen nicht mehr vorhandenen — Services gesendet werden und daraufhin fehlschlagen. Weiterhin kann es auch zu einer Reihe von Ausfällen in anderen Komponenten kommen. Einzelne Services oder Datenbanken können fehlerhaft sein und ausfallen oder aufgrund von Kommunikationsproblemen und Hardwareausfällen nicht erreichbar sein.

Trotz vieler Ausfallmöglichkeiten ist die Zuverlässigkeit im Vergleich zu einem traditionellen Betrieb einer Anwendung in einer Cloud höher. Durch zahlreiche Redundanzen der Hardware, Netzwerke, abstrahierten Cloud-Ressourcen und gespeicherten Daten werden die Auswirkungen einzelner Ausfälle reduziert. Bei einer Kompatibilität zu mehreren Cloud-Plattformen kann durch Vermeiden eines Vendor-Lock-ins die Cloud-Anwendung im Notfall auch zu einem anderen Cloud-Anbieter umgezogen werden [GSR13].

3.4.2 Skalierbarkeit

Vaquero et al. stellen, wie in Abbildung 3.5 gezeigt, verschiedene Ansätze vor, nach denen eine Anwendung und deren Infrastruktur skaliert werden kann. Diese Ansätze wurden auch von Patibandla et al. in [PKM12] aufgriffen und diskutiert.

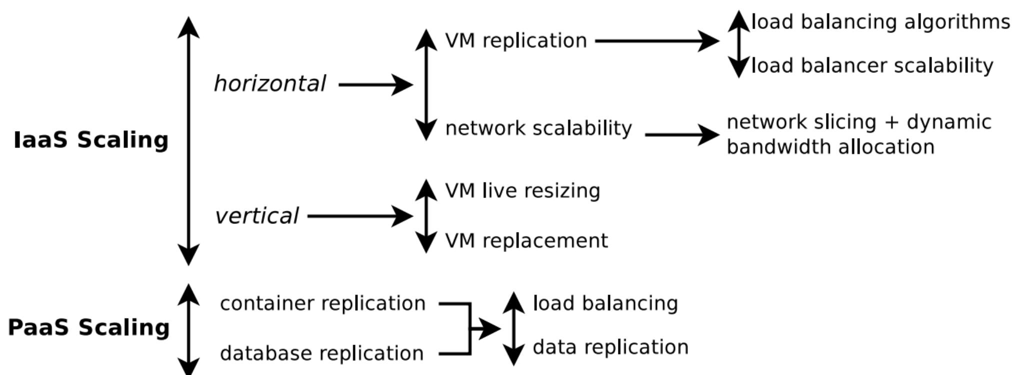


Abbildung 3.5: Ein Überblick über die Möglichkeiten, um eine Cloud-Anwendung und deren Umgebung zu skalieren [VRB11].

In einer IaaS-Cloud können nach dem Ansatz der vertikalen Skalierung bspw. virtuelle Maschinen zur Laufzeit rekonfiguriert werden, sodass ihnen mehr Prozessorkerne, Speicher oder andere Netzwerkadapter zu neuen Netzen zugänglich gemacht werden. Falls dies nicht möglich sein sollte, da evtl. das Betriebssystem hierfür keine Unterstützung bietet, ist es notwendig, die VM mit einer rekonfigurierten VM zu ersetzen. Der horizontale Ansatz bietet hierbei deutlich mehr Möglichkeiten, an denen angesetzt werden kann. Statt einer Ersetzung können einzelne Komponenten ergänzt werden und durch ebenfalls skalierbare Load-Balancer bei der Lastverteilung berücksichtigt werden. Diese Skalierbarkeit kann automatisch durch einen sogenannten *Elasticity Controller* erreicht werden, der die in der Cloud deployten Komponenten überwacht und nach ausgewählten Regeln und Kriterien entscheidet, ob er eine Skalierung über die API der Cloud initiiert. Werden bei einer IaaS-Anwendung nur die virtuellen Maschinen skaliert, ergibt sich eventuell ein Engpass bei den genutzten Netzwerkressourcen. Die Skalierung muss deshalb über verschiedene Ebenen hinweg den Zustand der Ressourcen verstehen und skalieren können.

Beim Einsatz einer PaaS-Cloud fallen diese Aufgaben der Cloud-Plattform zu, die nun mit eigenen Regeln die Anwendungscontainer bei der Skalierung berücksichtigen muss. Um hierbei die Replikation der Anwendungs- und Datenbankcontainer zu erleichtern, sollten die Anwendungen so entwickelt sein, dass sie den gesamten Zustand außerhalb verwalten, wie bspw. ebenfalls von der PaaS-Cloud verwaltete Key-Value-Stores.

3.5 Abgrenzung

Die verwandten Forschungsprojekte und Produkte sollen im Folgenden mit Hinblick auf die Anforderungen der Zielsetzung untereinander verglichen werden.

3.5.1 Service-Definition

	WSDL 2	TOSCA
Metadaten (Name, Autor, ...)	◑	●
Typsicherheit	●	●
Konfigurationsmöglichkeiten	○	●
Öffentliche Schnittstellen	●	○
Benutzerfreundlichkeit	○	◑

● voll erfüllt ◑ teilweise erfüllt ○ nicht erfüllt — unbekannt

Tabelle 3.1: Ein Vergleich von WSDL 2 und TOSCA bezüglich der Wiederverwendbarkeit als Service-Definition.

Die Definition eines Services mittels der WSDL oder TOSCA bietet bereits wichtige Anhaltspunkte wie eine Typisierung von Schnittstellen, zieht allerdings auch eine gewisse Komplexität hinter sicher her (vgl. Tabelle 3.1). Da die WSDL hauptsächlich dafür gedacht ist, die Schnittstellen eines erreichbaren Services zu spezifizieren, gibt es keine Möglichkeit, die eigentlich dahinterstehende Anwendung zu beschreiben und zu konfigurieren. Mit TOSCA kann allerdings genau das Gegenteil erreicht werden: Einer oder mehrere zusammenhängende Services können mit der Definitionsdatei konfiguriert, instanziiert und verknüpft werden. Mit einem Service-Template von TOSCA können allerdings nicht direkt Schnittstellen für die Kommunikation mit externen Services angegeben werden. Darüber hinaus bietet WSDL keine direkte Möglichkeit, um Metadaten anzugeben, und ist aufgrund des XML-Formats weniger gut von Menschen benutzbar, als das YAML-Format von TOSCA.

3.5.2 Container-Management

Container-Management-Plattformen wie Kubernetes, Docker Swarm, Apache Mesos und auch Rancher bieten mächtige Orchestrierungsmöglichkeiten von Docker-Containern in einem Rechner-Cluster und eignen sich somit alle als gute Grundlage. Mit den Plattformen ist es möglich, die grundlegenden Anforderungen an das Deployment über eine Programmierschnittstelle zu steuern. Aufgrund der teilweise unterschiedlichen Ziele ergibt sich ein Mehraufwand beim Einrichten

	Kubernetes	Docker Swarm	Apache Mesos	Rancher
Docker-Container	●	●	●	●
Container-Deployment	●	●	●	●
Constraints auf Hosts	●	●	●	●
Programmierschnittstelle	●	●	●	●
Einfachheit	◑	●	○	●
Automatische Skalierung	◑	○	◑	○
Provisionierung	◑	○	○	●

● voll erfüllt ◑ teilweise erfüllt ○ nicht erfüllt — unbekannt

Tabelle 3.2: Eine Gegenüberstellung verschiedenerer Container-Management-Plattformen.

der Plattformen und bei der Orchestrierung der Container, der dem Erreichen der Zielsetzung dieser Arbeit eher hinderlich ist. Kubernetes und Apache Mesos verfügen zwar über die Fähigkeit, Container automatisch zu skalieren, erlauben allerdings keine Skalierung der zugrundeliegenden Cloud-Ressourcen. Rancher unterstützt außerdem die Provisionierung von Ressourcen von deutlich mehr Cloud-Plattformen als Kubernetes, welcher nur die Google Cloud unterstützt.

3.5.3 Cloud-Deployment-Tools

	Apache jclouds	Ansible	Rancher
Multi-Cloud	●	●	●
Monitoring	◑	◑	●
Programmierschnittstelle	●	◑	◑

● voll erfüllt ◑ teilweise erfüllt ○ nicht erfüllt — unbekannt

Tabelle 3.3: Eine Gegenüberstellung verschiedenerer Container-Deployment-Tools.

Wie die bisherigen Arbeiten zu einem einheitlichen Multi-Cloud-Deployment zeigen, sind direkte Deployments auf verschiedene Cloud-Plattformen mit einer großen Komplexität verbunden. Apache jclouds, Ansible und Rancher bieten hierbei eine ähnliche Funktionalität und unterscheiden sich hauptsächlich in ihrer restlichen Funktionalität, wie dem Konfigurationsmanagement von Ansible oder der Container-Orchestrierung von Rancher. Während Rancher nur eine REST-API als Programmierschnittstelle anbietet, können Ansible und jclouds direkt in die jeweiligen Programmiersprachen eingebunden werden.

3.5.4 Cloud-Management

Die Unterstützung von OASIS CAMP als Standard zur Verwaltung von Cloud-Plattformen ist nach aktuellem Stand kaum in gutem Umfang vorhanden. Im Gegensatz dazu ist Apache Brooklyn weit in seinem Funktionsumfang fortgeschritten und unterstützt typische Anwendungsszenarien als

Teil in einem komplexen Container-Management-Prozesse. Die Ergebnisse des Seaclouds-Projekts sind ebenfalls relevant für eine Service-Plattform für IaaS- und PaaS-Clouds, beschränken sich allerdings auf die rein konzeptionelle Ebene.

3.5.5 Industrie-Cloud-Plattformen

	ADAMOS IIoT-Plattform	AXOOM IoT Plattform	Project pICASSO	Virtual Fort Knox
Öffentlich verfügbar	○	○	●	○
Implementierung vorhanden	●	—	○	●
Unterstützung für Container	—	—	—	—
Vendor-Lock-in vermieden	—	—	—	—

● voll erfüllt ◑ teilweise erfüllt ○ nicht erfüllt — unbekannt

Tabelle 3.4: Der Vergleich von Cloud-Plattformen für die Industrie zeigt, dass bisher kaum öffentlich verfügbare Erkenntnisse vorliegen.

Vollständige Industrie-Plattformen, die eine dynamische Nutzung von Services zu lassen, sind bis jetzt meist rein konzeptionell oder unter Verschluss der Öffentlichkeit. Weiterhin hat keine der untersuchten Plattformen Konzepte gezeigt, mit denen der Vendor-Lock-in bei einem oder wenigen Cloud-Plattformen verhindert oder umgangen wird. Einzelne Konzepte bieten bereits gute Anhaltspunkte, die bei der Konzeption einer Cloud-Plattform für Services einfließen können. Manche Arbeiten verfolgen bereits Ideen für eine Plattform mit Deployment in einem Multi-Cloud-Szenario, allerdings meist unvollständig und ohne praktische Ergebnisse, welche validiert werden könnten. Etwaige Ideen für die Gewährleistung des Betriebs unter produktionstechnischen Bedingungen im Zusammenhang mit einer solchen Cloud-Plattform sind ebenfalls nicht vorhanden.

3.5.6 Zusammenfassung

Bisherige Service-Definitionen sind für andere Anwendungsfälle gedacht und ermöglichen keine kompakte und benutzbare Beschreibung eines Services mitsamt seiner Schnittstellen. Die große Anzahl an Tools zum Deployen und Verwalten von Cloud-Anwendungen kann genutzt werden, um die Services in die Cloud zu bringen und dort zu steuern. Ähnliche Plattformen — auch mit Industrie-Fokus — beschreiben nur Teile der Zielsetzung dieser Arbeit und sind kaum in Forschungsarbeiten veröffentlicht worden. Insgesamt mangelt es an einem validierten Konzept für eine durchgängige Plattform von Service zur Komposition und zum anschließendem Multi-Cloud-Deployment.

4 Konzept – Plattform

In diesem Kapitel soll eine Service-Plattform als Basis und zentraler Teil dieser Arbeit konzipiert werden. Angefangen bei den Anforderungen an eine solche Plattform soll auf die einzelnen Bestandteile geschlossen werden, um diese anschließend im Detail zu beschreiben. Erweiterungen der grundlegenden Plattform werden anschließend in den folgenden zwei Kapiteln diskutiert.

4.1 Anforderungen

Im Folgenden sollen die Anforderungen solch einer Plattform ausgearbeitet werden. Hierbei wird zwischen funktionalen und nicht-funktionalen Anforderungen unterschieden.

4.1.1 Funktionale Anforderungen

Die funktionalen Anforderungen geben Aufschluss darüber, welchen Funktionsumfang eine solche Plattform haben muss, um das in der Zielsetzung beschriebene Szenario zu unterstützen.

Services

Als einer der Kernbestandteile sind die Services in vielen Anwendungsfällen von zentraler Bedeutung. Services müssen zum einen in einer Form entwickelt werden, die von einer Plattform interpretiert und ausgeführt werden kann. Darüber hinaus müssen Metadaten den Service beschreiben, um einfache Attribute wie den Namen oder Schnittstellen von diesem erfahren zu können. In einer Komposition tauschen Services untereinander Daten aus, was bedeutet, dass mehrere Services einheitliche Kommunikationsprotokolle verwenden müssen. Um die Kommunikationswege oder den Service allgemein zu konfigurieren, muss ebenfalls eine Möglichkeit vorhanden sein, Werte an diesen zu übergeben. Unabhängig von allen Teilanforderungen besteht dabei immer noch die Anforderung der Multi-Cloud-Fähigkeit. Als Cloud-Anwendung soll ein Service außerdem die IDEAL-Eigenschaften aus Abschnitt 2.1.1 aufweisen.

Workflows abbilden

Nutzer dieser Plattform sollen in der Lage sein, produktionstechnische und allgemeine Prozesse durch Verknüpfen einzelner Services abzubilden (vgl. Abbildung 4.1). Es soll deshalb mit Services möglich sein, Maschinendaten abzurufen und an andere Services weitergeben zu können. Kommunizieren mehrere Services miteinander oder mit einem zentralen Service, muss eine flexible Verbindung mit ihnen möglich sein.

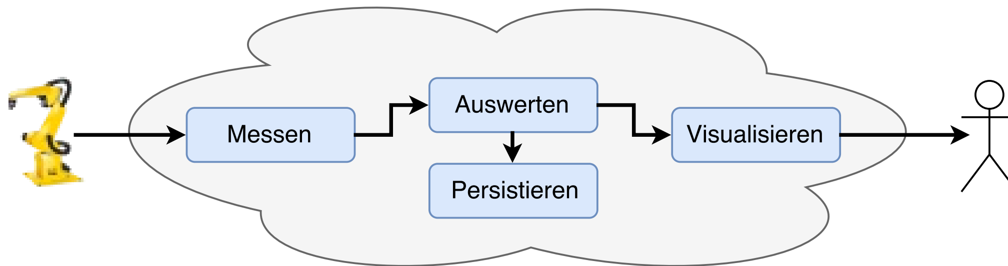


Abbildung 4.1: Ein exemplarischer Aufbau eines Workflows bei dem Daten von einer Maschine abgerufen, persistiert und Nutzern in einer Visualisierung angezeigt werden.

Marktplatz für Services

Services sollen in einem Marktplatz kategorisiert aufgelistet sein. Anwender sollen aus der Plattform heraus Services in diesem Marktplatz suchen und beziehen können. Hierbei müssen gegebenenfalls Lizenzmodelle angegeben werden können, die beschreiben, wie die gekauften Services zu nutzen sind. In dieser Arbeit wird jedoch angenommen, dass solch ein Marktplatz bereits existiert und deshalb nicht weiter im Konzept der Plattform berücksichtigt.

Dynamisches Deployment

Im einfachsten Szenario muss ein einzelner, unkonfigurierter Service in eine Cloud deployt werden. Optimalerweise läuft er dort, ohne auszufallen oder überlastet zu sein und ohne einen Ausfall der Cloud-Umgebung zu erfahren. Benötigt ein Service eine Konfiguration beim ersten Deployment, muss es vom Deployment-Prozess und der Cloud-Plattform unterstützt werden. Bei einer Konfiguration zur Laufzeit muss der Service dies zum einen unterstützen und zum anderen müssen nicht-funktionale Anforderungen wie eine hohe Verfügbarkeit weiterhin gewährleistet sein. Das Deployment soll außerdem in der Hinsicht flexibel sein, dass zur Laufzeit mit Ausfällen gerechnet werden muss und eine Skalierung je nach Auslastung möglich sein soll. Bei einem komplexeren Aufbau aus einer Komposition mit mehreren Services, die untereinander kommunizieren, gestaltet sich das Deployment komplizierter. Abhängigkeiten an die Startreihenfolge müssen eingehalten werden, da sonst ein erfolgreicher Start aller Services und somit auch das Deployment fehlschlagen könnte. Müssen einzelne deployte Services konfiguriert werden, soll die Kommunikation mit ihnen für diese Zeit nicht ausfallen.

Multi-Cloud

Services sollen in verschiedene Clouds deployt werden können, um etwaige Standort- oder Kostenvorteile nutzen zu können. Die Auswahl der Ziel-Cloud-Plattform muss prinzipiell für jeden Service möglich sein. Infrastruktur-Services, die bspw. für die Kommunikation benötigt werden, sollen ebenfalls im Multi-Cloud-Szenario berücksichtigt werden. In Service-Kompositionen sollen

einzelne Services auch auf verschiedene Clouds deployt werden können, wobei die Kommunikation hierbei wie in einem Szenario mit nur einer genutzten Cloud-Plattform ungestört möglich sein soll.

Benutzeroberfläche

Wie aus den vorigen Anforderungen bereits hervorgeht, soll ein Anwender die Möglichkeit haben, alle Prozesse zu steuern und zu beobachten. Hierfür muss eine Benutzeroberfläche konzipiert werden, die dem Anwender in verschiedenen Ansichten die Möglichkeit gibt, Services zu verwalten, Workflows aus mehreren zu konzipieren und diese zu deployen.

4.1.2 Nicht-funktionale Anforderungen

Außer den funktionalen Anforderungen existieren auch Anforderungen an den Betrieb einer Service-Plattform und etwaige Rahmenbedingungen.

Benutzbarkeit

Die Service-Plattform soll in vielerlei Hinsicht eine hohe Benutzbarkeit besitzen, um die technische Komplexität zu verstecken und Anwendern einen leichten Einstieg in die verwendeten Konzepte zu ermöglichen. Der Nutzer soll ohne große technische Vorkenntnisse in der Lage sein, die oben genannten Funktionen in einer Benutzeroberfläche zu nutzen, welche ihn hierbei führt. Er soll Services nutzen können, ohne Details über deren inneren Aufbau oder die Implementierung kennen zu müssen und soll in der Lage sein, diese mit anderen Services zu verknüpfen, ohne sich Gedanken über den Aufbau der Kommunikationswege zu machen. Das Deployment auf verschiedene Clouds soll möglich sein, ohne tiefere Kenntnisse zu den Eigenschaften der Cloud-Provider zu kennen.

Auf der anderen Seite sollen Entwickler einen leichten Einstieg in die Service-Entwicklung haben, indem bspw. bekannte Konzepte und Standards genutzt werden. Es soll ebenfalls möglich sein, vorhandene Anwendungen aus einem eigenen — eventuell schon automatisierten — Prozess mit wenig Aufwand für diese Service-Plattform aufzubereiten, sodass der Aufwand bei der Entwicklung verringert, die Auswahl an potenziellen Services vergrößert und somit der Umstieg erleichtert wird. Entwickler sollen auch in der Lage sein, möglichst ihnen bereits bekannte Technologien verwenden zu können, um die Hürden der Service-Entwicklung deutlich zu senken und eine hohe Erlernbarkeit zu ermöglichen. Aspekte wie evtl. verschiedene Laufzeitumgebungen der Services, der Aufbau von Kommunikationswegen oder allgemein die Sicherheit der beiden Aspekte, sollen bei der Entwicklung keine Rolle spielen.

Verfügbarkeit & Resilienz

Die Plattform sowie die deployten Services sollen eine hohe Verfügbarkeit aufweisen, um zuverlässig produktionstechnische Prozesse steuern und verwalten zu können. Bei Ausfällen von einzelnen Teilen der Plattform sollen die restlichen Teile möglichst unbeeinflusst bleiben und soweit möglich automatisch versuchen, das aufgetretene Problem zu beheben. Bei einer zu hohen

Auslastung soll die Verfügbarkeit von Anwendungen weiterhin hoch bleiben, indem Services beispielsweise automatisch skalieren können.

Plattformunabhängigkeit

Alle Teile der Plattform sollen möglichst unabhängig von einer speziellen Laufzeitumgebung funktionieren, um auch konsequent zum vermiedenen Vendor-Lock-in bei Cloud-Anbietern zu bleiben. So sollen sowohl Plattform als auch Services und die verwendeten Technologien plattformunabhängig sein.

Wartbarkeit

Die Wartbarkeit der Plattform und der Services soll unter anderem durch Verwenden von Standards und bekannten Technologien auf einem hohen Niveau gehalten werden, da diese oft gut dokumentiert sind und Support bspw. von einer Community um diese Technologien herum gegeben werden kann. Es darf außerdem nicht außer Acht gelassen werden, dass bereits deployte Service-Kompositionen gewartet werden können, während andere Anforderungen wie die Verfügbarkeit bestmöglich eingehalten werden.

4.1.3 Stakeholder

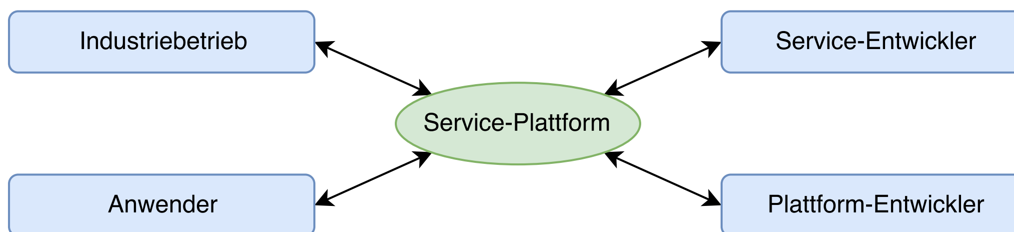


Abbildung 4.2: Eine Übersicht über die Stakeholder der Service-Plattform.

Effektiv sind vier Stakeholder an der Service-Plattform beteiligt: Service-Entwickler, Anwender, Industriebetriebe und die Plattformentwickler. Die Service-Entwickler erstellen die Grundlage, mit der die Plattform erst ihre eigentliche Funktion ausführen kann. Diese Services können anschließend von den Nutzer miteinander verknüpft, konfiguriert und deployt werden. Sie haben Kenntnis davon, was für einen Zweck die Services erfüllen und setzen sie zu ihrem Nutzen ein. Eine Differenzierung zwischen administrativen Nutzern, die die Konfiguration der Plattform für bspw. die Zugangsdaten der Cloud-Plattformen vornehmen, soll hierbei nicht stattfinden. Der Industriebetrieb ist insofern beteiligt, dass er die Plattform mit seinen Maschinen einsetzen will, um durch sie einen Mehrwert zu erhalten. Er hat Interesse daran, dass bspw. die Datenschutzrichtlinien durch Nutzen einer bestimmten Cloud-Plattform nicht verletzt werden und ein Mehrwert aus der Bildung eines Workflows in der Cloud auf Basis der Industriemaschinen stattfindet. Als letzte wesentlich beteiligte Nutzergruppe sind die Entwickler dieser Plattform zu nennen.

4.2 Services

Das detaillierte Konzept der Services kann in der zu Beginn erwähnten Partnerarbeit gefunden werden, weshalb an dieser Stelle nur eine Zusammenfassung folgt. Services sollen nach Anforderung eine isolierte Laufzeitumgebung besitzen und mit verschiedenen Technologien umgesetzt werden können, weshalb nur eine Kapselung in einer virtuellen Maschine oder Containern infrage kommt. Aufgrund der Vorteile gegenüber virtuellen Maschinen bspw. hinsichtlich des Ressourcenbedarfs oder des Toolings kommen hierfür nur Container in Frage. Hierbei wird Docker aufgrund der guten Dokumentation, der weiten Verbreitung und der Werkzeuge um Docker herum gewählt.

Durch die Docker-Container wird grundsätzlich vorausgesetzt, dass jeder Service in einer Linux-Umgebung lauffähig ist, welche allerdings in hohem Maße vom Service-Entwickler angepasst werden kann. Services in Containern können durch Docker-eigene Mittel über Umgebungsvariablen konfiguriert werden, was auch bei nicht-containerisierten Anwendungen eine übliche Konfigurationsmöglichkeit ist. Sollen Umgebungsvariablen eines Docker-Containers während dessen Betrieb geändert werden, ist es allerdings erforderlich, dass der betroffene Service wiederholt deployt wird (vgl. Abschnitt 4.5). Bei der Erstellung eines Docker-Images können mit dem sogenannten `HEALTHCHECK`-Befehl der Docker-Engine Anweisungen gegeben werden, mit der regelmäßig der Zustand der Anwendung im Container geprüft wird. Hiermit kann beispielsweise registriert werden, ob die Anwendung im Container erfolgreich gestartet ist oder ob sie aufgrund eines Fehlers oder wegen zu hoher Last nicht mehr reagiert, woraufhin weiterführende Maßnahmen ergriffen werden können.

Da die Services über Umgebungsvariablen konfiguriert werden, sollen über diesen Weg auch Informationen zur Kommunikation an den Service gegeben werden, wie bspw. die Quelle von eingehenden Nachrichten. Wie zuvor beschrieben, sollen sich Service-Entwickler nicht mit den Details der Service-Kommunikation beschäftigen, sondern sich auf die reine Service-Entwicklung konzentrieren können. Da es trotzdem notwendig ist, dass die Service-Entwickler Nachrichten senden und empfangen können, müssen sie sich zumindest indirekt mit den Kommunikationstechnologien befassen. Als Erleichterung hierfür ist das Konzept eines sogenannten *Service-Core* vorgesehen. Der Service-Core soll als Programmbibliothek für verschiedene Programmiersprachen existieren, sodass diese von den Entwicklern nativ aufgerufen werden kann. Er soll die Abfrage von Konfigurationswerten und die Schnittstelle zu den verwendeten Kommunikationstechnologien abstrahieren und somit die Entwicklung dagegen deutlich vereinfachen.

Jeder Service soll in einem Spezifikationsdokument mitsamt seinen Metadaten und Schnittstellen beschrieben werden. Die Service-Definition mit WSDL oder TOSCA ist hierbei nicht wirklich praktikabel, da entweder Konfiguration oder öffentliche Service-Schnittstellen nicht angegeben werden können. In einer dynamischen Service-Plattform soll ein Service außerdem nicht wie mit WSDL anderen Services Kommunikationsmethoden vorschreiben, da dies die Flexibilität beim Verknüpfen einschränken würde. Zwar geben die sogenannten Plans bei TOSCA einen generischen Ansatz, um Services zu starten und zu konfigurieren, welcher allerdings zusammen mit der Topologie im Fall einer containerisierten Lösung nicht notwendig ist. In einer neuen Service-Definition sollen deshalb die praktikablen Konzepte der YAML-Spezifikation von TOSCA [OAS16] aufgegriffen werden, zu denen bspw. `Properties` zur Konfiguration oder die Angabe von Metadaten gehören. Da dieses Dokument voraussichtlich ausschließlich manuell angelegt und bearbeitet wird, wird hierfür keine komplexe und wortreiche Markup-Sprache wie XML gewählt, sondern YAML.

Listing 4.1 Definition eines einfachen Services zur Grenzwertüberwachung mit YAML.

```
GrenzwertService:
  Metadata:
    Title: "Grenzwert-Überwachungs-Service"
    Description: "Meldet das Überschreiten eines konfigurierbaren Grenzwertes."
    Author: "Vincent Link <mail@linkvt.de>"
  Image:
    Name: "plattform-services/grenzwert-service"
    Tag: "1.0.0"
  Properties:
    MinimalerGrenzwert:
      Name: Minimaler Grenzwert
      Description: Der minimal zulässige Grenzwert.
      DefaultValue: 19
      Type: int
  ...
  Input:
    String:
      Id: "messwert"
      Schema: '"int"'
  Output:
    Boolean:
      Id: "gueltig"
      Schema: '"boolean"'
```

Ein Beispiel anhand eines Services zur Grenzwertüberwachung ist in Listing 4.1 abgebildet. Unter dem `Metadata`-Wert untergeordnet können Informationen, wie der Name, Autor oder eine kurze Beschreibung, gefunden werden. Anschließend wird die Quelle des Services mit dem Namen des Docker-Images, einer Version und eventuell auch einer Docker-Registry angegeben, sodass die Plattform damit den Service herunterladen und deployen kann. Die Plattform benötigt nun noch weitere Informationen zu den möglichen Konfigurationswerten, wie bspw. den minimal und maximal zulässigen Grenzwerten sowie möglichen Schnittstellen des Services. In diesem Beispiel nimmt der Service einen Messwert vom Typen `int` entgegen und kann einen booleschen Wert ausgeben, der die Gültigkeit dieses Messwerts beschreibt. Anstelle der einfachen Typen `int` oder `boolean` können auch komplexe Schemadefinitionen angegeben werden (vgl. Abschnitt 4.3.5).

4.3 Service-Kommunikation

Die Kommunikation zwischen Services innerhalb einer Komposition oder auch mit externen Services ist essentiell, um einen Mehrwert aus der Kombination von verschiedenen Services und dem Abbilden von Workflows zu erhalten. Zu Beginn dieses Abschnitts soll offen gelassen werden, wie genau die Kommunikation stattfinden kann. So könnten direkt typisierte Funktionsaufrufe mit Techniken wie Remote-Procedure-Calls (RPC) oder auch das Senden von generischen Inhalten an festgelegte Adressen wie bei einer Web-Schnittstelle nach dem REST-Paradigma genutzt werden. Zum Zweck der Vereinfachung werden die Begriffe „Nachricht“ für auszutauschende Inhalte, „Sender“ für Services mit ausgehenden Nachrichten und „Empfänger“ für Services mit eingehenden Nachrichten verwendet. Grundsätzlich soll nicht der Service bestimmen, mit wem er kommuniziert, sondern der Plattformnutzer, da er das Wissen über den aufgebauten Workflow und somit die Kommunikationsstrukturen besitzt.

4.3.1 Kopplung zwischen Services

Die Kopplung zwischen den Services soll innerhalb einer Komposition so gering wie möglich gehalten werden, um die Auswirkungen von Ausfällen zu verringern und die Austauschbarkeit zu erhöhen. Für die direkte Kommunikation wäre es notwendig, dass Nachrichten an den Empfänger adressiert werden können. Direkt bedeutet, dass der Sender die Identität des Empfängers kennen muss und somit ein tief gehendes Wissen über die gesamte Topologie der Service-Komposition haben muss — und das bereits zum Start des Services. Werden zwei miteinander kommunizierende Services gestartet, kann ihre Konfiguration aufgrund von Einschränkungen durch die Docker-Engine bis zum nächsten Deployment nicht mehr geändert werden, wodurch sie bereits alle zur Kommunikation notwendigen Informationen besitzen müssen. Soll nun einer der beiden Services ausgetauscht werden, soll der andere Service hiervon unbeeinflusst weiter laufen können. Um den für die Kommunikation notwendigen Aufwand im Service weiter zu reduzieren, soll der Aufbau der Verbindung durch die Plattform durchgeführt werden, sodass diese bspw. einen Nachrichtenkanal zwischen den Services aufbaut und der Overhead hierfür in den Services entfällt.

Interaktionsmodell Ein synchroner Nachrichtenaustausch (vgl. Abschnitt 2.3.1) würde eine höhere Kopplung zwischen den kommunizierenden Services bedeuten. Der Sender einer Nachricht wartet blockiert so lange, bis die Antwort des Empfängers eintrift. Im Szenario mit mehreren Services deployt auf verschiedenen Cloud-Providern — potentiell weltweit verteilt — muss mit Ausfällen der Services und der Kommunikation gerechnet werden. Da weiterhin nach Ausfällen oder bei großem Lastaufkommen ein einzelner synchron kommunizierender Service leicht überlastet werden könnte, soll durch einen asynchronen Austausch der Nachrichten eine Entkopplung zwischen den Services erreicht und die Möglichkeit geschaffen werden, durch eine bessere Skalierbarkeit (vgl. Kapitel 6) den Durchsatz zu erhöhen.

4.3.2 Szenarien

Die Kommunikation zwischen Services kann hierbei unabhängig von den verwendeten Technologien oder Konzepten strukturell in wenigen möglichen Szenarien stattfinden, die im Folgenden beschrieben und für die anschließende Anforderungsdefinition genutzt werden.

Senden an einen Service

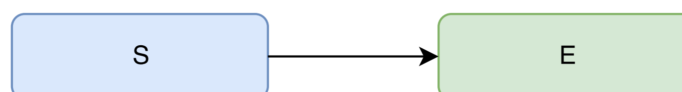


Abbildung 4.3: Die Kommunikation zwischen zwei Services im einfachsten Szenario: Ein Sender schickt Nachrichten an einen einzigen Empfänger.

Im einfachsten Szenario schickt ein Service alle Nachrichten an genau einen anderen Service (vgl. Abbildung 4.3). Hierbei soll nicht ausgeschlossen werden, dass beide Services jeweils Sender als auch Empfänger sein können und die Kommunikation zwischen ihnen bidirektional abläuft.

Damit die Kommunikation zwischen beiden Services stattfinden kann, müssen gewisse Voraussetzungen gegeben sein. Ein einheitliches Kommunikationsprotokoll wird durch die Abstraktion im Service-Core bereits sichergestellt, sodass technisch jeder Service mit jedem anderen Service kommunizieren kann. Eine explizite Adressierung an einen Empfänger ist überflüssig, da der Service nur die vorhandene Verbindung nutzen muss. Soll zwischen verschiedenen Nachrichtentypen unterschieden werden, muss beim Senden dieser Typ adressiert sein, sodass der Empfänger ebenfalls die eingehenden Nachrichten unterscheiden kann.

Senden an mehrere Services

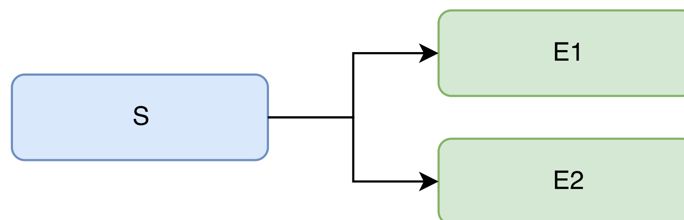


Abbildung 4.4: Senden von allen Nachrichten an mehrere empfangende Services.

Werden alle Nachrichten gleichen oder unterschiedlichen Typs an mehrere Empfänger gesendet, ändert sich für den Sender im Vergleich zum einfachsten Szenario nichts. Er nutzt weiterhin den von der Plattform aufgebauten Kanal zwischen den Services und sendet alle Nachrichten ohne besonderen Adressaten. Der Aufbau der Verbindung wird auch hier von der Plattform vorgenommen.

Da allerdings nicht immer alle Services alle Nachrichtentypen verstehen können und nicht ihre Ressourcen mit irrelevanten eingehenden Nachrichten aufbrauchen sollen, ist es notwendig, dass auf Plattformebene ein komplexeres Kommunikationsmodell aufgebaut werden kann. Wie im in Abbildung 4.5 dargestellten Aufbau, soll ein Nachrichtentyp (gelb) vom Sender *S* an beide Empfänger *E1* und *E2* gesendet werden, während ein anderer Nachrichtentyp (rot) nur an den Empfänger *E1* gesendet wird. Die Plattform muss hierfür mehrere Kanäle aufbauen können, die zwar vom selben Sender ausgehen, allerdings in unterschiedlichen Empfängergruppen enden. Dem Sender wird hiermit allgemein die Aufgabe genommen, selbst entscheiden zu müssen, an welche Empfänger die Nachrichten gesendet werden. Eine Auswahl von Empfängern innerhalb eines Services beim Versand von Nachrichten des gleichen Typs soll explizit nicht möglich sein, da dieser die Kommunikationsstrukturen nicht kennt.

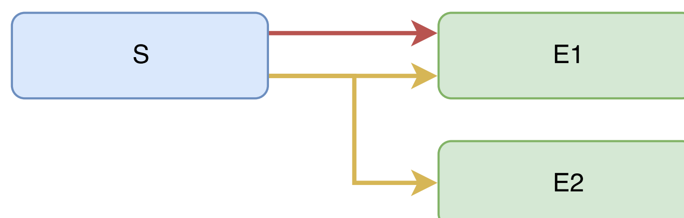


Abbildung 4.5: Senden von Nachrichten unterschiedlichen Typs an verschiedene Services.

Empfangen von mehreren Services

Grundsätzlich soll es möglich sein, dass nicht nur Nachrichten an mehrere Services versendet werden können, sondern auch, dass ein Service Nachrichten mehrerer Sender empfangen kann (vgl. Abbildung 4.6). Wie beim Senden soll hier umgekehrt ein empfangender Service den Ursprung der Nachrichten nicht kennen müssen. Die Plattform muss hierfür mehrere Kanäle in einem Kanal bündeln können, sodass dem Empfänger die tatsächliche Anzahl der Empfangskanäle nicht bekannt sein muss.

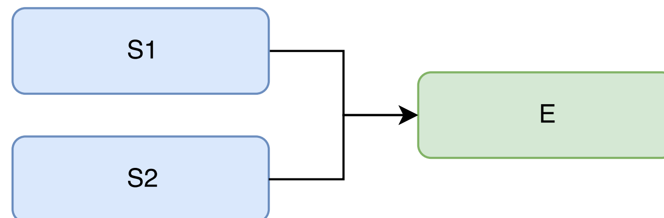


Abbildung 4.6: Empfangen von Nachrichten mehrerer Services in einem einzelnen Service.

4.3.3 Zuverlässigkeit

Die Kommunikation zwischen Services muss als Kernbestandteil der Service-Orchestrierung eine sehr hohe Zuverlässigkeit vorweisen. Services sollen sich darauf verlassen können, dass gesendete Nachrichten ankommen — auch wenn einzelne Teile der Infrastruktur oder einzelne Empfänger kurzzeitig nicht verfügbar sind. Aufgrund von Problemen beim Nachrichtenversand kann es vorkommen, dass einzelne Nachrichten verloren werden oder sie durch einen erneuten Versand bei falscher Annahme eines Verlusts mehrfach gesendet werden. Dass fehlende Nachrichten fatale Auswirkungen haben können, steht außer Frage. Jedoch können auch realistische Szenarien konstruiert werden, die bei duplizierten Nachrichten fatal enden, wie im folgenden Beispiel in Abbildung 4.7 eines Steuerungsservices und einer Werkzeugmaschine gezeigt werden soll.

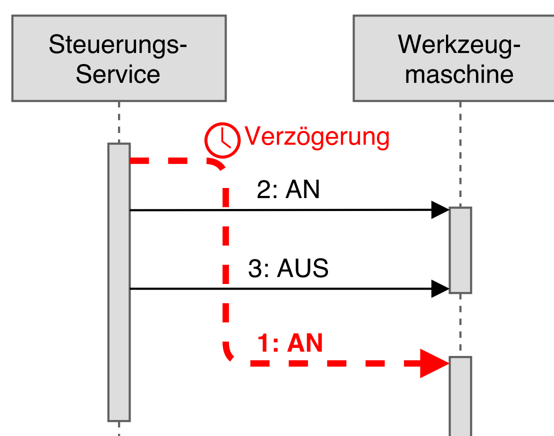


Abbildung 4.7: Nachrichten werden aufgrund einer falschen Annahme (Verlust der Nachricht) mehrfach versendet und lösen ein Ergebnis aus, das nicht beabsichtigt wurde.

Ein Steuerungs-Service sendet Nachricht 1 zum Anschalten einer Werkzeugmaschine, welche allerdings scheinbar verloren geht. Der Versand wird deshalb mit Nachricht 2 wiederholt und die Maschine effektiv angeschaltet. Nach einer bestimmten Zeit wird die Werkzeugmaschine mit Nachricht 3 wieder ausgeschaltet. Mit einer großen Verzögerung wird Nachricht 1 nun doch noch an die Werkzeugmaschine zugestellt und schaltet sie in einem ungünstigen Zeitpunkt wieder an, sodass es zu Sach- oder sogar Personenschäden kommt.

Aus diesem Grund ist es notwendig, dass Nachrichten nicht nur mindestens ein Mal beim Empfänger, sondern auch nach dem *Exactly-Once-Pattern* höchstens ein Mal eintreffen sollen. Es ist ebenfalls von zentraler Bedeutung, dass die gesendeten Nachrichten nicht unterwegs verfälscht werden, sodass der Empfänger nicht mehr die ursprünglich gesendete Nachricht erhalten würde. Sollte ein Service Nachrichten doch nicht erfolgreich versenden können, soll dies dem Service mitgeteilt werden, damit dieser ggf. weitere Maßnahmen einleiten kann.

Als weitere Anforderung an den Betrieb der Kommunikationsschicht steht die Ausfallsicherheit und allgemein die Verfügbarkeit mit an oberster Stelle. Die Kommunikation soll möglichst ausfallsicher und redundant betrieben werden, da mit Ausfällen jederzeit gerechnet werden muss. Bei Engpässen aufgrund von Lastspitzen soll die Zuverlässigkeit des Versands weiterhin eingehalten werden, wie in Kapitel 6 weiter ausgeführt wird.

4.3.4 Datendurchsatz

Komplexe Industriemaschinen besitzen teilweise tausende Aktoren und Sensoren, die ggf. mit einer hohen Frequenz eine gigantische Menge an Daten liefern. Wenn diese Daten an Services gegeben und zwischen Services weitergereicht werden, müssen die für die Kommunikation verwendeten Technologien auch hierfür ausgelegt sein. Da die Plattform nicht für spezielle Szenarien, sondern flexibel ausgelegt ist, existieren keine harten Limits an den Durchsatz. Faktoren wie die für die Zuverlässigkeit geforderte *Exactly-Once-Lieferung* der Nachrichten stehen mit einem maximalen Datendurchsatz gewissermaßen in Konflikt, da von der Kommunikationsschicht hierfür ein zusätzlich aufzubringender Aufwand verlangt wird. Harte Anforderungen an die Latenz des Nachrichtenversands stehen ebenfalls mit einer in der Cloud verteilten Service-Komposition in Konflikt, da sie stark von der zugrundeliegenden Infrastruktur abhängt, die im PaaS-Modell nicht direkt beeinflusst werden kann. Eventuell vorhandene verlustfreie Kompressionsverfahren können den Durchsatz durch eine Reduzierung der Datengröße erhöhen.

4.3.5 Typsicherheit

Bereits vor der Verknüpfung von Services wird durch die Service-Spezifikation (vgl. Listing 4.1) angegeben, welche Art von Nachrichten ein Service senden und empfangen kann, und somit die Schnittstelle des Services definiert. Prinzipiell wäre es möglich, dass Services beliebige Nachrichten senden bzw. empfangen können und der Nachrichtentyp beiden Services implizit bekannt ist. Hiermit würde die größtmögliche Flexibilität erreicht werden, da keine aufwendigen Schemas definiert werden und beliebige Ein- und Ausgänge miteinander verbunden werden können.

Dagegen spricht jedoch, dass es auch möglich sein soll, Services verschiedener Service-Entwickler miteinander zu kombinieren, sodass ein erheblicher Kommunikationsaufwand notwendig wäre, um impliziten Schemas anderen Entwicklern mitzuteilen. Der Nutzer der Plattform soll außerdem

beim Anlegen von Workflows mit der Plattform auch mit diesen Typinformationen unterstützt werden, um evtl. fatale Folgen aufgrund von Verwechslungen zu vermeiden — bspw. wenn ein Service Temperaturinformationen in Fahrenheit sendet, während ein anderer Celsius-Werte erwartet. Diese Typsicherheit soll auch beim Versand und Empfang von Nachrichten sichergestellt sein. So sollen Services Nachrichten nur mit Typangabe an einen anderen Service senden können, wobei der Versand zuvor von der Plattform auf diesen speziellen Typen eingeschränkt wurde, um die Möglichkeit von Fehlern weiter zu reduzieren.

4.3.6 Vergleich

Die oben definierten Anforderungen suggerieren stark eine bestimmte Methode für die Service-Kommunikation. Mit einem starken Fokus auf der Entkopplung einzelner Services für komplexe Sende- und Empfangsszenarien sowie die Ausfallsicherheit und Zuverlässigkeit, zeigt sich Messaging über eine Middleware als klarer Favorit. Mit dem REST-Pattern würde jede Nachricht alle notwendigen Zustandsinformationen enthalten. Somit würden Service-Ausfälle weniger Auswirkungen haben, da sie keine Zustandsinformationen zur Laufzeit benötigen, die für die Weiterverarbeitung der empfangenen Nachrichten notwendig sein könnte. Auch RPC und SOAP würden durch eine starke Typisierung der verwendeten Nachrichten eine bedeutende Anforderung erfüllen. Die enge Kopplung bei einer Kommunikation über RPC, SOAP und REST erschwert allerdings einen dynamischen Aufbau der Kommunikationsschicht, die einen flexiblen Austausch von Services ermöglichen soll.

Mit einer Message-oriented-Middleware können die Vorteile der asynchronen Kommunikation innerhalb der verteilt arbeitenden Service-Komposition voll ausgeschöpft werden. Nachrichten werden bei Ausfällen einzelner Services zugestellt, wenn diese wieder erreichbar sind, und verursachen damit keine unerwarteten Wartezeiten wie bei blockiert-sendenden Services mit RPC oder SOAP. Durch eine Entkopplung der Kommunikation von den Services selbst wird hierdurch eine Skalierbarkeit der Kommunikationsschicht ermöglicht, um große Datenaufkommen besser bewältigen zu können. Sollen Nachrichten an mehrere Empfänger gesendet werden, wird hierfür bei den weniger präferierten Technologien eine Konfiguration im Service benötigt, da dort jeder Empfänger explizit angegeben werden muss.

Nach einer Evaluation verschiedener Tests vorhandener Middlewares wie in [Nan15] auch bezüglich deren Performance[1], der Typsicherheit[2] oder der geforderten Exactly-Once-Delivery[3], hat sich Apache Kafka auch aufgrund seiner Familiarität als Kommunikations-Middleware durchgesetzt. Der typsichere Nachrichtenversand von RPC und SOAP ist zwar nicht standardmäßig in MOMs vorhanden, kann aber durch den Einsatz von zusätzlichen Technologien wie Apache Avro[4] als Serialisierungstechnologie ergänzt werden.

[1] Apache Kafka vs RabbitMQ, http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/
[2] Apache Kafka und Apache Avro, https://www.confluent.io/blog/avro-kafka-data/
[3] Apache Kafka – Exactly-Once-Delivery, https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/
[4] Apache Avro, https://avro.apache.org/

4.4 Back-End

An zentraler Stelle zur automatisierten Steuerung und Überwachung der oben geforderten Prozesse wird hierfür eine im Hintergrund arbeitende Server-Komponente (Back-End) benötigt, die vom Nutzer über eine Benutzeroberfläche (Front-End) angesprochen werden kann. Hierbei soll auch berücksichtigt werden, dass die IDEAL-Eigenschaften von Cloud-Anwendungen möglichst gut umgesetzt werden.

Eine einzelne Anwendung, die diese Funktionalität und sowohl die Benutzeroberfläche implementiert, könnte einfach bei einem Anwender auf einem lokalen Computer genutzt werden. Um allerdings auch mehrere Anwender die Plattform nutzen zu lassen, muss sie zentral auf (mindestens) einem Server betrieben werden. Durch eine Entkopplung von Back-End und Benutzeroberfläche kann weiterhin eine bessere Skalierbarkeit der Anwendung erreicht werden, da bspw. bei einer großen Auslastung des Front-Ends aufgrund von vielen lesenden Nutzern nicht auch zusätzlich eine weiteres Back-End betrieben werden muss.

Die einfachste Aufgabe, die das Back-End übernehmen muss, ist die Stammdatenverwaltung der gehaltenen Daten. Der Nutzer muss in der Lage sein, beispielsweise die Zugangsdaten der verwendeten Cloud-Plattformen und eventuelle Rechnersysteme in der Service-Plattform zu hinterlegen und dauerhaft abzuspeichern. Diese Daten können zusammen mit verwalteten Services in Service-Kompositionen und Deployments im Back-End abgelegt werden. Weiterhin gehören auch Nutzer der Plattform oder allgemeine Konfigurationen zu den Stammdaten.

Der Anwender soll aus diesen Daten Services in einer Service-Komposition verknüpfen, um sie anschließend auf bestimmte Cloud-Plattformen deployen zu können. Um dem Nutzer in der Oberfläche die technische Konfiguration einer Orchestrierung zu ersparen und ihm stattdessen eine nutzerfreundliche Abstraktion anzubieten, müssen im Front-End und Back-End unterschiedliche Objektmodelle genutzt werden. Deshalb muss das Back-End die Komposition mit Hinblick auf die containerisierten Services, die Kommunikation zwischen ihnen und deren Konfiguration umwandeln. Im daraufhin angesteuerten Deployment-Prozess, der im folgenden Abschnitt erläutert wird, sollen diese Bestandteile Stück für Stück in den produktiven Betrieb eingespielt werden. Anschließend muss ein laufendes Monitoring der deployten Bausteine der Komposition den Status überwachen und gegebenenfalls weitere Maßnahmen ergreifen, während dem Nutzer diese Informationen transparent über die Schnittstelle zur Benutzeroberfläche gezeigt werden.

4.5 Deployment

Die vom Anwender in der Oberfläche konfigurierten Service-Kompositionen müssen anschließend in einer oder mehreren Clouds in Betrieb genommen werden. Als Teilprozess des Back-Ends werden beim Deployment die Container und die Messaging-Konfiguration für deren Betrieb in eine oder mehrere Cloud-Umgebungen deployt. Als Grundlage für die Container der Services werden hierbei die Ziel-Cloud-Plattform, Informationen zum verwendeten Docker-Image und eventuell die Konfiguration für eine Anbindung an die Messaging-Plattform benötigt.

4.5.1 Infrastruktur-Services

Obwohl die Konfiguration der Message-oriented-Middleware nicht vom Nutzer, sondern von der Service-Plattform selbst vorgenommen wird, muss der Anwender in der Lage sein, die Middleware in seiner Orchestrierung zu verwenden. Der Nutzer könnte die Anforderung haben, dass mehrere Middlewares für eine Service-Komposition genutzt werden können. Dieser Anforderung soll allerdings nicht weiter nachgegangen werden, da eine Komposition als kleinste Einheit zusammengehöriger Services betrachtet werden soll. Im Gegensatz dazu bringt eine von allen Kompositionen genutzte Middleware erhebliche Nachteile mit sich. Eine hohe Auslastung einzelner Kompositionen könnte sich auf den Nachrichtenversand anderer Kompositionen auswirken. Weiterhin wären trotz Multi-Cloud-Szenario alle Services dazu verpflichtet, die auf einer Cloud laufende Middleware zu nutzen.

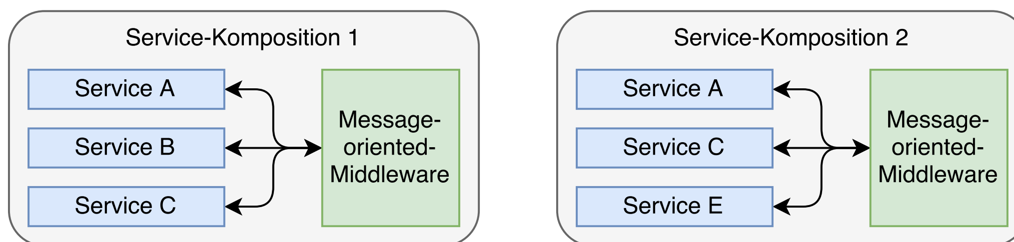


Abbildung 4.8: Die Message-oriented-Middleware wird immer nur von Services einer Komposition genutzt.

Daher soll im optimalen Fall pro Deployment — also pro Service-Komposition — eine Messaging-Middleware-Instanz deployt werden (vgl. Abbildung 4.8), sodass auch keine Konflikte entstehen, wenn mehrere Service-Kompositionen eine MOM nutzen. Wenn Ressourcen gespart oder die Daten aus mehreren Service-Kompositionen geteilt genutzt werden sollen, kann es für eine Erweiterung der Plattform sinnvoll sein, wenn mehrere Deployments eine Instanz der MOM nutzen können.

4.5.2 Zielplattformen

Grundsätzlich soll das Deployment der Service-Komposition nicht vom Deployment-Modell der verwendeten Cloud-Plattformen abhängen, also ob eine Private-Cloud oder eine Public-Cloud angesprochen wird. Nach dem Multi-Cloud-Konzept sollen auf jeden Fall die großen Cloud-Provider wie Google, Amazon oder Microsoft unterstützt werden, um deren breite Palette an gebotener Funktionalität nutzen zu können. Es soll allerdings nicht ausgeschlossen werden, dass bspw. eine private Open-Stack-Cloud als Zielplattform genutzt werden kann. In manchen Szenarien dürfen so bspw. aus datenschutzrechtlichen Gründen die Daten nicht außerhalb der firmeneigenen Netzwerke gelangen, während dennoch eine firmeninterne Cloud genutzt werden soll.

Als große Herausforderung bei einer Vielfalt unterschiedlicher Cloud-Plattformen zeigt sich die Provisionierung der vom Anbieter bereitgestellten Umgebungen. Hierbei muss eine für die Service-Plattform nutzbare Umgebung eingerichtet werden, die bspw. die Docker-Laufzeitumgebung zur Verfügung stellt oder Port-Freigaben für die Kommunikation zwischen Services unterschiedlicher

Plattformen eingerichtet hat. Im besten Fall kann eine bereits vorprovisionierte Umgebung angesprochen werden, die bspw. mit allen notwendigen Portfreigaben auf Standardports zusammen mit Docker konfiguriert ist und direkt von der Plattform genutzt werden kann. Die Provisionierung kann vor einer Anbindung an die Service-Plattform auch manuell stattgefunden haben. Im für den Nutzer einfachsten Fall übernimmt die Anwendung die Provisionierung der Cloud-Umgebung, indem bspw. generische Skripte für Linux-Umgebungen automatisiert über Schnittstellen zur Umgebung ausgeführt werden.

4.5.3 Deployment-Reihenfolge

Als Grundlage für den Betrieb aller Services muss die Kommunikationsebene funktionsfähig sein, um direkt nach dem ersten Start eines Services Nachrichten annehmen zu können. Abhängigkeiten unter Services können beim Deployment-Prozess nur indirekt berücksichtigt werden, da ein abgeschlossenes Deployment eines Services nicht bedeutet, dass dieser lauffähig ist, sondern nur eine grobe Indikation gibt. Über die von Docker bereitgestellte Möglichkeit den Zustand der Anwendung mit Hilfe des `HEALTHCHECK`-Befehls prüfen zu lassen, könnten prinzipiell der abgeschlossene Start eines deployten Services festgestellt werden. Wird allerdings die Message-oriented-Middleware vor den Services deployt, kann diese direkt die Nachrichten einzelner Services annehmen und an die Empfänger weiterleiten, sobald diese von der MOM erreicht werden. Hierdurch ergibt sich die einzige Abhängigkeit jedes Deployments an ein abgeschlossenes MOM-Deployment und deren Start, unabhängig von der letztendlich deployten Service-Konstellation. Sollen einzelne deployte Services aktualisiert werden, kann die Middleware während des erneuten Deployments alle aufkommenden Nachrichten zwischenspeichern, sodass sie vom aktualisierten Service anschließend empfangen werden können.

4.5.4 Monitoring

Im Deployment-Prozess und darüber hinaus wird ein Monitoring des Prozesses und der deployten Services mitsamt der Infrastrukturservices benötigt. Während des Deployments muss für die Service-Plattform feststellbar sein, wie der Zustand einzelner deployter Container ist, um dem Nutzer den Abschluss des Deployments signalisieren zu können. Die Plattform benötigt diese Informationen ebenfalls auch später, um bei Ausfällen von Cloud-Umgebungen die Services zu einer anderen Umgebung umziehen zu können. Werden Änderungen am Deployment vorgenommen, so muss die Plattform abgleichen können, inwiefern diese Änderungen in das aktive Deployment eingespielt werden müssen und ob der Vorgang anschließend erfolgreich war. Im einfachsten Fall ist hierbei der reine Status des Services ausreichend; bei komplexeren Anforderungen je nach den verwendeten Methoden zur Gewährleistung einer Ausfallsicherheit und einer automatischen Skalierbarkeit (vgl. Kapitel 6) werden eventuell mehr Daten benötigt.

4.5.5 Constraints

Die Bereitstellung einer Multi-Cloud-Zielumgebung für die einzelnen Services ist nur mit der Einschränkung sinnvoll, dass kontrolliert werden kann, welche Cloud-Provider für welchen Service genutzt werden. So soll es im einfachsten Fall möglich sein, einzelne Services oder die gesamte Komposition direkt in die Umgebung eines spezifischen Cloud-Anbieters deployen zu

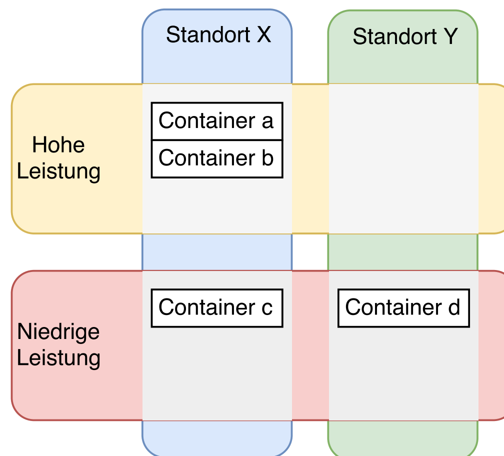


Abbildung 4.9: Die Kombinationen von zwei Constraints auf den Standort und die Leistung der Cloud-Umgebung.

können. Da das Multi-Cloud-Szenario vor allem mit Hinblick auf die Kosten für eine bestimmte Leistung oder einen bestimmten Standort gewählt wird (vgl. Abbildung 4.9), könnten ähnliche Plattformen in Gruppen zusammengefasst werden. Hiermit kann statt einem expliziten Anbieter eine Gruppe von Anbietern als Zielplattform gewählt haben, die bspw. den Standort *Deutschland* oder eine besonders gute Netzwerkanbindung besitzen.

4.5.6 Tools

Wie in Abschnitt 3.5 zusammengefasst wurde, bieten praktisch alle Management-Plattformen ausgereifte Orchestrierungsmöglichkeiten für Service-Kompositionen. Sie unterstützen mehrere Cloud-Anbieter, die Auswahl bestimmter Zielplattformen durch Constraints und bieten teilweise Mittel an, mit denen die Hosts und Services überwacht werden können. Apache Mesos bringt zusammen mit Apache Marathon als verteilter Systemkernel zwar eine sehr mächtige Plattform mit sich, die allerdings auch eine erhebliche Komplexität mit sich zieht. Im Vergleich dazu sind die anderen drei Plattformen explizit für den Zweck der Verwaltung von Containern ausgelegt und sind deshalb weniger generisch aufgebaut als Mesos. Rancher unterstützt allerdings als einzige Plattform direkte Integration von Ressourcen mehrerer Cloud-Anbieter und macht somit die Anforderung nach separaten Werkzeugen zur Provisionierung überflüssig. So kann Rancher für alle mit dem Deployment zusammenhängenden Prozesse verwendet werden.

4.6 Benutzeroberfläche

Die gesamte bisher konzipierte Funktionalität einer solchen Service-Plattform muss vom Anwender über eine passende Benutzeroberfläche steuerbar sein.

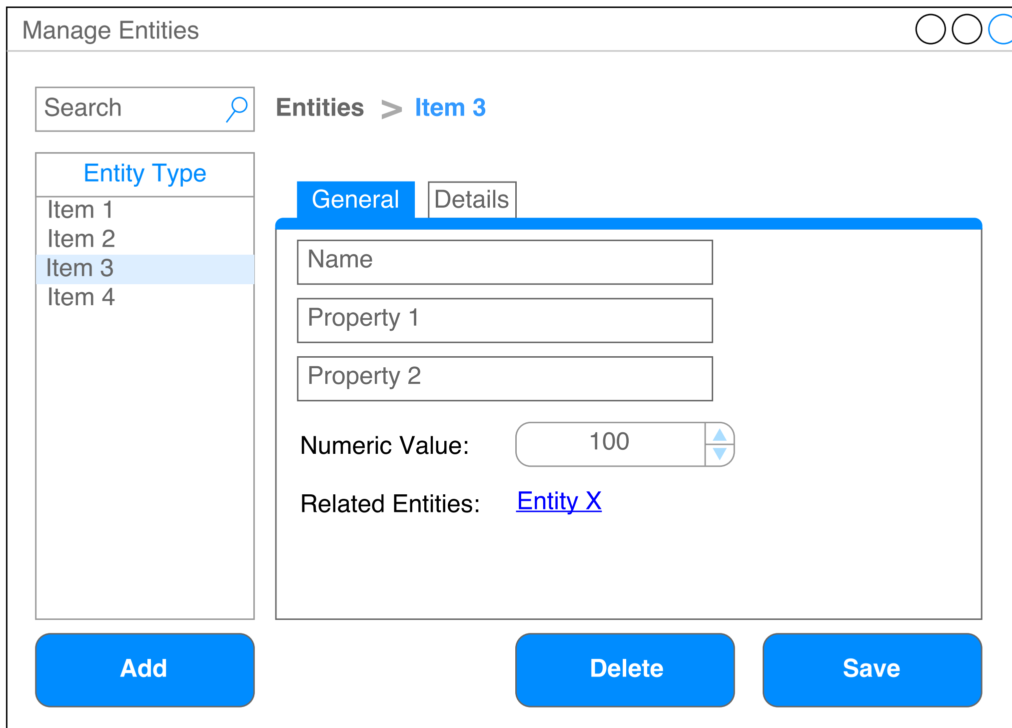


Abbildung 4.10: In der Ansicht für die Stammdatenverwaltung kann ein Element bspw. gesucht oder abgeändert werden.

4.6.1 Stammdatenverwaltung

Die vom Back-End implementierte Stammdatenverwaltung der Services, Cloud-Plattformen oder deployten Messaging-Middlewares und Service-Kompositionen muss dem Nutzer ebenfalls in der Oberfläche (Front-End) zur Verfügung stehen. Hierbei sollen ihm die hinter `CRUD` stehenden Datenoperationen zum Anlegen, Abrufen, Abändern und Löschen von Entitäten angeboten werden, die im Mock-Up in Abbildung 4.10 dargestellt sind. Um auch bei großen Datenmengen den Überblick zu behalten, sollen die Daten nach dem Overview-Detail-Pattern dargestellt werden, sodass er sich in einer Übersicht einen Überblick bilden kann und bei Bedarf Detailinformationen zu einer bestimmten Auswahl anzeigen kann. Mit Hilfe von Filter- und Suchmechanismen oder Verknüpfungen zwischen Entitäten kann der Umgang mit den Daten weiter vereinfacht werden. Um die manuelle Übertragung einer Service-Definition in die Anwendung zu vereinfachen, soll hierfür eine Datei-Upload-Schnittstelle bereitstellen, sodass die Plattform die Definitionsdatei automatisch einlesen kann.

4.6.2 Abbildung der Hintergrundprozesse

Für den Anwender sollen die im Hintergrund auf den Stammdaten ablaufenden Prozesse transparent dargestellt werden und kontrollierbar sein. Zu diesen Prozessen gehören hauptsächlich mit dem Deployment zusammenhängende Prozesse, wie die Provision von Cloud-Umgebungen oder auch das eigentliche Deployment, bei dem der Anwender die zuvor angelegten Stammdaten miteinander verbindet und ihnen eine Funktion gibt.

4.6.3 Abstraktion der Fachprozesse

Mit der Anforderung an eine einfache und möglichst von technischen Laien bedienbare Oberfläche ergibt sich, dass technische Prozesse abstrahiert werden sollen. Die Vorbereitung des eigentlichen Deployments aus einer Orchestrierung, die einzelnen Aufrufe der Docker-Laufzeitumgebung oder die Einzelheiten zur Konfiguration der Kommunikationsebene sollen dem Benutzer zwar transparent gezeigt werden, jedoch auf sinnvolle Teilschritte aufgeschlüsselt werden. Diese Teilschritte, wie bspw. *Deployment starten* oder *Deployment stoppen*, sollen je nach Zustand steuerbar sein.

4.6.4 Visualisierung der Orchestrierung

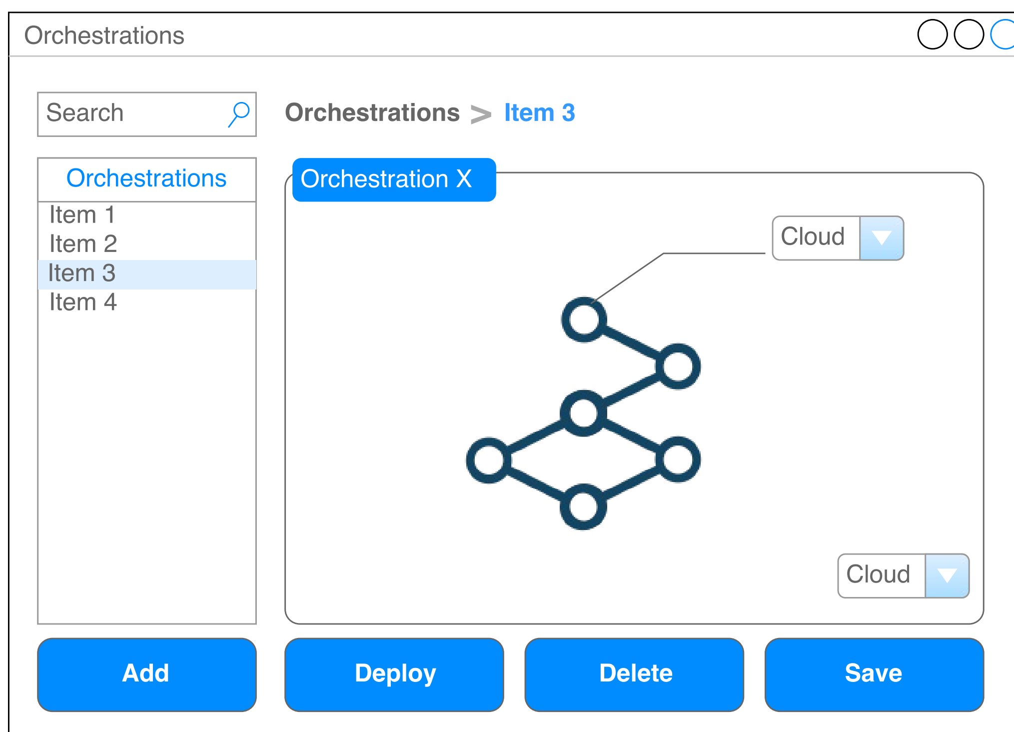


Abbildung 4.11: In der Ansicht für die Visualisierung der Orchestrierung kann der Nutzer die Orchestrierung bearbeiten, konfigurieren und deployen.

Die Visualisierung der Orchestrierung inklusive des Deployments zählt als grafische Hauptkomponente der Nutzeroberfläche. Der Anwender soll eine unkomplizierte Verknüpfung der Services zusammen mit einer Angabe der Deployment-Konfiguration bspw. für die Zielplattform vornehmen können. In einem Graphdiagramm sollen wie in Abbildung 4.11 Services als Knoten einfach eingefügt und je nach verfügbarer Schnittstelle mit anderen Services verknüpfbar sein (Kanten). Aus dem hieraus entstehenden Graphen soll der Anwender direkt den Aufbau der Service-Komposition und alle weiteren wichtigen Informationen entnehmen können.

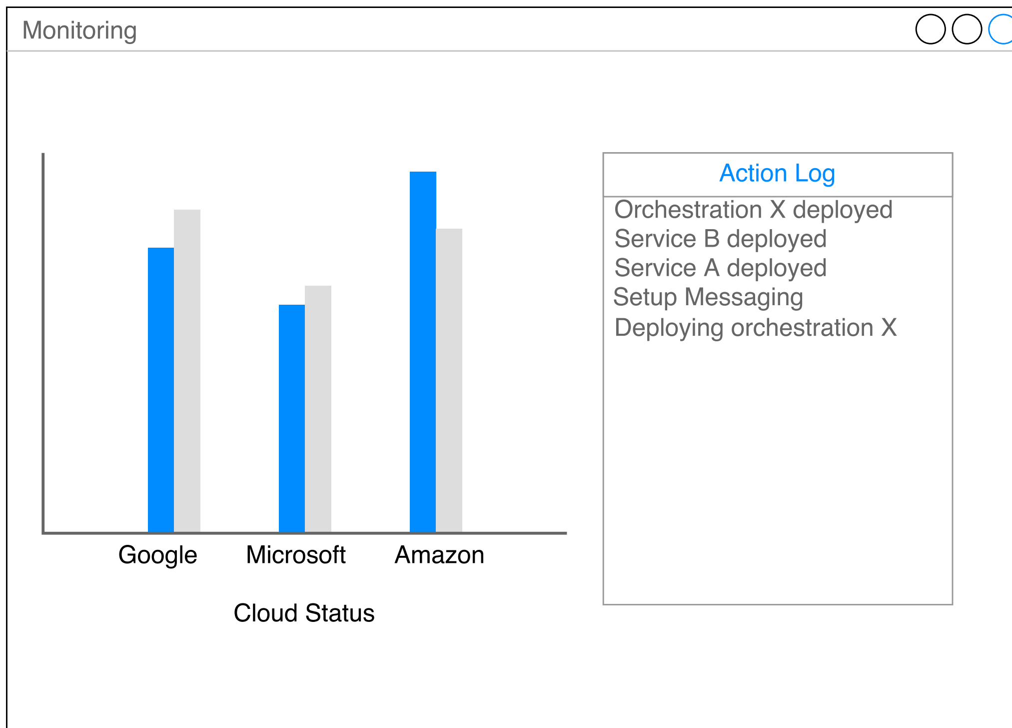


Abbildung 4.12: Über verschiedene Ansichten kann der Anwender auf der Monitoring-Seite den Status der Plattform und der Deployments prüfen.

4.6.5 Monitoring

Außer einem Überblick über die laufenden Hintergrundprozesse soll der Anwender wie in Abbildung 4.12 den Status der gesamten Plattform und einzelner deloyter Orchestrierungen laufend mitgeteilt bekommen. Hierzu gehören bereits einfache Informationen zu abgeschlossenen Aktionen inklusive einer Fehlerbeschreibung, sofern sinnvoll. Es sollen ebenfalls die Ergebnisse der Hintergrundprozesse oder plötzlich auftretende Ereignisse wie bspw. ein Ausfall in der Oberfläche dargestellt werden. Darüber hinaus können allgemeine Informationen wie problematische Deployments oder inaktive Cloud-Umgebungen, die einen Nutzereingriff erfordern, in separaten Ansichten, wie einem einfachen Dashboard, direkt in den Fokus gelangen.

5 Konzept – Maschinenanbindung

Mithilfe der im vorigen Kapitel konzipierten Plattform soll es ebenfalls möglich sein, Industriemaschinen direkt aus Services anzusprechen. Bisher wäre notwendig gewesen, dass für jede Maschine ein neuer Service entwickelt wird, der die entsprechenden Daten auslesen, verarbeiten und weiterreichen kann. Durch Verwenden eines generischen Services mit einer entsprechenden Konfiguration sollen allerdings flexibel Industriemaschinen angesprochen werden, um die Daten in wiederverwendbaren Services weiterverarbeiten zu können.

5.1 Anforderungen

Nutzer der Service-Plattform sollen ihre Industriemaschinen unkompliziert integrieren können, sodass deren Daten einfach nutzbar sind. Der Nutzer soll hierfür nicht für jede neue Maschine und jeden neuen Anwendungsfall einen eigenen Service entwickeln müssen, sondern in der Benutzeroberfläche ohne technische Kenntnisse direkt die Anbindung an die Maschine anlegen können. Die in der Oberfläche generierten Services sollen als reiner Schnittstellenservice zur Maschine funktionieren. Hierfür ist eine einheitliche Schnittstelle zu allen Maschinen erforderlich, die vom Service angesprochen werden kann.

5.2 Generischer Maschinenservice

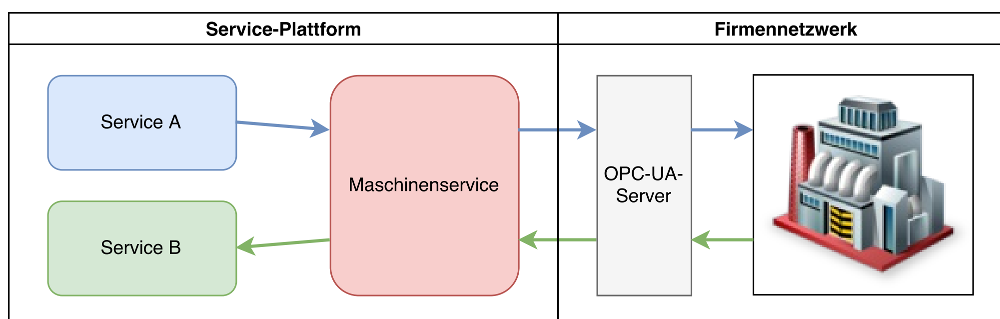


Abbildung 5.1: Der Integration eines in die Plattform integrierten Maschinenservices, der in einer Service-Komposition Daten mit einer Maschine austauscht.

Der Maschinenservice soll wie in Abbildung 5.1 als Schnittstelle zwischen einfachen Services und einer Industriemaschine stehen. Für die flexible Anbindung an diese Maschine wird eine einheitliche Schnittstelle zu ihr benötigt, da sonst eine Implementierung eines speziellen Kommunikationsprotokolls pro Werkzeugmaschine notwendig wäre. Da OPC-UA (siehe OPC Unified

Architecture in den Grundlagen) ein unter Werkzeugmaschinen weit verbreitetes Protokoll ist, bietet es sich bei der Wahl als Standard-Kommunikationsprotokoll an. Hierfür wird allerdings ein mit der Maschine verknüpfter OPC-UA-Server benötigt. Der Maschinenservice kann diesen Server nun über das OPC-UA-Protokoll ansprechen und Daten aus der Maschine abrufen oder an ihn senden. Um den OPC-UA-Server anzusprechen, werden die folgenden Informationen benötigt:

OPC-UA-Server-URL Die URL gibt an, welches Protokoll (bspw. TCP oder HTTPS) zur Kommunikation genutzt werden soll und wo der OPC-UA-Server lokalisiert ist.

Abrufbare Werte Für jede abrufbare Datenquelle der Maschine wird der Namespace und der dahinter liegende Knoten benötigt. Ein Sampling-Intervall gibt dabei an, wie oft der OPC-UA-Server Werte von der Maschine abruft. Der Publish-Intervall legt fest, wie oft die abgerufenen Werte zum Client — in dem Fall dem generischen Service — gesendet werden.

Veränderbare Werte Für jeden änderbaren Wert muss allein der Namespace und der Knoten auf dem OPC-UA-Server bekannt sein.

Ein OPC-UA-Client kann nach Konfiguration mit den oben beschriebenen Eigenschaften flexibel OPC-UA-Server ansprechen sowie deren Werte lesen und schreiben. Um den OPC-UA-Client plattformunabhängig in der Service-Plattform nutzen zu können, muss er in einen Service verpackt werden. Die Konfiguration kann anschließend von der Plattform aus wie gewohnt über die Umgebungsvariablen stattfinden. Damit der Client auch von anderen Services genutzt werden kann, muss er an die Kommunikationsschicht der Plattform angebunden werden. Für den Austausch mehrerer unterschiedlicher Daten der Werkzeugmaschine muss der Maschinenservice bei der Kommunikation die einzelnen Arten unterscheiden und verknüpfen können. Hierfür muss mithilfe einer Id oder eines Namens die Datenquelle (bzw. der Zielort) bei der Kommunikation mit anderen Services sowie mit dem OPC-UA-Server identifizierbar sein, was für die Verknüpfung mit anderen Services bekannt sein und somit in der Konfiguration stehen muss.

5.3 Integration in Service-Plattform

Nach dem bisherigen Konzept werden Services durch Einlesen einer Service-Definition, welche alle notwendigen Informationen enthält, in die Plattform geladen. Hierbei beschreibt eine Definitionsdatei genau einen bestimmten Service mit seinen Schnittstellen in einer Version. Für einen generischen Maschinenservice mit einer variablen Schnittstelle zur Plattform hin werden nun bei gleichem Docker-Image verschiedene Service-Definitionen benötigt, da die Schnittstelle des Services über seine Konfiguration in der Definitionsdatei festgelegt wird. Diese Definition kann beim Maschinenservice in einem Konfigurationsschritt angelegt werden, wo die im obigen Abschnitt festgelegten Parameter konfiguriert werden können. Im Unterschied zu einem traditionellen Service unterscheidet sich die Definition zum einen in den Properties, die mit Standard-Werten für die dauerhafte Konfiguration genutzt wird, und zum anderen in den Ein- und Ausgängen des Services, die je nach geforderter Schnittstelle anders aussehen können.

Die Properties bieten allerdings nur wenige Möglichkeiten, um einen Service strukturiert zu konfigurieren. Properties können grundlegende Datentypen sein wie Zahlen, Texte oder boolesche Werte, mit denen eine Liste von Datenquellen der Werkzeugmaschine nicht direkt angegeben werden kann. Die OPC-UA-Server-URL sowie alle Ein- und Ausgänge könnten in Textform in jeweils einer Property angegeben werden, wobei ein einheitliches Schema zur Benennung

gewählt werden muss, damit der Service nach dem Deployment die Konfiguration vollständig einlesen kann. Hierbei ergibt sich allerdings ein Problem beim Einlesen der Konfiguration, da diese über Umgebungsvariablen stattfindet und der Service deshalb herausfinden müsste, wie viele Umgebungsvariablen vorhanden sind. Daher soll die Konfiguration in einem komplexeren Datenformat innerhalb von einer Property stattfinden, die unter anderem auch Auflistungen unterstützt.

Listing 5.1 Eine im JSON-Format beschriebene Konfiguration für OPC-UA-Endpunkte.

```
{
  "drehzahl": {"namespace": 2, "knoten": 3, "samplingIntervall": 5000},
  "temperatur": {"namespace": 2, "knoten": 4, "samplingIntervall": 5000}
}
```

Mit XML, JSON oder auch YAML können, wie in Listing 5.1 dargestellt, in getrennten Properties alle Ein- bzw. Ausgänge in einer Art Key-Value-Map definiert werden, wobei der Key zur Identifikation des Ein- bzw. Ausgangs und der Value zur Konfiguration genutzt werden kann. Zur vollständigen Integration in die Service-Plattform wird weiterhin eine gültige Angabe aller Ports benötigt, die zur Verknüpfung mit anderen Services genutzt werden kann.

5.4 Benutzeroberfläche zur Generierung

Abbildung 5.2: Im Mock-Up der Oberfläche zum Hinzufügen eines generischen Maschinenservices kann der Anwender alle notwendigen Informationen erfassen und die Service-Konfiguration generieren.

Die Benutzeroberfläche für die Generierung des Maschinenservices kann prinzipiell an der Oberfläche zum Anlegen eines normalen Services angelehnt sein (vgl. Abbildung 5.2). Für einen konventionellen Service sollen in der Oberfläche alle Felder gepflegt werden können, die in der Definitionsdatei enthalten sein können. Beim Anlegen eines generischen Maschinenservices wird die Auswahl auf das Docker-Image des generischen Services bereits festgelegt, sodass diese Daten nicht mehr gepflegt werden müssen. Beim abschließenden Hinzufügen des Services soll aus diesen Angaben die Konfiguration der OPC-UA-Ein- und -Ausgänge sowie der Ports des Services generiert werden.

6 Konzept – Zuverlässigkeit & Skalierbarkeit

In diesem Kapitel wird ein Konzept ausgearbeitet, mit dem die zuvor beschriebene Service-Plattform und deren Services zuverlässig betrieben werden können. Dabei soll ebenfalls auf die Skalierbarkeit der einzelnen Komponenten eingegangen werden, die stark mit einem redundanten Betrieb korreliert, um auch in Situationen mit großer Auslastung einen Normalbetrieb zu ermöglichen. Die Reife (vgl. Abschnitt 2.6) wird hierbei nicht weiter berücksichtigt, da erwartet wird, dass jede Komponente bereits im Normalbetrieb zuverlässig funktioniert. Die folgenden Abschnitte sollen klären, wie eine hohe Zuverlässigkeit der Plattform-Komponenten erreicht werden kann, indem die Verfügbarkeit, die Fehlertoleranz und die Wiederherstellbarkeit beeinflusst werden.

6.1 Notwendigkeit bei Cloud-Computing

Cloud-Plattformen wie Google[1] oder Microsoft[2] werben mit einer hohen Verfügbarkeit ihrer Dienste von über 99,95%, was einer nicht verfügbaren Zeit von knapp 4,5 Stunden pro Jahr entspricht. Zum besseren Verständnis dieser Angabe muss der Begriff *Verfügbarkeit* allerdings genauer definiert werden.

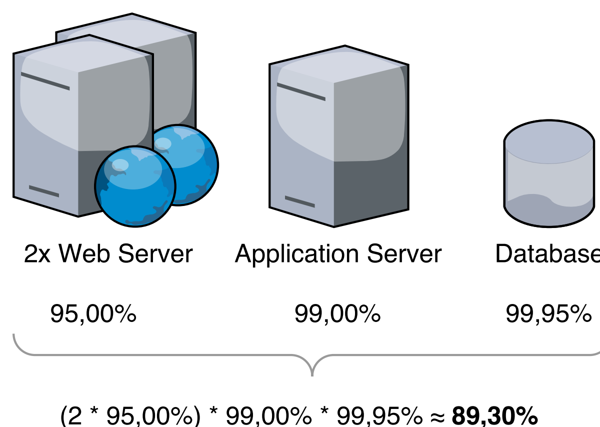


Abbildung 6.1: Bei einer Knoten-basierten Verfügbarkeit werden die Verfügbarkeiten der einzelnen Ressourcen multipliziert, um die Verfügbarkeit der gesamten Anwendung zu erhalten.

[1]Verfügbarkeit der Google Compute Engine, https://cloud.google.com/compute/sla
[2]Verfügbarkeit der Microsoft Cloud Dienste, https://azure.microsoft.com/de-de/support/legal/sla/cloud-services/v1_5/

Mit der Verfügbarkeit soll angegeben werden, wie lange eine Ressource relativ gesehen ihre Funktion erfüllen kann und nicht ausgefallen ist. Fehling et al. unterscheiden in [FLR+14] zwischen zwei Arten der Verfügbarkeit. Sie kann beispielsweise nach der *Node-based Availability* auf Grundlage der Verfügbarkeit der einzelnen Komponenten wie etwa virtuellen Maschinen oder einer Datenbank angegeben werden. Hierbei ergibt sich die Gesamtverfügbarkeit durch Multiplizieren der einzelnen Verfügbarkeiten, wie in Abbildung 6.1 an einem Beispiel demonstriert wird. Im Gegensatz dazu kann mit der *Environment-based Availability* angegeben werden, wie hoch die Verfügbarkeit der gesamten Plattform ist. Hierbei wird bspw. mit eingerechnet, wie hoch die Wahrscheinlichkeit ist, dass eine frei virtuelle Maschine verfügbar ist und angefordert werden kann. Nach dieser Definition kann die Verfügbarkeit einzelner Ressourcen deutlich geringer sein, wie auch ein Interview über Googles Rechenzentren[3] von 2008 zeigt. So fielen im ersten Jahr eines Clusters mit 1800 Servern 1000 einzelne Server aus und es bestand eine Wahrscheinlichkeit von etwa 50%, dass ein Cluster überhitzt. Diese Statistiken zeigen, dass der Ausfall einer einzelnen Ressource deutlich wahrscheinlicher als zuerst angenommen ist [FLR+14].

6.2 Hosts

Auf unterster Ebene im Application-Stack der Service-Plattform stehen in einer IaaS-Cloud die virtuellen Maschinen bzw. in einer PaaS-Cloud die jeweiligen Anwendungscontainer — im Folgenden als Hosts bezeichnet. Wie in Abschnitt 3.4.1 beschrieben, kann es in einer Cloud zu Ausfällen in verschiedenen Komponenten wie dem Netzwerk oder der zugrundeliegenden Hardware kommen.

Die Skalierbarkeit sowie die Zuverlässigkeit der Hosts liegen bei einer PaaS-Cloud außerhalb der Reichweite der Plattform und können somit nicht weiter beeinflusst werden. Werden von der Service-Plattform mehr Ressourcen benötigt, könnten diese je nach Art direkt angefordert und werden, da sie vom elastischen Cloud-Anbieter stets bereitgehalten werden. Da außerdem ein großer Anteil an Ressourcen wie Datenspeichern oder Netzwerken bereits virtualisiert ist, wirken sich die Ausfälle der Hardware nicht direkt auf sie aus.

In einer IaaS-Cloud kann die Zuverlässigkeit der Ressourcen ebenfalls nicht direkt beeinflusst werden. Für eine vollständige Skalierbarkeit der genutzten Ressourcen sind dahingegen Controller zur Skalierung von Hosts, Netzwerk-Komponenten oder Load-Balancern notwendig. Im Multi-Cloud-Szenario muss entweder ein Controller alle Cloud-Plattformen und die dort genutzten Ressourcen verwalten können oder mehrere Cloud-spezifische Controller verwendet werden. Für den Deployment-Prozess der Service-Plattform ist es nun erforderlich, dass entweder immer ausreichend freie Hosts vorhanden sind, oder dass währenddessen die Skalierung stattfinden kann und das Deployment anschließend fortgesetzt wird.

[3] Statistiken zu Googles Rechenzentren, http://www.datacenterknowledge.com/archives/2008/05/30/failure-rates-in-google-data-centers

6.3 Services

Die Zuverlässigkeit eines Services in Form eines Docker-Containers kann durch die Service-Plattform nicht beeinflusst werden, da sie ausschließlich von externen Ressourcen wie den Hosts oder der Docker-Engine abhängt. Betrachtet man einen Service allerdings als abstrakte Einheit mehrerer Container, die bspw. hinter einem Load-Balancer liegen, kann die Plattform durch Redundanzen die Zuverlässigkeit effektiv erhöhen (vgl. Abbildung 6.2).

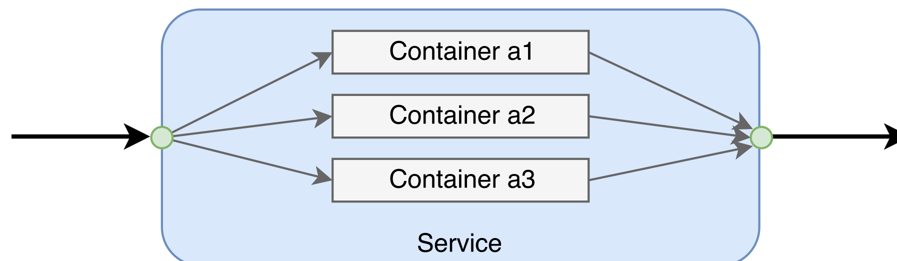


Abbildung 6.2: Durch redundant laufende, äquivalente Container, die dennoch als ein Service gewertet werden, kann die Zuverlässigkeit erhöht werden.

Mit mehreren äquivalenten Docker-Containern des gleichen Docker-Images mit der gleichen Konfiguration kann die Service-Plattform einen zuverlässigeren Service deployen. Hierfür wird eine Art Load-Balancer benötigt, der zum Beispiel ein separater Container sein könnte, welcher allerdings ebenfalls redundant ausgelegt sein sollte und somit wieder einen Load-Balancher hierfür benötigen würde. Außer der einfachen Replikation der Container müssen die Nachrichten in diesem Szenario nicht mehr nur an einen, sondern an mehrere Container verteilt werden. Aus diesem Grund soll die Message-oriented-Middleware nicht nur die Kommunikation zum logischen Service übernehmen, sondern auch die zu den tatsächlichen Containern des Services. Um die Zuverlässigkeit weiter zu erhöhen, sollten die einzelnen Container mindestens in verschiedenen Zonen der ausgewählten Region (gleiche geografische Lage, aber unterschiedliche Rechenzentren) der festgelegten Cloud-Anbieter deployt werden.

Als Grundlage hierfür werden allerdings skalierbare Services benötigt. Wie in [VRB11] und mit den IDEAL-Eigenschaften einer Cloud-Anwendung beschrieben wurde, ist die Verwaltung des Zustands außerhalb der Container — beispielsweise in einer Datenbank — erforderlich. Der Zugriff auf diesen Zustand muss in jedem Fall so erfolgen, dass trotz gleichzeitiger Zugriffe die Daten konsistent bleiben. Ist diese Anforderung nicht erfüllt, kann ein inkonsistenter Zustand entstehen, der beispielsweise einen redundant aufgebauten Zähler-Service untauglich machen würde. Da die Konfiguration der Kommunikation nicht von einem Container, sondern von der Plattform vorgenommen wird, besteht an dieser Stelle kein Mehraufwand auf Seite des Services bzw. Containers.

6.4 Kommunikation

Mit der Realisierung der Kommunikationsschicht durch Apache Kafka, welcher zusammen mit den weiteren notwendigen Komponenten als Infrastruktur-Service betrieben wird, kann für eine Steigerung der Zuverlässigkeit von Apache Kafka selbst prinzipiell das bei den Services

beschriebene Konzept angewendet werden. Apache Kafka nennt die einzelnen zur Kommunikation notwendigen Komponenten *Broker*. Die einzelnen Broker, über die die Nachrichten laufen, werden mit weiteren administrativen Aufgaben von *Zookeeper* koordiniert, der den gleichzeitigen Betrieb mehrerer Instanzen unterstützt. Kafka unterstützt von sich aus den redundanten Betrieb mehrerer Broker und erfordert standardmäßig mindestens drei Broker-Instanzen, auf denen die Replikation der Nachrichtenkanäle (Topics) stattfindet. Der Grad der Replikation kann konfiguriert werden und erfordert ein entsprechendes Deployment der dafür notwendigen Broker. Sobald ein Service eine Nachricht über Kafka versendet, wird erst nach einer erfolgreichen Replikation in Kafka signalisiert, dass die Nachricht erfolgreich empfangen wurde. Somit ist sichergestellt, dass die Nachricht nach erfolgreichem Versand im Service nicht mehr verloren gehen kann. Die Kafka-API im Service-Core kann hierbei die Adressen mehrerer Broker annehmen, sodass bei einem Ausfall eines Brokers die Nachrichten an einen anderen verfügbaren gegeben werden können [Sha16].

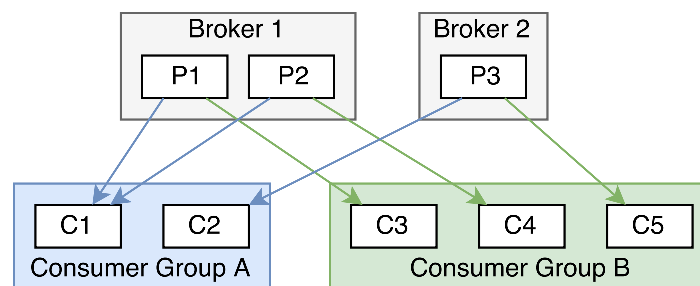


Abbildung 6.3: Apache Kafka realisiert logische Empfänger mit Consumer-Groups. Die Consumer innerhalb einer Consumer-Group erhalten die Nachrichten eines Topics gleichmäßig verteilt. Quelle: https://kafka.apache.org/documentation/

Nach dem jetzigen Konzept erhält jeder Container alle Nachrichten, die an den zugehörigen Service gesendet werden, wodurch das oben beschriebene Szenario nicht anwendbar wäre. Apache Kafka bietet allerdings mit den sogenannten *Consumer Groups* bereits eine Unterstützung für das Konzept von einem logischen Empfänger, hinter dem sich mehrere tatsächliche Empfänger befinden (vgl. Abbildung 6.3). Eine Consumer-Group ist eine Gruppe von mehreren Empfängern, die die Nachrichten von einem Topic empfängt. Ein Service lässt sich somit einer Consumer-Group gleichsetzen, während die Container des Services die eigentlichen Consumer sind. Apache Kafka unterteilt Topics in *Partitions*, welche eine Skalierbarkeit eines Topics durch Verteilen der zugehörigen Partitions ermöglicht. Soll ein Topic von einer Consumer-Group konsumiert werden, leitet Kafka alle vorhandenen Partitions (P1–P3) dementsprechend an die einzelnen Empfänger — gegebenenfalls auch mehrere Partitionen auf einen Consumer wie im Beispiel der abgebildeten *Consumer Group A*. Hierfür ist es notwendig, dass jeder Consumer der Kafka-API angibt, zu welcher Gruppe er gehört. Diese Parameter kann die Service-Plattform vor dem Deployment aller Services sammeln und anschließend alle Services mit einer zuverlässigeren Konfiguration der Kommunikation deployen.

6.5 Plattform

Die Service-Plattform selbst kann prinzipiell ebenfalls als zustandsloser Service betrieben werden, in dem sich mehrere Docker-Container befinden. Bei Services übernimmt die Plattform die Aufga-

ben zur Skalierung und verteilt sie entsprechend der Anforderungen. Da die Plattform nicht direkt innerhalb der Plattform selbst deployt werden kann, muss hierfür eine außenstehende Komponente mit entsprechenden Orchestrierungsfähigkeiten das zuverlässige Deployment umsetzen. Damit die Plattform überhaupt Services orchestrieren kann, ist es erforderlich, dass entweder direkt die Schnittstellen einer PaaS-Plattform angesprochen werden können, oder dass eine Middleware als Abstraktion verwendet werden kann. Sollen Services auf einer IaaS-Plattform skaliert werden, müssen hier ebenfalls entsprechende Schnittstellen genutzt werden können, um neue Maschinen und weitere (virtualisierte) Hardware zu provisionieren.

Da ein Plattform-Container während eines Deployment-Prozesses ausfallen kann, könnte bspw. jeder einzelne Schritt mitsamt des vollständigen Kontexts zur späteren Fortsetzung persistiert werden. Hierbei ist allerdings eine Koordination der einzelnen Container untereinander notwendig, sodass unvollständige Deployments unmittelbar fortgesetzt werden können. Die Maßnahme ist allerdings wirkungslos, wenn zum Beispiel die Persistierung fehlschlägt und der Zustand in der Datenbank inkonsistent mit dem unvollständigen Deployment ist. Aus diesem Grund soll außerdem erkannt werden, wenn der Zustand inkonsistent ist und ggf. unvollständige Deployments aufgeräumt und neu begonnen werden.

7 Implementierung

In einer prototypischen Implementierung wurde in Partnerarbeit eine Service-Plattform entwickelt, mit der die grundlegenden Anforderungen umgesetzt und validiert werden konnten. Die Plattform ermöglicht eine Orchestrierung mehrerer Services in einer Komposition und unterstützt ein Deployment auf Maschinen verschiedener Cloud-Provider. Die Services können mithilfe eines generischen Services (vgl. Kapitel 6) einfach Daten von Industriemaschinen abgreifen und auswerten. Die Zuverlässigkeit und Skalierbarkeit der einzelnen Plattform-Komponenten aus dem vorigen Kapitel konnte nicht mehr im Rahmen dieser Arbeit umgesetzt werden. Im Folgenden wird die allgemeine Umsetzung der Service-Plattform beschrieben, gefolgt von detaillierteren Erläuterungen der umgesetzten Hauptanforderungen.

7.1 Architektur

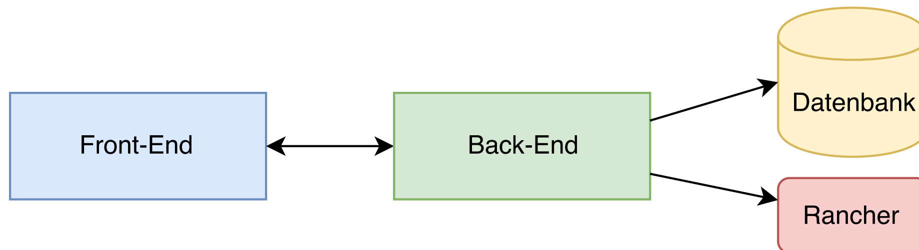


Abbildung 7.1: Der Nutzer greift auf die Funktionalität der Anwendung über das Front-End zu, welches über das Back-End auf die Datenbank zugreift oder mit Rancher kommuniziert.

Die Service-Plattform wurde als Webanwendung, bestehend aus Front-End, Back-End, Datenbank sowie der zentralen externen Komponente *Rancher* (siehe Abschnitt 3.2.4), konzipiert (vgl. Abbildung 7.1). Das Back-End implementiert eine REST-Schnittstelle für das Front-End, verwaltet Stammdaten und steuert die Hintergrundprozesse, während das Front-End dem Anwender eine Nutzeroberfläche bietet. Alle drei Komponenten sind nach dem Service-Konzept ebenfalls mit Docker containerisiert lauffähig.

7.1.1 Front-End

Das Front-End besteht hauptsächlich aus entkoppelten Komponenten, die eine Sicht auf die Funktionen und Hintergrundprozesse des Back-Ends bieten. Die meisten Komponenten haben hauptsächlich den Zweck, die einzelnen Ansichten der Weboberfläche (vgl. Abbildung 7.2) gekapselt darzustellen:

Dashboard Im Dashboard erhält der Anwender einen Überblick über den Status der Cloud-Hosts sowie Statistiken über die Service-Plattform.

Confluent Die Konfiguration der Messaging-Komponente Apache Kafka und der dazugehörigen Schema-Registry von Confluent werden in diesem Reiter vorgenommen.

Cloud Virtuelle Maschinen und andere Cloud-Hosts können hier verwaltet werden. Über eine Status-Anzeige kann der Anwender erfahren, ob die Ressource aktuell verfügbar ist.

Services Die einzelnen Services können an dieser Stelle mitsamt Konfigurationsmöglichkeiten und der Service-Schnittstelle gepflegt werden.

Graph Der Anwender kann zuvor angelegte Services in einer Graphvisualisierung verknüpfen und daraus Orchestrierungen bilden, die auf verschiedene Cloud-Plattformen deployt werden können (vgl. Abbildung 7.2).

Schemas Um einen einfachen Zugriff auf die Schema-Registry zu gewähren, ist sie ebenfalls in die Service-Plattform eingebettet.

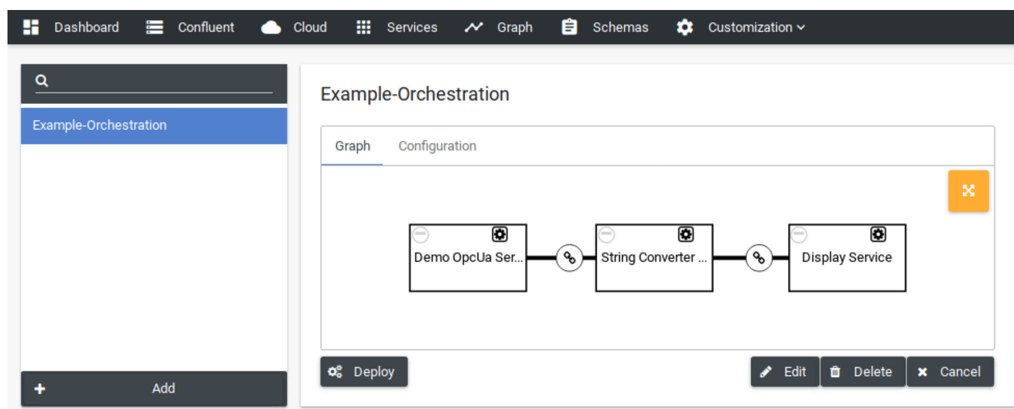


Abbildung 7.2: Der Anwender kann Service-Kompositionen in einer Graphvisualisierung erstellen und das Deployment auf verschiedene Cloud-Plattformen ansteuern.

Die Webanwendung wurde mit dem Open-Source Framework *Angular*[1] in der Version 4 entwickelt. Angular ermöglicht eine modulare Entwicklung komplexer Webanwendungen und unterstützt dabei mit einer Vielzahl an Programmbibliotheken — bspw. zum Verwalten der unterschiedlichen Quellcodedateien für Logik und die Ansichten oder zur Kommunikation mit dem Back-End über eine REST-Schnittstelle. Angular 4 unterstützt die Programmiersprache *TypeScript*, die ein typsicheres Superset von JavaScript ist und hierdurch den Plattformentwickler unterstützt. Die Modularisierung des Front-Ends ist weiter durch das Framework *ngrx*[2] ermöglicht worden, das die Prinzipien der reaktiven Programmierung mit Angular kombiniert. Die Oberflächengestaltung wird hauptsächlich mit *PrimeNG*[3] realisiert, welches eine tiefgreifende Integration in Angular und die Angular-eigenen Komponenten bietet sowie durch *D3.js*[4] für die Graphvisualisierung ergänzt wird.

[1] Angular, https://angular.io/
[2] ngrx, https://github.com/ngrx/platform
[3] PrimeNG, https://www.primefaces.org/primeng/
[4] D3.js, https://d3js.org/

7.1.2 Back-End

Über die REST-Schnittstelle und einen Websocket als bidirektionalen Kommunikationsweg wird die gesamte Funktionalität für die Weboberfläche bereitgestellt. Diese Schnittstelle verknüpft alle Komponenten im Back-End, wie die Stammdatenverwaltung aller Entitäten und die anschließende Persistierung in einer Datenbank. Nach einer Umwandlung und Umstrukturierung aller Informationen einer Orchestrierung wird das Deployment über Rancher angestoßen. Über Rancher werden außerdem alle Hosts verwaltet und die deployten Services überwacht.

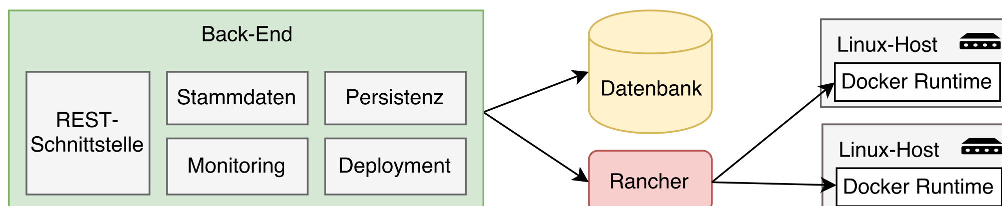


Abbildung 7.3: Das Back-End verwaltet die Stammdaten und alle Prozesse, um mit Rancher die Container auf den Hosts zu steuern.

Das Back-End ist in der Programmiersprache *Kotlin*[5] entwickelt worden, welche die Java-Virtual-Machine verwendet und somit einem breiten Spektrum an Java-Programmbibliotheken offen steht. Mit Hilfe des *Spring Boot*[6]-Frameworks konnte das Back-End sehr modular entwickelt werden. Die Persistierung in einer Datenbank sowie die REST-Schnittstelle konnten hiermit mit sehr geringem Aufwand umgesetzt werden. Durch das Build-Tool *Gradle*[7] werden hierbei die Abhängigkeiten verwaltet und aus Back-End sowohl als Front-End Build-Artefakte und zugehörige Docker-Images erzeugt.

7.2 Services

Die Implementierung von verschiedenen Services zu Entwicklungs- und Validierungszwecken wurde in Form von mit Docker containerisierten Java- und Kotlin-Anwendungen vorgenommen. Hierfür wurde ein Service-Core in Kotlin entwickelt, der aufgrund der gemeinsamen Java-Laufzeitumgebung ebenfalls von den Java-Services genutzt werden kann. Jeder Service kann über diesen Service-Core die in Form von Umgebungsvariablen vorgenommene Konfiguration konsistent auslesen. Die Kommunikation über Apache Kafka wurde ebenfalls deutlich abstrahiert und automatisch konfiguriert. Jeder Service nutzt die Kommunikationsschnittstelle, um Nachrichten in Nachrichtenkanäle mittels der dem Service bekannten Ausgänge zu senden. Um Nachrichten zu empfangen, registriert der Service unter dem Namen des Eingangs einen Callback im Service-Core, welcher bei Eintreten einer neuen Nachricht aufgerufen wird. Die Container wurden daraufhin in die öffentliche Docker-Registry *Docker Hub* hochgeladen, um anschließend von der Plattform verwendet zu werden.

[5]Kotlin, https://kotlinlang.org/
[6]Spring Boot, https://projects.spring.io/spring-boot/
[7]Gradle, https://gradle.org/

7.3 Kommunikation

Die Kommunikation zwischen Services wurde, wie im Konzept beschrieben, mit Apache Kafka realisiert. Um die Komplexität eines automatischen Kafka-Cluster-Deployments für den Prototypen zu vermeiden, erfordert die Plattform einen fest hinterlegten Kafka-Cluster mitsamt der notwendigen Adressen bspw. zu den einzelnen Brokern. Dieser Cluster muss vor dem ersten Deployment einer Orchestrierung manuell deployt und konfiguriert werden, um anschließend ohne weitere Konfiguration von der Plattform genutzt werden zu können. Die im Konzept beschriebene Anforderung eines Clusters pro Orchestrierungs-Deployment konnte aufgrund des fehlenden automatischen Deployments nicht umgesetzt werden.

7.4 Deployment

Die Multi-Cloud-Unterstützung und das Deployment der Docker-Container wurde mit der Rancher-Plattform umgesetzt. Rancher ermöglicht eine vollständige Steuerung über die bereitgestellte REST-API, wodurch Hosts verschiedener Cloud-Plattformen hinzugefügt, provisioniert und gesteuert werden können. Rancher baut hierfür selbstständig ein Netzwerk auf, in welchem sich jeder Host automatisch befindet, sodass die Services von jedem Host mit anderen Services kommunizieren können. Während des Deployments in den Rancher-Cluster und beim Betrieb der Services kann über die Schnittstelle der Status der Hosts und Services abgefragt werden, um ihn in der Service-Plattform anzuzeigen. Die Rancher-eigene Orchestrierung sorgt bei Ausfällen einzelner Services und Hosts für ein automatisches erneutes Deployment auf anderen verfügbaren Hosts. Aktualisierungen an deployten Services werden von Rancher in zwei Schritten vorgenommen, sodass zuerst ein Service mit der neuen Konfiguration gestartet und anschließend der alte Service gelöscht wird, um eine annähernd durchgängige Verfügbarkeit zu gewährleisten.

7.5 Typsicherheit

Der typsichere Datenaustausch zwischen Services ist einer der grundlegenden Anforderungen an die Service-Plattform, die auf allen Ebenen gewährleistet sein soll. In jeder Service-Definition muss bei jedem Input und Output das zugehörige Schema spezifiziert werden. Hierfür stehen ihm zwei Möglichkeiten zur Verfügung: Die direkte Angabe des Schemas oder eine darauf verweisende URL. So können in der Service-Definition (vgl. Listing 4.1) sowohl einfache als auch komplexe Schemas wie in Listing 7.1 direkt angegeben werden.

Die Schemas werden hierbei im JSON-Format angegeben und sind flexibel definierbar. Apache Avro unterstützt einfache Typangaben wie boolesche Werte, Ganz- und Fließkommazahlen sowie Bytes und Texte, aber auch komplexe Strukturen wie Auflistungen, Schnittmengen von Typen oder Kombinationen in sogenannten Records. Im obigen Beispiel wird das Schema eines Steckbriefs einer `Person` definiert. Hierfür wird ein Record erstellt, der mehrere Felder auf deren Typen abbildet. Dieser Record kann alternativ auch unter dem Typen `Mensch` gefunden werden, wie im `aliases`-Tag angegeben ist. Eine `Person` kann über einen Namen in Textform, ein Alter als Zahl und ein Geschlecht verfügen. Dieses Geschlecht ist entweder `männlich` oder `weiblich`, aber allgemein nicht erforderlich, da es optional ist (`null`).

Listing 7.1 Einfache Schemadefinition eines Steckbriefs einer Person für Apache Avro.

```
{
  "type": "record",
  "name": "Person",
  "aliases": ["Mensch"],
  "fields" : [
    {"name": "name", "type": "string"},
    {"name": "alter", "type": "int"},
    {"name": "geschlecht", "type": [null, "männlich", "weiblich"]}
  ]
}
```

Direkt angegebene Schemas werden von der Service-Plattform eingelesen und an die *Schema Registry*[8] von Confluent weitergegeben. Die Schema-Registry dient von da an als zentrale Schema-Verwaltung und ermöglicht Schema-Evolutionen sowie den Vergleich unterschiedlicher Schemas. Sie erlaubt die Identifizierung einzelner Schemas über eine URL, welche als zweite Möglichkeit in den Service-Definitionen angegeben werden kann. Die Plattform nutzt deren Funktionen für die typsichere Verknüpfung von Services in der Benutzeroberfläche sowie für die Fixierung eines Kommunikationskanals auf einen spezifischen Typ.

7.6 Generischer Maschinen-Service

Mit einem generischen Maschinen-Service können Anwender in der Nutzeroberfläche allein mit der Adresse des OPC-UA-Servers einer Werkzeugmaschine und der abzufragenden Datenquellen die Anbindung an eine Maschine herstellen. Hierfür wird ein mit *Eclipse Milo*[9] entwickelter Service angewiesen, die Daten aus den konfigurierten Datenquellen zu entnehmen. In der Benutzeroberfläche wird in einem Dialog im Hintergrund die Service-Definition mitsamt Konfiguration und typsicheren Schnittstellen generiert und in die Anwendung eingelesen, ohne dass der Benutzer mit weiteren technischen Details in Berührung kommt. Aufgrund der dynamischen Konfiguration des Services können die stark typisierten Maschinendaten von Eclipse-Milo allerdings nicht im richtigen Typen empfangen und weitergegeben werden, da hierfür notwendige Informationen zur Konvertierung fehlen. Deshalb ist der Maschinen-Service zum aktuellen Stand darauf beschränkt, reine Text-Werte zurückzugeben.

[8] Schema Registry, https://docs.confluent.io/current/schema-registry/docs/index.html
[9] Eclipse Milo, https://projects.eclipse.org/projects/iot.milo

8 Validierung

Im Folgenden werden die Konzepte anhand der prototypischen Implementierung mit hierfür geeigneten Szenarien validiert. Hierbei wird allerdings nur die allgemeine Service-Plattform und der generische Maschinen-Service in Betracht gezogen, da die Zuverlässigkeit und Skalierbarkeit nicht im Rahmen dieser Arbeit implementiert werden konnten.

8.1 Validierung mit Demo-Services

Während der Entwicklung und zum Ende hin wurde die Funktionalität der Service-Plattform mittels verschiedener kleinerer Services getestet, die zum isolierten Test der einzelnen Anforderungen und Komponenten sowie für die Validierung von Service-Kompositionen dienen. Die Services wurden extra für diesen Zweck entwickelt und haben somit wenig praktischen Bezug zur Einsetzbarkeit in einer Produktionsumgebung, ermöglichen allerdings eine technische Validierung der Plattform. Mit diesen Services wurde die gesamte geforderte Funktionalität von der Entwicklung über die Integration in die Plattform bis zum Deployment verfolgt.

8.1.1 Service-Entwicklung

Zur Validierung der Services und des Service-Cores wurden verschiedene Services entwickelt. Um den für JVM-kompatible Sprachen entwickelten Service-Core nutzen zu können, wurden hierbei Java und Kotlin als Programmiersprachen gewählt. Prinzipiell können allerdings auch annähernd beliebige andere Programmiersprachen genutzt werden, insofern sie in einem Docker-Container ausgeführt werden können. Falls für eine Programmiersprache kein Service-Core vorhanden ist, muss dieser direkt die Abfrage er Umgebungsvariablen zur Konfiguration und die Anbindung an Apache Kafka implementieren. Hierfür existieren eine Vielzahl verschiedener Kafka-Clients[1] in mehreren Programmiersprachen — sowohl hardwarenah in C/C++ als auch für Sprachen wie Node.js. Falls also weder ein Service-Core noch ein Kafka-Client für eine Technologie vorhanden sind, ist die Nutzung eines Workarounds erforderlich, wie er im weiteren Validierungsszenario in diesem Kapitel beschrieben ist (siehe Abschnitt 8.2). Falls im sehr unwahrscheinlichen Fall eine Technologie nicht unter Linux bzw. Docker genutzt werden kann, ist sie nicht in Kombination mit der Service-Plattform nutzbar.

Mit einem Service zum Generieren zufälliger Werte wurde die allgemeine Funktionalität eines Services, die Konfigurierbarkeit über die Umgebungsvariablen und die Sendefunktionalität getestet. Ein Anzeige-Service kann beliebige Werte empfangen und diese in einer Weboberfläche darstellen, sodass die Empfangsfunktionalität mit der dazugehörigen Konfiguration als auch

[1] Apache Kafka – Clients, https://cwiki.apache.org/confluence/display/KAFKA/Clients

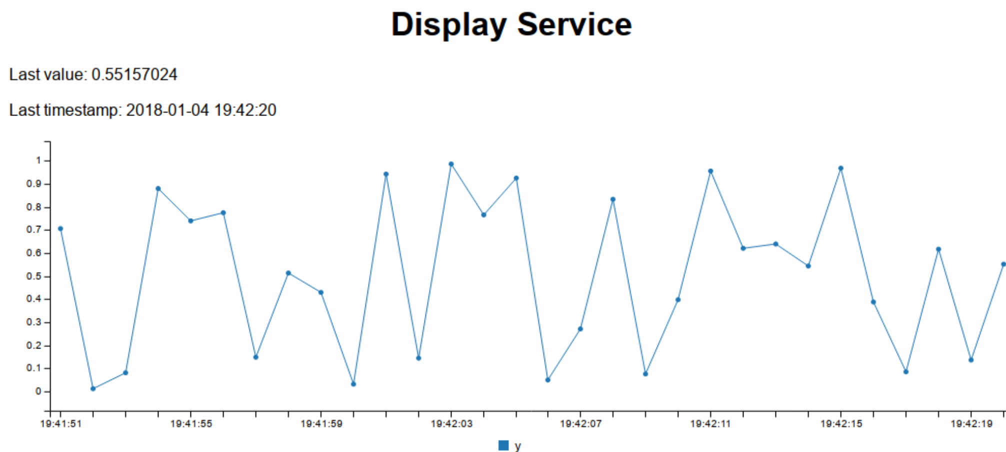


Abbildung 8.1: Ein einfacher Display-Service empfängt Zahlenwerte und visualisiert für die Anwender in einer Weboberfläche.

die Zugänglichkeit eines Services von außen getestet werden kann (vgl. Abbildung 8.1). Da die Entwicklung des Service-Cores parallel mit diesen Services stattfand und getestet wurde, entfiel hierfür der Integrationsaufwand. Für den Aufruf der Weboberfläche war es erforderlich, dass bei der jeweiligen Cloud-Plattform die Port-Freigabe des für die Oberfläche verwendeten Ports freigeschaltet ist, damit diese von außerhalb des internen Cloud-Provider-Netzwerks aufgerufen werden kann. Da die Port-Freigabe allerdings nicht aus der Service-Plattform gesteuert werden kann, war es notwendig, dem Service einen vom Anwender freigegebenen Port über die Konfiguration mitzuteilen. Durch Rancher hierdurch der konfigurierte Port auf dem jeweiligen Host geöffnet und mit dem Service verbunden.

8.1.2 Kommunikation

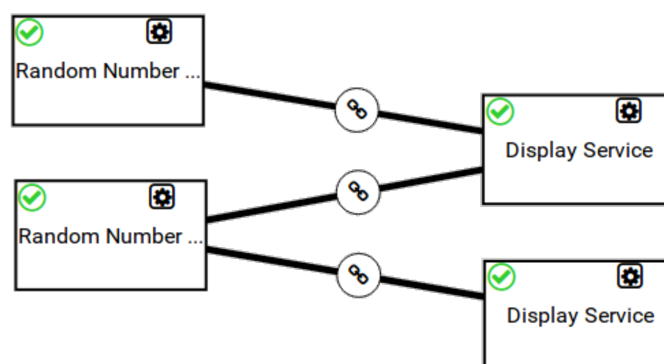


Abbildung 8.2: Der Aufbau der Service-Orchestrierung, mit der die Plattform validiert wurde.

Mit diesen Services konnten die verschiedenen Szenarien zum Nachrichtenaustausch aus Abschnitt 4.3.2 getestet werden. So wurde wie in Abbildung 8.2 die direkte Kommunikation von einem Service an einen anderen (bspw. links oben nach rechts oben), von einem an mehrere (links

unten nach rechts) und von mehreren an einen (links nach rechts oben) erfolgreich getestet. Über die Kommunikationskanäle wurden hierbei Nachrichten mit verschiedenen Typen ausgetauscht. In diesen Szenarien fand die Kommunikation in verschiedenen Aufbauten mit vergleichsweise wenigen Services und einer hoher Nachrichtenfrequenz verlustfrei und unmittelbar statt. Aufgrund der fehlenden Ressourcen konnten allerdings keine Auswertungen mit einer großen Anzahl an Services oder Verbindungen zwischen Services vorgenommen werden, also keine Aussagen über das Verhalten bei einer großen Auslastung getroffen werden können.

Das Konzept hat vorgesehen, dass für einzelne Orchestrierungen die Infrastrukturservices zur Kommunikation ausgewählt oder auf eine bestimmte Cloud deployt werden können. Diese Anforderung wurde nicht umgesetzt, sodass nur eine einzige Instanz der Kommunikationsservices genutzt werden kann, die außerdem zuvor manuell deployt und konfiguriert werden muss.

8.1.3 Deployment

Die Services wurden anschließend auf die Plattformen unterschiedlicher Cloud-Provider deployt, zu denen auch lokale virtuelle Maschinen und ein OpenStack-Cluster gehörten. Verschiedene Service-Konstellationen aus dem vorigen Abschnitt wurden unterschiedlich auf Cloud-Providern verteilt, um damit die Kommunikation zwischen verschiedenen Cloud-Plattformen testen zu können. Die Deployments je nach vom Nutzer ausgewählter Cloud sowie die Kommunikation mit allen Komponenten liefen hierbei fehlerfrei über das von Rancher aufgespannte Netzwerk ab, in welchem sich alle Hosts befinden. Die Clusterbildung aus einzelnen Cloud-Hosts sowie die Möglichkeit, die Hosts nach Standort zu gruppieren, sind nicht implementiert worden und konnten deshalb nicht validiert werden.

8.1.4 Benutzeroberfläche

Die oben durchgeführten Aktionen wurden allesamt über die Benutzeroberfläche ausgeführt. Über sie kann ein Anwender im Prinzip ohne technisches Wissen Services in die Plattform einbinden, sie miteinander verknüpfen und sie auf verschiedene Cloud-Plattformen deployen. Der Anwender muss allerdings die für die Kommunikation notwendigen Konfigurationsdaten eintragen sowie die einzelnen virtuellen Maschinen in der Anwendung hinterlegen, was ein technisches Hintergrundwissen erfordert. In verschiedenen Ansichten muss der Nutzer zuerst die Plattform mit den Cloud-Providern, Services sowie der Konfiguration pflegen, bevor die eigentliche Orchestrierung angelegt und deployt werden kann. Aufgrund fehlender Wizards oder unterstützenden Tooltips ist die Anwendung allerdings nicht direkt von Laien nutzbar, erfährt allerdings über Benachrichtigungen den Fortschritt seiner Aktionen. Das Monitoring der Hosts inklusive nicht-verfügbarer Maschinen funktionierte oberflächlich, ließ allerdings keine direkten Rückschlüsse über die betroffene Maschine zu.

8.2 Validierung an exemplarischem Fließbandaufbau

Im Rahmen einer anderen Masterarbeit wurden Anwendungen für einen Fließbandaufbau mit mehreren Sensoren und Aktoren entwickelt, die mit Maschinendaten und verschiedenen Auswertungsalgorithmen die Ursachen von fehlerhaften Produkten erkennen sollen. Mithilfe der

erkannten Ursachen sollen fehlerhafte Produkte frühzeitig aussortiert werden, um weitere unnötig anfallende Kosten bspw. für die Lagerung zu vermeiden. In Abbildung 8.3 laufen drei Produkte über das Fließband (Würfel, Pyramide, Kugel) an mehreren Sensoren (links und mittig) vorbei, die unter anderem deren Form und Farbe erkennen. Die Anwendungen konnten nun auf eigenen Maschinen mit einer definierten Umgebung ausgeführt werden, um jeweils Daten auszulesen, auszuwerten und anzuzeigen. In den einzelnen Services waren hierbei die Adressen der anderen Services für die Kommunikation hart kodiert. Die Ergebnisse der Auswertungen entschieden daraufhin, ob die Produkte aktuell die Anforderungen eines fehlerfreien Produkts erfüllen, oder ob sie vom weiteren Verarbeitungsprozess ausgeschlossen werden sollten (Schranke).

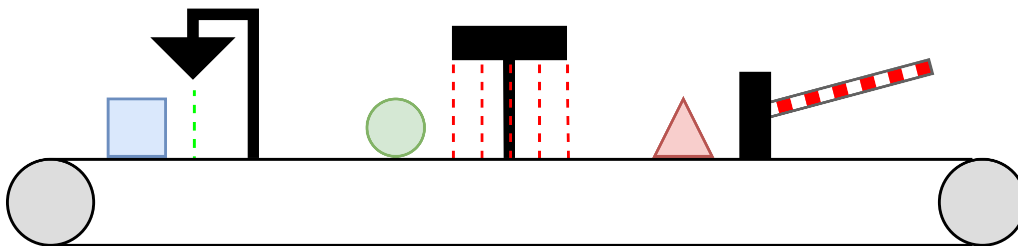


Abbildung 8.3: Mit den Sensoren eines Fließbands verbundene Services entscheiden anhand von in der Cloud berechneter Werte über den Weiterverlauf der Produkte.

Um die Vorteile des Cloud-Computings nutzen zu können und um die einzelnen Anwendungen zu entkoppeln und hiermit skalierbarer zu gestalten, sollte die Service-Plattform verwendet werden, um sie als Services zu verwalten. Hierfür war es zu Beginn notwendig, die in der Programmiersprache *R* geschriebenen Anwendungen in Docker-Container nutzen zu können und darin die R-Umgebung mitsamt allen notwendigen Abhängigkeiten aufzubauen. Anschließend fand die Integration in die Service-Plattform statt, damit die Services über die Kommunikationsschicht Daten untereinander austauschen können. Obwohl aus R durch die Programmbibliothek *rkafka*[2] mit Kafka kommuniziert werden kann, sollte die Integration unter Annahme eines fehlenden Service-Cores sowie eines fehlenden Kafka-Clients vorgenommen werden. Hierfür wurde der Service-Core und die R-Anwendungen erweitert, um empfangene Nachrichten im Dateisystem ablegen und auslesen zu können. Anschließend musste händisch eine Service-Definition angelegt werden, sodass die neuen Services in die Plattform eingebunden werden konnten. Die Services wurden daraufhin in einer Komposition verknüpft, konfiguriert und in der Cloud deployt. Hierbei wurde gezeigt, dass die Integration von Anwendung vorhandener Anwendungen trotz eines fehlenden Service-Cores sowie Kafka-Clients mit geringem Aufwand vorgenommen werden kann.

8.3 Validierung der Maschinen-Services

Zur Validierung des generischen Maschinen-Services sollte nach Konzept ein Service entwickelt werden, der sich mit einem OPC-UA-Server verbinden und Daten typsicher abrufen kann. Hierfür sollte der Anwender in der Nutzeroberfläche die Datenquelle und den Datentyp für die sichere Weitergabe an andere Services bestimmen. Bei der Implementierung hat sich allerdings herausgestellt,

[2]rkafka, https://cran.rstudio.com/web/packages/rkafka/index.html

dass das Konzept in der Hinsicht unvollständig ist, dass die Informationen, um den OPC-UA-Wert in den angegebenen Datentypen umzuwandeln, fehlen. Einfache Daten können so zwar generisch in Textform ausgegeben werden, was allerdings bei komplexen Datenstrukturen nicht mehr möglich ist. Da der OPC-UA-Client außerdem zum Senden die Werte mit den korrekten Datentypen benötigt, ist das Senden an einen OPC-UA-Server nicht möglich, da ebenfalls Informationen zur Konvertierung vom mit Avro spezifizierten Typen in den OPC-UA-Typen fehlen.

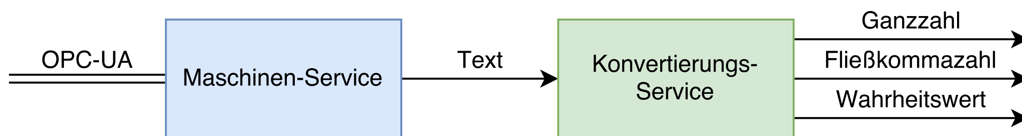


Abbildung 8.4: Mit einem zusätzlichen Service können die Textwerte zur weiteren Verwendung nach Möglichkeit in andere Typen konvertiert werden.

Um dennoch die Anbindung an eine Werkzeugmaschine validieren zu können, wurde ein weiterer Service entwickelt, der Werte in Textform an einem Eingang annimmt und in verschiedenen Typen als Zahl oder Wahrheitswert über verschiedene Eingänge ausgibt. Dieser Konvertierungs-Service (vgl. Abbildung 8.4) nimmt als eine Art Nachrichten-Mapper einen Textwert entgegen und versucht ihn daraufhin in verschiedene andere Typen wie eine Zahl oder einen Wahrheitswert umzuwandeln, damit diese mit dem korrekten Typen von anderen Services empfangen werden können. So wurden vom Maschinen-Service Zahlenwerte über OPC-UA abgerufen, in Textform an den Konvertierungs-Service ausgegeben und anschließend wieder als Zahl an den Display-Service gereicht. Über den Display-Service konnte daraufhin die (eingeschränkte) Funktion des Maschinen-Services validiert werden.

9 Zusammenfassung und Ausblick

Industriebetriebe wollen im Rahmen der Wandlung zur Industrie 4.0 ihre Produktion weiter digitalisieren und die Vorteile der in die Cloud verlagerten Prozesse voll ausschöpfen. Mithilfe intelligenter Systeme soll nun die Automatisierung einzelner Teilprozesse auf der nächsten Ebene fortgeführt werden, um flexibler, individueller und vor allem effektiver zu produzieren. Hierfür werden jedoch Plattformen benötigt, mit denen diese Teilprozesse verknüpft und die Produktion in vollem Umfang abgebildet und somit intelligenter werden kann. Solche Plattformen müssen bereits an den Datenquellen der Sensoren von Werkzeugmaschinen ansetzen und bis zu einem Konzept zur Verwendung von Cloud-Ressourcen reichen, um weiter die Standort- und Kostenvorteile im Multi-Cloud-Szenario nutzen zu können.

Teile dieser Anforderung wurden bereits in verwandten Arbeiten angegangen, jedoch unvollständig oder mit einem anderen Fokus und Zweck verfolgt. Die in letzter Zeit angekündigten Plattformen für das Industrial-Internet-of-Things verfolgen zwar einen ähnlichen Ansatz, allerdings großteils ohne Resultate und ohne eine Cloud-Affinität zu zeigen. In dieser Masterarbeit wurde als Teil des Forschungsprojekts MultiCloud eine Plattform konzipiert, die jedoch genau diese Anforderungen aufgreift und eine Produktion der Industrie 4.0 unterstützen soll. Mit einem Konzept für den Aufbau von Services und deren Entwicklung wird hierbei die Grundlage geschaffen, um einzelne Teilprozesse und -aufgaben in der Plattform abzubilden. Anschließend wurde konzipiert, wie die Verknüpfung von einzelnen Services aussieht und sicher stattfinden kann, um leichtsinnige, aber fatale Fehler besser vermeiden zu können. Die resultierenden Orchestrierungen sollen anschließend von einem Anwender auf verschiedenen Cloud-Plattformen ausfallsicher betrieben werden können, während er ohne Ausfall des gesamten Systems einzelne Teilprozesse abändern und an die veränderten Anforderungen anpassen kann. Mit dem Ziel der Unterstützung mehrerer Cloud-Anbieter wird hierbei außerdem ein Vendor-Lock-in vermieden, der die Nutzung von Kosten- und Standortvorteilen verhindert hätte. Diese komplexen Aufgaben sollen zusätzlich noch von einem technisch unerfahrenen Anwender gesteuert und ohne komplexes Hintergrundwissen der einzelnen Prozesse durchgeführt werden können.

In der Validierung hat sich abschließend in einem künstlichen und auch einem realistischen Szenario mit einfachen Maschinen die Tauglichkeit der konzipierten Service-Plattform gezeigt. Sowohl die Neuentwicklung von Services als auch die Migration vorhandener Anwendungen ist den Service-Entwicklern mit verschiedenen Technologien möglich. Über einen Service-Core können hierbei die Konfiguration und Verbindung zu anderen Services mit geringem Aufwand und niedriger Kopplung vorgenommen werden. Die Kommunikation der Services funktioniert mit den unterschiedlichsten Datentypen typsicher und ist durch den Einsatz bekannter Messaging-Patterns wie eines Nachrichten-Mappers ausbaubar. Durch den inhärent vermiedenen Vendor-Lock-in auf eine Cloud-Plattform können die Service-Orchestrierungen abschließend von Anwendern flexibel, auf mehrere Clouds verteilt, deployt werden. Mit einer direkten Anbindung an Werkzeugmaschinen aus der Nutzeroberfläche heraus, wird somit die Möglichkeit gegeben, auch Teilprozesse auf unterster Ebene ansetzen zu lassen und in einer komplexen Prozesskette zu verknüpfen.

9.1 Ausblick

In der Validierung hat sich allerdings auch gezeigt, dass es einige Bereiche gibt, welche in zukünftigen Arbeiten weiter ausgearbeitet und verbessert werden müssen.

9.1.1 Implementierung und Validierung fehlender Plattform-Konzepte

Einige im Konzept spezifizierte Anforderungen konnten nicht mehr im Rahmen dieser Masterarbeit umgesetzt und validiert werden. Die Möglichkeit, die Ressourcen von Cloud-Plattformen in Clustern bspw. nach Standort oder einer anderen Eigenschaft zu gruppieren, wurde nicht weiter in der Implementierung erfolgt und bietet vor allem auch mit Hinblick auf die dynamische Provisionierung von neuen Ressourcen im Fall einer Skalierung interessante Ansatzpunkte. Hierzu gesellen sich ebenfalls die Herausforderungen, eine oder mehrere Kommunikationsplattformen mit einer Service-Komposition zu konfigurieren und zu deployen, um vor allem datenschutzrechtliche Gesichtspunkte oder auch den Datendurchsatz weiterverfolgen zu können. Ebenso soll die Benutzeroberfläche der Anwendung mit realistischen Szenarien von Anwendern in einer Studie evaluiert werden, um hier weitere Ansatzpunkte zur Verbesserung der Benutzbarkeit und der Führung durch die einzelnen Prozesse zu erhalten.

9.1.2 Neukonzeption des generischen Maschinen-Services

Bei der Implementierung und der anschließenden Validierung des generischen Maschinen-Services hat sich gezeigt, dass die Komplexität hierbei deutlich unterschätzt wurde. Zwar war es möglich mit einer Werkzeugmaschine zu kommunizieren, allerdings nur lesend und mit einer Einschränkung an die Struktur der ausgetauschten Daten. Der Maschinen-Service als Schnittstelle zwischen zwei typsicheren Kommunikationsschichten kann hierbei die Abbildung der einzelnen Datentypen nicht korrekt vornehmen, sodass die Konfiguration und Steuerung von Werkzeugmaschinen überhaupt nicht validiert werden konnte.

9.1.3 Implementierung und Validierung der Zuverlässigkeit & Skalierbarkeit

Aufgrund der Komplexität der übrigen Anforderungen konnte das Konzept zum zuverlässigen Betrieb von Services sowie deren Skalierbarkeit nicht in voller Tiefe ausgearbeitet und anschließend überhaupt nicht implementiert und validiert werden. Für den Betrieb einer Service-Plattform in einer realistischen produktionsnahen Umgebung ist es allerdings notwendig, diese Anforderungen zu unterstützen. Die Steuerung von Produktionsprozessen muss maximal zuverlässig sein, um den teuren Stillstand von Werkzeugmaschinen zu vermeiden. Gleichzeitig sollen aber auch alle vorhandenen Ressourcen eines Betriebs genutzt werden können und nicht durch die zu starke Auslastung einzelner Services ausgebremst werden.

9.1.4 Evaluation in praxisnahen Szenarien

Mit der Validierung an einem exemplarischen Fließband mit mehreren Sensoren und Aktoren in Abschnitt 8.2 wurde ein praxisnahes Szenario in den Grundzügen bereits nachgestellt und erfolgreich ausgeführt. Die hierbei verwendeten Maschinen waren allerdings nur kaum mit echten Industriemaschinen vergleichbar, die bereits einzeln mehrere Tausend Sensoren und Aktoren besitzen und einem komplexen Arbeitsprozess verbaut sein können. Die in die Service-Plattform integrierten Services hatten keinen Fokus auf Maschinennähe, sondern nutzten nur deren Daten. Es bleibt zum jetzigen Stand also offen, wie sich wirklich maschinen- und hardwarenahe Services verhalten und auch, wie mit Echtzeitanforderungen der Werkzeugmaschinen umgegangen werden soll. Von zentraler Bedeutung ist daher auch, wie sich die Plattform unter Lasttests mit einer großen Anzahl an Services, Verbindungen zwischen Services oder einem hohen Datenvolumen verhält.

9.1.5 Dynamische Provisionierung beim Deployment

Bei allen Deployment-relevanten Prozessen sowie bei der Skalierbarkeit wird bisher angenommen, dass die vorhandenen Cloud-Hosts in einer konstanten, ausreichenden Menge vorhanden sind, was allerdings unabhängig von der Nutzung die gleichen maximalen Kosten bedeuten würde. In einer weiteren Arbeit kann hierbei erforscht werden, wie die genutzten Ressourcen auf ein Minimum beschränkt und ungenutzte Ressourcen vermieden werden können. Hierbei muss zum einen beachtet werden, dass während eines Deployment-Prozesses erst die Notwendigkeit zusätzlicher Ressourcen oder deren Abbau bestehen kann. Weiterhin kann auch eine Skalierung einzelner Services neue Ressourcenanforderungen ergeben, welche anschließend eine hohe Anzahl kaum genutzter Ressourcen zurücklassen kann. Darauf aufbauend kann auch ein Konzept entworfen werden, wie eine Kompaktierung auf wenige, aber gut ausgelastete Ressourcen stattfinden kann, während gleichzeitig eine hohe Verfügbarkeit und die Erfüllung von Anforderungen an bspw. den Standort gewährleistet werden kann.

Literaturverzeichnis

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA - A Runtime for TOSCA-Based Cloud Applications“. In: *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings.* Hrsg. von S. Basu, C. Pautasso, L. Zhang, X. Fu. Bd. 8274. Lecture Notes in Computer Science. Springer, 2013, S. 692–695. DOI: 10.1007/978-3-642-45005-1_62 (zitiert auf S. 29).

[BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Vinothek - A Self-Service Portal for TOSCA“. In: *Proceedings of the 6th Central-European Workshop on Services and their Composition, ZEUS 2014, Potsdam, Germany, February 20-21, 2014.* Hrsg. von N. Herzberg, M. Kunze. Bd. 1140. CEUR Workshop Proceedings. CEUR-WS.org, 2014, S. 69–72. URL: http://ceur-ws.org/Vol-1140/paper11.pdf (zitiert auf S. 29).

[BIS+14] A. Brogi, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, E. Pimentel, F. D’Andria. „SeaClouds: A European Project on Seamless Management of Multi-Cloud Applications“. In: *ACM SIGSOFT Software Engineering Notes* 39.1 (2014), S. 1–4. DOI: 10.1145/2557833.2557844 (zitiert auf S. 32–34).

[CCMW01] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C Note. World Wide Web Consortium, März 2001. URL: http://www.w3.org/TR/wsdl (zitiert auf S. 27).

[CFH+98] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, A. L. Wolf. *A characterization framework for software deployment technologies.* Techn. Ber. COLORADO STATE UNIV FORT COLLINS DEPT OF COMPUTER SCIENCE, 1998 (zitiert auf S. 23).

[Chi17] A. Chifor. *Container Orchestration with Kubernetes: An Overview.* 30. Mai 2017. URL: https://medium.com/onfido-tech/container-orchestration-with-kubernetes-an-overview-da1d39ff2f91 (zitiert auf S. 30).

[CMRW07] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language.* World Wide Web Consortium, Recommendation REC-wsdl20-20070626. Juni 2007 (zitiert auf S. 27).

[Cor] CoreOS. *rkt vs Other Projects.* URL: https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html (zitiert auf S. 20).

[Cur04] E. Curry. „Message-Oriented Middleware“. In: *Middleware for Communications.* Hrsg. von Q. H. Mahmoud. Chichester, England: John Wiley und Sons, 2004. Kap. 1, S. 1–28. ISBN: 978-0-470-86206-3. DOI: 10.1002/0470862084.ch1. URL: http://www.edwardcurry.org/publications/curry_MfC_MOM_04.pdf (zitiert auf S. 20–23).

[Die14] J. Diemer. „Sichere Industrie 4.0-Plattformen auf Basis von Community-Clouds“. In: *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung · Technologien · Migration*. Hrsg. von T. Bauernhansl, M. ten Hompel, B. Vogel-Heuser. Wiesbaden: Springer Fachmedien Wiesbaden, 2014, S. 369–396. ISBN: 978-3-658-04682-8. DOI: 10.1007/978-3-658-04682-8_19 (zitiert auf S. 36).

[Doc17] Docker Inc. *What is a container*. 2017. URL: https://www.docker.com/what-container (zitiert auf S. 16).

[DYDZ09] Y.-S. Dai, B. Yang, J. Dongarra, G. Zhang. „Cloud service reliability: Modeling and analysis“. In: *15th IEEE Pacific Rim International Symposium on Dependable Computing*. 2009, S. 1–17 (zitiert auf S. 36).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. ISBN: 978-3-7091-1567-1. DOI: 10.1007/978-3-7091-1568-8 (zitiert auf S. 13, 14, 64).

[GSR13] P. Gupta, A. Seetharaman, J. R. Raj. „The usage and adoption of cloud computing by small and medium businesses“. In: *International Journal of Information Management* 33.5 (2013), S. 861–874 (zitiert auf S. 37).

[ISO10] ISO/IEC. *ISO/IEC 25010 System and software quality models*. Techn. Ber. 2010 (zitiert auf S. 26).

[LM06] S.-H. Leitner, W. Mahnke. „OPC UA - Service-oriented Architecture for Industrial Applications“. In: *Softwaretechnik-Trends* 26 (2006) (zitiert auf S. 24, 25).

[LS17] R. Langmann, M. Stiller. „Industrial Cloud – Status und Ausblick“. In: *Industrie 4.0: Herausforderungen, Konzepte und Praxisbeispiele*. Hrsg. von S. Reinheimer. Wiesbaden: Springer Fachmedien Wiesbaden, 2017, S. 29–47. ISBN: 978-3-658-18165-9. DOI: 10.1007/978-3-658-18165-9_3 (zitiert auf S. 35).

[Mer14] D. Merkel. „Docker: Lightweight Linux Containers for Consistent Development and Deployment“. In: *Linux J.* 2014.239 (März 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (zitiert auf S. 17, 18, 20).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Techn. Ber. Gaithersburg, MD, United States, 2011 (zitiert auf S. 13, 15).

[Nan15] N. Nannoni. *Message-oriented middleware for scalable data analytics architectures*. 2015 (zitiert auf S. 51).

[Neu94] B. C. Neuman. „Scale in Distributed Systems“. In: *Readings in Distributed Computing Systems*. IEEE Computer Society Press, 1994, S. 463–489 (zitiert auf S. 26).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. 25. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 28).

[OAS14] OASIS. *Cloud Application Management for Platforms Version 1.1*. 9. Nov. 2014. URL: http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html (zitiert auf S. 32).

[OAS16] OASIS. *TOSCA Simple Profile in YAML Version 1.1*. 25. Aug. 2016. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/csprd01/TOSCA-Simple-Profile-YAML-v1.1-csprd01.html (zitiert auf S. 28, 45).

[OPC16] OPC Foundation. *OPC Unified Architecture – Wegbereiter der 4. industriellen (R)Evolution.* Mai 2016. URL: https://opcfoundation.org/wp-content/uploads/2016/05/OPC-UA-Interoperability-For-Industrie4-and-IoT-DE-v5.pdf (zitiert auf S. 24, 25).

[Pah15] C. Pahl. „Containerization and the PaaS Cloud". In: *IEEE Cloud Computing* 2.3 (Mai 2015), S. 24–31. ISSN: 2325-6095. DOI: 10.1109/MCC.2015.51 (zitiert auf S. 16, 17).

[PKM12] R. S. M. L. Patibandla, S. S. Kurra, N. B. Mundukur. „A Study on Scalability of Services and Privacy Issues in Cloud Computing". In: *Distributed Computing and Internet Technology: 8th International Conference, ICDCIT 2012, Bhubaneswar, India, February 2-4, 2012. Proceedings.* Hrsg. von R. Ramanujam, S. Ramaswamy. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 212–230. ISBN: 978-3-642-28073-3. DOI: 10.1007/978-3-642-28073-3_19 (zitiert auf S. 37).

[Ran] Rancher. *Overview of Rancher - Rancher Documentation.* URL: https://rancher.com/docs/rancher/v1.6/en/ (zitiert auf S. 31).

[Sha16] G. Shapira. *Confluent Enterprise Reference Architecture.* 7. Okt. 2016 (zitiert auf S. 66).

[VRB11] L. M. Vaquero, L. Rodero-Merino, R. Buyya. „Dynamically scaling applications in the cloud". In: *Computer Communication Review* 41.1 (2011), S. 45–52. DOI: 10.1145/1925861.1925869 (zitiert auf S. 26, 37, 65).

Alle URLs wurden zuletzt am 07.01.2018 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift