Institut für Softwaretechnologie

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Konzeption und Realisierung einer dynamischen Service Plattform zur Verbindung von internen und externen Komponenten

Eduard Marbach

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer/in:	Dr. rer. nat. Asim Abdulkhaleq, Timur Tasci, M.Sc.
Beginn am:	10. Juli 2017
Beendet am:	21. Februar 2018

Kurzfassung

In der Zeit der Digitalisierung und der Industrie 4.0 nehmen die verfügbaren Daten aus der Produktion und den Maschinen einen immer wichtigeren Standpunkt ein. Die Daten können wertvolle Informationen zum Zustand der Maschinen, der Effizienz des Prozesses oder Engpässen im System aufdecken. In vielen Fabriken bleiben diese Daten ungenutzt und werden nicht verwertet. Es mangelt an einer Möglichkeit zur herstellerübergreifenden Integration von Maschinen und Services, damit diese verbunden und anschließend analysiert werden können. Durch die großen Datenmengen, die Maschinen kontinuierlich produzieren, muss eine ausreichende Rechenkapazität zur Verarbeitung der Daten verfügbar sein. Daher bietet es sich an Cloud Computing und Cloud Dienste zu integrieren.

Dieser Mangel an einer effizienten Möglichkeit zur vollständigen Abbildung eines Prozesses inklusive Maschinen, Analyse- und Überwachungsservices soll in einer Plattform abgebildet werden. Die verbreiteten Technologien und Möglichkeiten der Cloud sollen in dieser Plattform gebündelt und genutzt werden. Der Nutzer soll mit wenig Konfiguration eine Prozesskette abbilden können, mit der die Maschinendaten verwertet werden und einen Mehrwert liefern können. Maschinen und Services sollen sich mithilfe einer modernen Oberfläche verknüpfen lassen, um den Datenfluss abzubilden. Neben internen Services, die mit in der Plattform integriert werden, soll eine Möglichkeit zur Einbindung von externen Diensten möglich sein. Dazu zählen laufende Datenbanken, REST-Schnittstellen oder andere Kommunikationsframeworks. Die internen Services sollen dabei von Service-Entwicklern beigetragen werden. Zur Erleichterung der Integration wird eine einfache Schnittstelle bereitgestellt, mit der die benötigte Integrationszeit minimal sein soll. Mit einer anschließenden Validierung wird der Integrationsaufwand, die Funktionalität und Nutzbarkeit der Plattform geprüft.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

1 Einleitung

Der digitale Wandel breitet sich in allen Bereichen des Lebens aus. Sei es im personellen Computerumfeld, im Haushalt und weiteren Bereichen. Ein wichtiges Gebiet, das von der Digitalisierung profitieren kann, ist die Industrie und der Maschinenbau. Hier lassen sich die bestehenden Fertigungsprozesse und Arbeitsschritte durch die Vorteile von digitalen Komponenten vielseitig optimieren. Ein bekannter Begriff hierfür ist *Industrie 4.0* [BTV14]. Dieser steht für die vierte industrielle Revolution, wobei der Fokus auf der Digitalisierung mit Themen wie intelligenter Robotik, dem Internet der Dinge und Smart Factories [BTV14] einhergeht. Der Fortschritt bei der Automatisierung und selbstständigen Arbeit der Maschinen wird mit einer Steigerung der Produktivität von bis zu 30% geschätzt.

Mit dem Fokus auf der Digitalisierung und der Unterstützung von beliebigen Cloud Anbietern ist das Forschungsprojekt MultiCloud [Ins15] entstanden. Das Ziel des Forschungsprojekts mit mehreren Firmenpartnern ist es, die Erkenntnisse aus dem Cloud Computing und den bestehenden Kenntnissen der führenden Anbieter wie Amazon AWS, Microsoft Azure oder Google Cloud mit der maschinellen Produktion zu vereinen. Dabei sollen die unterschiedlichen Komponenten, Sensoren und Aktoren von Maschinen teilweise mit der Cloud in Verbindung gebracht werden und die Daten entsprechend für nachfolgende Prozessschritte zur Verfügung gestellt werden.

Mithilfe der bestehenden Möglichkeiten zur Verknüpfung der Maschinen mit der Cloud und dem World Wide Web entsteht eine riesige Quelle mit verwertbaren Daten. Jedoch wird diese Menge an verfügbaren Daten aus unterschiedlichsten Sensoren und Steuerungskomponenten nicht weiter verwertet. Die Daten der Maschinen können genutzt und ausgewertet werden. Um bspw. defekte Teile im Vorfeld aussortieren zu lassen. Durch Überwachen von Grenzwerten können fehlerhafte Teile frühzeitig erkannt und entsprechend gehandelt werden. Es ist nicht notwendig einen Werkstoff in einer Produktionskette weiter zu bearbeiten, wenn bereits zu Beginn der Kette fehlerhafte Maße festgestellt werden. Hier bietet sich bereits einiges an Potential zur Optimierung an. Ein weiterer wichtiger Aspekt ist die frühzeitige Verschleißerkennung von Maschinenteilen. Hier könnten beispielsweise verschiedene Sensoren das Drehmoment oder die Temperatur überwachen und bei Grenzwertüberschreitungen eine Warnmeldung auslösen. Diese Verbesserungen im Prozess und der Maschinenwartung resultieren in einer Umsatzoptimierung der Fabrik. Dies hängt damit zusammen, dass ungewollte Stillstandzeiten der Maschinen reduziert werden und die Produktionsqualität verbessert wird.

1.1 Zielsetzung

Diese Arbeit wird sich mit der Konzeption eines geeigneten Serviceformats mit der dazugehörigen Schemadefinition beschäftigen. Für die Kommunikation ist es wichtig ein fest definiertes Format für die Service-Struktur zu haben. Ein anderer wichtiger Punkt ist die maschinelle Verarbeitung

der Struktur, die dadurch möglichst flexibel und einfach gehalten werden muss. Dabei werden zuvor die benötigten Anforderungen an einen Service diskutiert und anschließend beschrieben. Die Verknüpfung von mehreren Services und der Gewährleistung einer reibungslosen Kommunikation bildet einen wichtigen Teil in einem Produktionsprozess. Neben der Festlegung der Kommunikationsart müssen die Services das Nachrichtenformat im Vorfeld kennen, um die Daten korrekt verarbeiten zu können. Es ist nicht nur wichtig die Strukturen der Services zu definieren, sondern auch ein Format für die Nachrichten festzulegen.

Ausgehend aus den nun definierten Beschreibungen für Services und deren Kommunikationsmitteln wird die Verknüpfung von internen und externen Services konzipiert. Interne Service werden selbst entwickelt und externe Services entsprechen laufenden Programmen, die integriert werden müssen. Es wird ein Konzept entwickelt, mit dem die Verknüpfung leicht und flexibel umgesetzt werden kann. Zum Schluss ist es wichtig die Sicherheit des gesamten Systems zu gewährleisten. Dafür wird das System in mehrere Schichten unterteilt und für jede Schicht wird ein Schutzmechanismus konzipiert.

Als Ergebnis entsteht eine Plattform, in der Services miteinander verbunden werden. Mit einer Service-Definition werden die Schnittstellen eines Services festgehalten und für die Verarbeitung genutzt. Für jeden Service kann dabei eine Cloud Instanz gewählt werden, auf die der Service deployed wird. Die Kommunikation wird über ein Schemaformat definiert, sodass ein Service für jede Schnittstelle ein konkretes Format liefern kann.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Hier werden die Grundlagen dieser Arbeit beschrieben, die für ein besseres Verständnis benötigt werden.

Kapitel 3 – Verwandte Arbeiten: In diesem Teil werden verwandte Arbeiten diskutiert, die einen ähnlichen Schwerpunkt bearbeitet haben. Dabei wird der wissenschaftliche und technische Stand untersucht und die Vor-/Nachteile der verschiedenen Arbeiten in Relation mit der gewünschten Zielsetzung gebracht werden.

Kapitel 4 – Konzept stellt die verschiedenen Ideen und Konzepte im Detail dar. Es werden unterschiedliche Ansätze ausgewertet und die bestmögliche Lösung zum Abschluss eines jeden Unterkapitels präsentiert werden.

Kapitel 5 – Umsetzung stellt eine mögliche Form der zuvor vorgestellten Konzeption dar. Dabei werden die benötigten Komponenten prototypisch umgesetzt und das Vorgehen erläutert.

Kapitel 6 – Validierung nutzt die zuvor entstandene Umsetzung zum Überprüfen der vorgenommen Zielsetzung.

Kapitel 7 – Zusammenfassung fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

Dieser Teil der Arbeit beschäftigt sich mit den fundamentalen Grundlagen, die für das Verständnis der weiteren Ausarbeitung mit Hinblick auf die Konzeption und Umsetzung wichtig sind. Die einzelnen Themen werden dabei nicht im gesamten Umfang ihres Bereichs erläutert, sondern nur soweit es für das Gesamtverständnis nötig ist.

2.1 Industrie 4.0

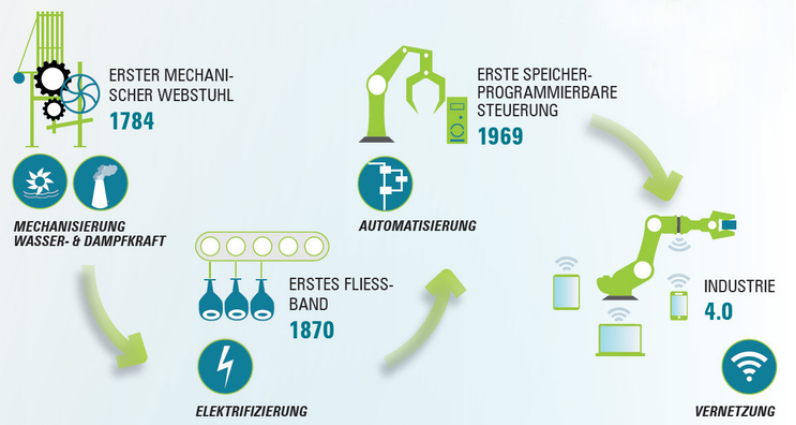

Abbildung 2.1: Die einzelnen Revolutionen der Industrie mit ihrem Schwerpunkt. [AG18]

Die Industrialisierung schreitet stets voran und durchläuft dabei immer wieder Revolutionen. Diese verändern das aktuelle Standbild grundlegend, wie in Abbildung 2.1 zu sehen ist. Die erste industrielle Revolution (1.0) entstand dabei durch die Mechanisierung der Industrie, gefolgt von der Massenfertigung durch Fließbänder (2.0). Anschließend wurde durch die Automatisierung die dritte Revolution (3.0) ausgelöst. Die vierte industrielle Revolution hat dabei die Digitalisierung und Verbindung der Maschinen und Fabriken mit den verfügbaren digitalen und informationstechnischen Mitteln im Vordergrund [WSCL13]. Die nachfolgenden beschriebenen Bereiche werden in dieser Arbeit teilweise umgesetzt.

Vernetzung Hierbei handelt es sich um Maschinen, Sensoren, Komponenten und andere verfügbare Teile der Industrie zusammenzubringen und eine Kommunikationsmöglichkeit herzustellen. Dies kann mit dem herkömmlichen Internet oder Internet der Dinge (IoT) realisiert werden. Diese Arbeit wird sich dabei zentral mit diesem Bereich beschäftigen.

Informationstransparenz Die vielen verfügbaren Sensordaten aus den Maschinen können genutzt werden, um bestehende Systeme zu erweitern und so effektiv verwenden zu können. Diese Transparenz in den Informationen und Daten dient als kontinuierliche Überwachung

und Verbesserung der Informationsqualität selbst. Dabei spielt die Aktualität der Informationen eine entscheidende Rolle für die Wahl der richtigen Entscheidung innerhalb des Produktionsverlaufs. Dadurch wird die Lage geschaffen ein virtuelles Abbild der gesamten Fabrik oder der Produktionskette zu erzeugen und digital zu steuern.

Technische Assistenz Dem Menschen soll durch geeignete Wahl an Methodiken und Hilfsmitteln die Arbeit weitestgehend erleichtert und die Fehlertoleranz gesenkt werden. Dies ist durch Erweiterung von bestehenden Softwareprodukten oder Neuentwicklungen mit entsprechend Hilfestellungen möglich.

Verteilte Verantwortung Daten können in der gesamten Fabrik verteilt und an verschiedenen Informationssystemen übertragen werden. Dabei sollen die Teilprozesse selbst dafür sorgen, dass sie mit den vorhandenen Informationen ihre vorgesehene Aufgabe dezentral lösen können. Bei Bedarf oder einem Fehlerzustand kann der lokale Zustand unterbrochen und eine andere Instanz hinzugezogen werden.

Für diese Bereichen haben sich neue Informationstechnologien entwickelt wie zum Beispiel die Smart Factories, Production-in-the-Loop oder das Grid Manufacturing [WSCL13].

2.2 Containerprinzip

Das bekannte Prinzip, um Software auf unterschiedlichen Maschinen in einem immer identischen Zustand zu starten, ist mithilfe von virtuellen Maschinen (VMs) möglich. Dabei wird die gewünschte Software auf dem virtuellen Betriebssystem mit allen benötigten Anwendungen und Bibliotheken installiert und anschließend ein komplettes Abbild dieser Konfiguration erzeugt. Die Konfiguration lässt sich auf einem gewünschten System starten, um anschließend mit dem identischen Zustand die Arbeit fortzusetzen. Das komplette Betriebssystem wird auf dem Abbild hinterlegt und benötigt einen sehr hohen Anteil des Speicherplatzes.

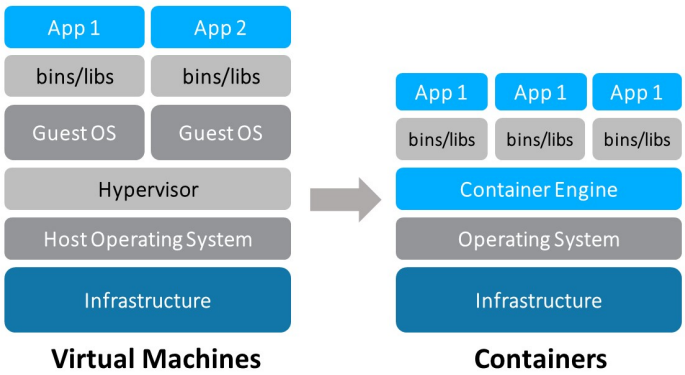

Abbildung 2.2: Darstellung der unterschiedlichen Bestandteile zwischen eines vollwertigen VM-Images und einem Container-Image. [Inc18]

Wie in Abbildung 2.2 zu sehen ist, werden durch das Containerprinzip zwei Schichten, die für die Virtualisierung nötig waren, obsolet. Die Container-Engine übernimmt dabei die Aufgabe

des virtuellen Betriebssystems und des darunterliegenden Hypervisors. Dadurch werden die Container-Images so klein wie nötig gehalten und nur auf das nötigste reduziert. Das entspricht auch der Idee der Containerisierung. Die Container sollen **leichtgewichtig**, **unabhängig** und ein Stück **ausführbare Software** sein. Die laufenden Container sind dabei unabhängig von der Umgebung in der sie ausgeführt werden und verhalten sich dabei immer identisch. Ein weiterer Vorteil ist die benötige Ausführungszeit. Bis eine komplette VM gestartet ist, können je nach Größe Minuten vergehen, wohingegen ein Container-Image nahezu sofort startet.

Die Nutzungsmöglichkeiten der Containertechnik lassen sich nahezu auf alle Bereiche übertragen. Vor allem bietet die deutliche Trennung Vorteile wie eine komfortablere Sicherungen (vor allem im Hinblick auf Datenbanken), aber auch die gezielte Verknüpfung von Containern an. Die Containertechnik legt einen hohen Wert auf eine saubere und definierte Verknüpfungen über Schnittstellen.

2.2.1 Umsetzungen

Für das Konzept gibt es mittlerweile mehrere Umsetzungen. Durch Docker ist die Containerisierung alltagstauglich geworden, da die Nutzung leicht, schnell und die Community sehr groß ist. Im folgenden werden die bekanntesten Umsetzung kurz erläutert.

Docker Wie bereits erwähnt, ist Docker die bekannteste Umsetzung für Containerisierung auf dem heutigen Markt[1]. Die Community ist sehr groß und dadurch sind viele Erweiterungen für Docker entstanden. Bei Docker wird ein Image mithilfe eines sogenannten Dockerfile's definiert. Dort wird festgelegt, was das Basis-Image ist, also beispielsweise ein Ubuntu System, und anschließend wird alles konfiguriert, was für die Anwendung erforderlich ist. Aufgebaut sind die Images aus Layern, wobei ein Layer immer einer Zeile des Dockerfile's entspricht. Dies ermöglicht sehr schnelle wiederholte Bauvorgänge durch Wiederverwendung, falls nur Kleinigkeiten an dem Programm verändert wurden. Die verfügbaren Images können im offiziellen DockerHub Repositry oder auch in privaten Docker Registries abgelegt werden.

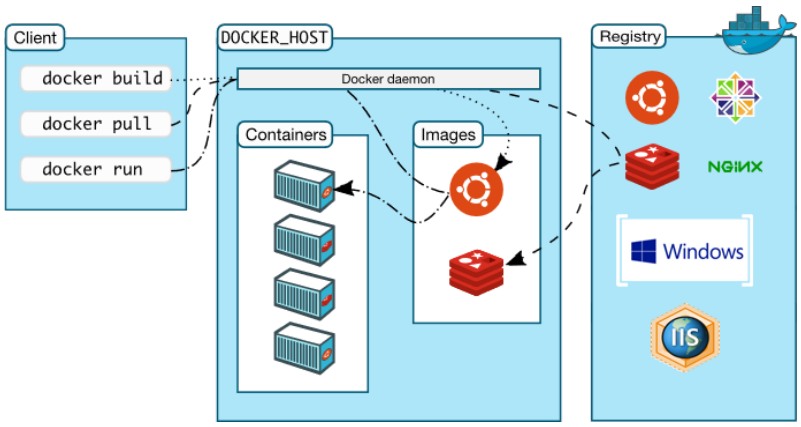

Abbildung 2.3: Überblick über das Docker Ökosystem. [Ter18]

[1] https://www.docker.com/survey-2016

Wie in Abbildung 2.3 zu sehen ist, werden bei der Ausführung eines Docker-Befehls viele Nebenaufgaben im Hintergrund angestoßen, die vom sogenannten *docker daemon* gesteuert werden. Dieser holt alle benötigten Images, falls nicht bereits auf dem System vorhanden, aus der Docker Registry und nutzt diese mehrfach für die benötigten Container. Zu den weiteren Erweiterungen von Docker zählen zum Beispiel Docker-Compose (Orchestrierung von mehreren Services innerhalb einer Datei) und Docker-Machine (Anwendung, um virtuelle Maschinen zu provisionieren).

Rkt (Rocket) Wurde von CoreOS, einem Startup aus dem Sillicon Valley, entwickelt. Rkt setzt für das Containerformat auf den heutigen unabhängigen Standard App-Container-Spezifikation (APPC). Zusätzlich ist Rkt in der Lage Docker Container direkt auszuführen. Der Fokus von Rkt liegt dabei auf der Verbesserung der Sicherheit von Containern. Dies gelingt durch Abtrennung der Container durch einen Hypervisor, SELinux-Support, Unterstützung für Trusted Platform Module (TPM) und weitere sicherheitsrelevante Punkte.

LXD Von den Entwicklern des Betriebssystems Ubuntu wurde LXD hervorgebracht. Diese Containerlösung soll nicht als Erweiterung für Docker gelten, sondern als möglicher Ersatz. Dabei wurde auf Basis der Container-Technologie LXS eine Erweiterung implementiert, die es ermöglichen soll komplette Betriebssysteme und nicht nur Anwendungen über Container bereitzustellen. Verschiedene Containertypen sollen nahtlos miteinander betrieben werden. Beispielweise lässt sich eine LXD Anwendung innerhalb einer LXD-Maschine betreiben.

Weitere Neben den vorgestellten Ansätzen existiert noch eine große Anzahl an kleineren Implementierungen. Dazu zählen Flockport, Windocks oder auch Boxfuse. Diese haben unterschiedliche Schwerpunkte mit denen sie sich von den anderen Anbietern unterscheiden.

Die Containerisierung bietet einen hohen Grad an Flexibilität und Unabhängigkeit, sowie Selbstständigkeit der Container. Für diese Arbeit werden die Container eine wichtige Rolle einnehmen, da diese als Paketformat für Services verwendet werden. Dadurch soll die Nutzung, Verteilung und Ausführung vereinfacht werden.

2.3 Messaging

In diesem Abschnitt wird das Prinzip von Messaging mit den zugehörigen Komponenten in ihren Grundlagen erläutert. Der Kern liegt dabei in der Kommunikation mittels Nachrichten. Die Struktur der übertragenen Nachrichten kann dabei beliebig sein, ist jedoch meistens in Form von Bytes vorzufinden, die komprimiert werden, um Größe und damit Zeit zu sparen. Nachrichten werden von sogenannten „Producern“ gesendet und verarbeitet werden sie von „Consumern“. Das sind die zentralen Kommunikationspartner. Je nach Implementierung können im Nachrichtenfluss noch zusätzliche Komponenten zwischen Consumer und Producer liegen. Als Beispiel kann ein *Broker* genannt werden, der die Kommunikation steuert und überwacht.

Die gesamte Software, die sich um das Messaging und deren Schnittstellen kümmert wird als Message-Oriented-Middelware (MOM) bezeichnet. Eine MOM bietet dabei viele Vorteile [Cur04]:

- Synchrone und asynchrone Kommunikationsmöglichkeiten

- Unabhängige Clients
- Lose Kopplung
- Kommunikation via definierten Schnittstellen

Die lose Kopplung nimmt in dieser Arbeit die wichtigste Rolle ein. Eine lose Kopplung beschreibt eine geringe Abhängigkeit zu anderen Komponenten. Änderungen an einer Komponente sollen dadurch keinen Einfluss auf andere haben. Es ist wichtig, dass die Service keine Informationen voneinander haben. Stattdessen sind nur die übertragenen Daten von Relevanz.

2.3.1 Nachrichtenübertragung

Ein wichtiger Punkt der MOM ist die Art wie Nachrichten übertragen werden. Eine mögliche Form der Kommunikation ist Remote-Procedure-Call (RPC), wo Applikation direkt Funktionen von anderen Anwendungen ausführen und warten. Der Prozess gibt die Kontrolle ab und wartet bis er wieder die Kontrolle erhält. Solche Aufrufe sind durch die Wartezeit kostspielig und wenig performant. Dieses Problem wird mithilfe von Messaging gelöst. Bei der MOM kann dabei zwischen zwei Interaktionsmodellen unterschieden werden: der synchronen und asynchronen Variante [Cur04].

Synchron Bei der synchronen Interaktion ruft ein Prozess (Caller) beispielsweise eine Funktion auf einem anderen Prozess auf (Called). Der Caller pausiert dabei seine weitere Verarbeitung solange bis die Funktion terminiert und eine Rückgabe zurückliefert. Erst zu diesem Zeitpunkt kann der Caller mit seiner ursprünglichen Verarbeitung fortfahren. Ein typisches Beispiel für die synchrone Kommunikation ist der RPC. Für die Prozesse bedeutet dies, dass sie nicht unabhängig voneinander agieren können. Dies ist in Abbildung 2.4 dargestellt. Der Prozess stoppt so lange bis er eine Antwort erhält

Asynchron Bei der asynchronen Variante wird ohne Blockieren des Prozesses gearbeitet. Dabei sendet der Caller eine Nachricht an die MOM, die die Nachricht aufnimmt und dies bestätigt. Der Caller kann dabei direkt mit seiner Verarbeitung fortfahren ohne auf die Antwort auf seine Nachricht zu warten. Dieses Prinzip wird als Send&Forget bezeichnet. Der Called Prozess kann dabei die Nachricht bei der MOM zyklisch abfragen (Pull) oder er bekommt die Nachricht direkt sobald sie da sind (Push). Nach der Verarbeitung sendet er die Antwort wieder an die MOM und diese leitet die Antwort dann nach dem gleichen Prozedere wieder an den ursprünglichen Caller weiter. Im Vergleich zu der synchronen Methode werden die Prozesse nicht blockiert und können dadurch kontinuierlich arbeiten (siehe Abbildung 2.4).

Je nach Anwendungsgebiet muss der Architekt oder Designer überlegen, welche Interaktionsmethode die richtige ist. Dabei sind die vorgestellten Vor-, sowie Nachteile zu beachten. Nachfolgend sind die wichtigsten Eigenschaften der asynchronen Methode erläutert [Cur04]. Wichtig dabei ist, dass keine der Eigenschaften von RPC erfüllt werden. Die MOM hingegen erfüllt diese Eigenschaften.

Kopplung Die Kopplung muss so gering wie möglich gehalten werden. Bei RPC sind die Anwendungen durch die Funktionsaufrufe direkt aneinander gebunden und daher nicht ohne weiteres trennbar. Bei der MOM hingegen wird dafür gesorgt, dass die Nachrichten zwischen Producern und Consumern korrekt geroutet werden.

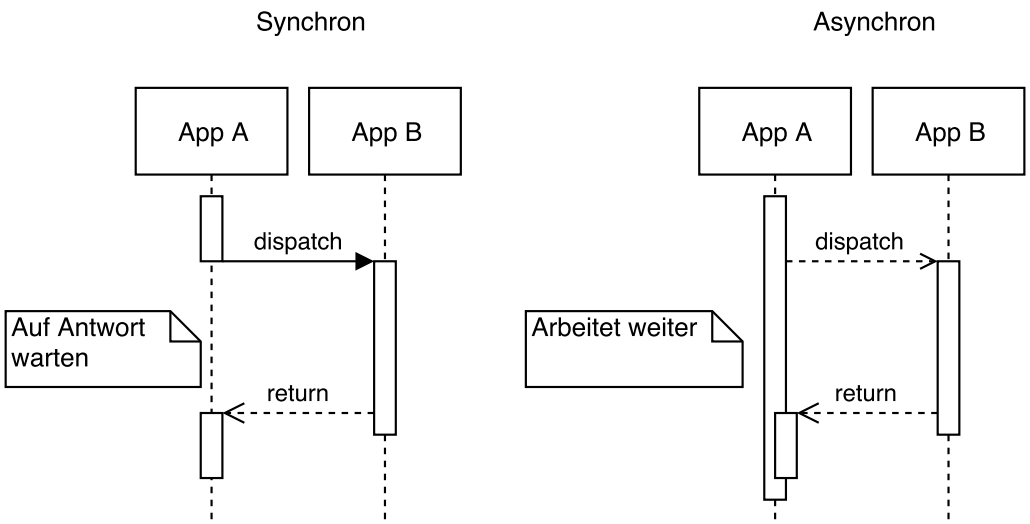

Abbildung 2.4: Vergleich zwischen synchroner Kommunikation und asynchroner Kommunikation.

Zuverlässigkeit Die Kommunikation zwischen den Partnern muss zuverlässig funktionieren und bei einem Ausfall müssen entsprechende Sicherheitsmechanismen vorhanden sein. Bei der MOM kann die Zuverlässigkeit je nach Implementierung definiert werden. Eine mögliche Definition ist das direkte persistieren der Nachrichten mit anschließendem Löschen, sofern die Empfänger die Nachrichten korrekt verarbeitet haben.

Skalierbarkeit Die Systeme und Subsysteme müssen unabhängig voneinander skaliert werden können. Notwendig ist dies vor allem dann, sobald Lastspitzen an bestimmten Zeitpunkten auftreten und reagiert werden muss Die MOM Architektur ermöglicht eine Skalierung in unterschiedlichen Teilbereichen. Genauere Arten der Skalierung sind in [Cur04] erläutert.

Verfügbarkeit Bei Angriffen oder Ausfällen von Teilkomponenten, beispielsweise einem Consumer oder Producer, darf nicht das gesamte System ausfallen. Hier hat die MOM den Vorteil, dass die Nachrichten persistiert sind und die Producer/Consumer ausfallen können ohne dem Gesamtsystem zu schaden. Nach einem Neustart können diese an ihrem alten Zeitpunkt fortsetzen.

2.3.2 Messaging-Methoden

Ein Kernelement von Messaging sind die verfügbaren Methoden, wie Nachrichten übertragen werden. Dabei wird im wesentlich zwischen zwei Methoden unterschieden: Publish-Subscribe und Point-to-Point. Je nach Nachrichten-Anforderungen können diese gemischt werden [Nan15]. Eine Darstellung der Unterschiede von den zwei Kernmethoden ist in Abbildung 2.5 illustriert.

Point-to-Point

Bei dieser Methode werden die Nachrichten mithilfe von einfachen Queues zwischen Producern/-Consumern versendet. Oftmals wird dabei eine First-in-First-out (FIFO) Queue verwendet [Cur04].

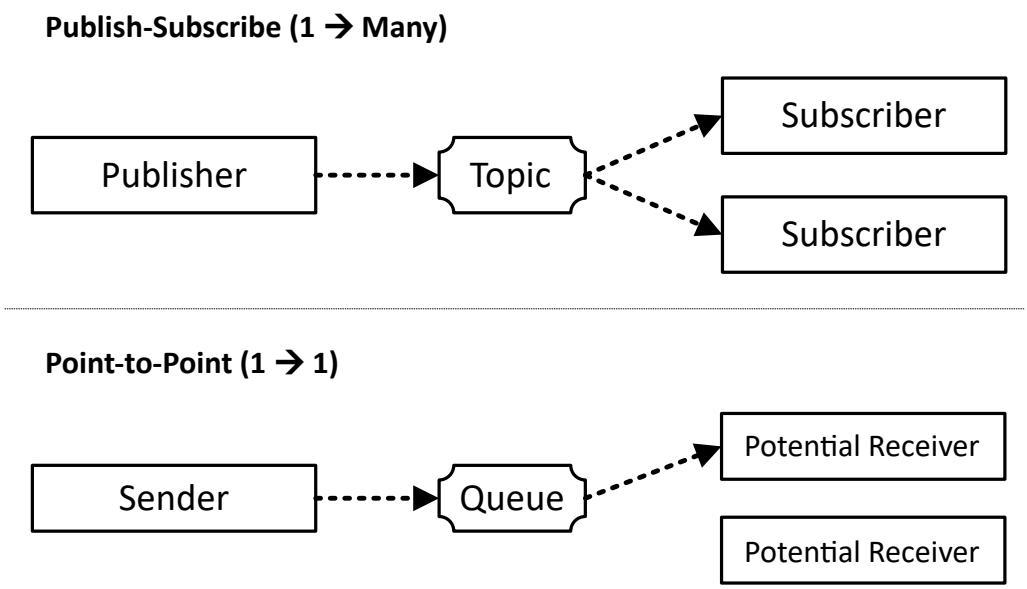

Abbildung 2.5: Nachrichtenübertragung der zwei Messaging-Methoden.

Dadurch wird die Reihenfolge, in der die Nachrichten der Queue hinzugefügt wurden, bewahrt. Die Standardimplementierung unterstützt dabei beliebig viele Producer und einen Consumer. Die Implementierung erlaubt es mehrere Consumer an die Queue zu hängen, jedoch wird eine Nachricht stets von maximal einem Consumer verarbeitet.

Request-Reply Aus dem Konzept des Internets (World-Wide-Web) wurde dieses Modell abgeleitet. Dabei geht es um die Anfrage für eine Website und der zugehörigen Antwort. Dabei muss ein Producer immer auf eine Antwort eines Consumers reagieren können. Dieses Modell kann aus einer Kombination von Point-to-Point und Publish-Subscribe realisiert werden.

Publish-Subscribe

Das Publish-Subscribe Modell trennt die Consumer und Producer weiter auf. Weder Producer noch Consumer wissen wer die Nachrichten verarbeitet. Es wird mit sogenannten Topics gearbeitet, die von der MOM verwaltet werden. Ein Producer published seine Nachrichten an diese Queue ohne dabei zu wissen, ob jetzt einer oder beliebig viele Consumer an dem Topic subscribed haben. Das gleiche gilt auch aus Seiten der Consumer. Diese wissen nicht von anderen Consumern oder wie viele Producer genau mit der Topic arbeiten. Sie erhalten ihre Nachrichten, die sie verarbeiten können. Es gibt auch keinerlei Beschränkungen für den Client. Dieser kann sowohl Producer, als auch Consumer sein.

Nachdem die möglichen Modelle für das Messaging vorgestellt wurden, kann noch die Art der möglichen Zustellung für die Nachrichten definiert werden. Dabei wird zwischen *at-most-once* (Nachrichten höchstens einmal empfangen), *at-least-once* (Nachrichten mindestens einmal empfangen) und *exactly-once* (Nachricht wird genau einmal empfangen) unterschieden. Je nach Wahl muss das System mit der Art der Übermittlung umgehen können.

2.4 Cloud Computing

Mit stetigem Anstieg von Nutzern der Anbieter wie Amazon AWS, Google Cloud und Microsoft Azure ist die Cloud und Cloud Computing heutzutage ein fester Bestandteil der Informatik [LFWW16]. Zahlreiche Unternehmen sehen einen immer höheren Nutzen von externen Rechenlösungen. Dadurch können die lokalen Rechenkapazitäten reduziert werden. Auch im Rahmen der Industrie 4.0 spielt die Cloud eine zentrale Rolle. Die Verteilung der Maschinen und deren Kommunikation soll dynamisch und unabhängig in unterschiedlichen Clouds funktionieren und problemlos miteinander in Verbindung gebracht werden. Die Anwendungen, die in einer Cloud betrieben werden sollen, müssen dabei entsprechende Eigenschaften aufweisen. Diese werden als *IDEAL* bezeichnet [LFWW16; MG12]. Das IDEAL steht für Isolation of state, Distribution, Elasticity, Automated Management und Loose coupling. Die Anwendungen sollen einen isolierten Zustand haben, um mithilfe von unterschiedlichen Ressourcen bearbeitet werden zu können. Des Weiteren sollen die Anwendungen zur Verteilung geeignet sein und eine lose Kopplung aufweisen. Sind die Anforderungen berücksichtigt und umgesetzt, sind die Anwendungen für das Cloud Computing vorbereitet [MG12].

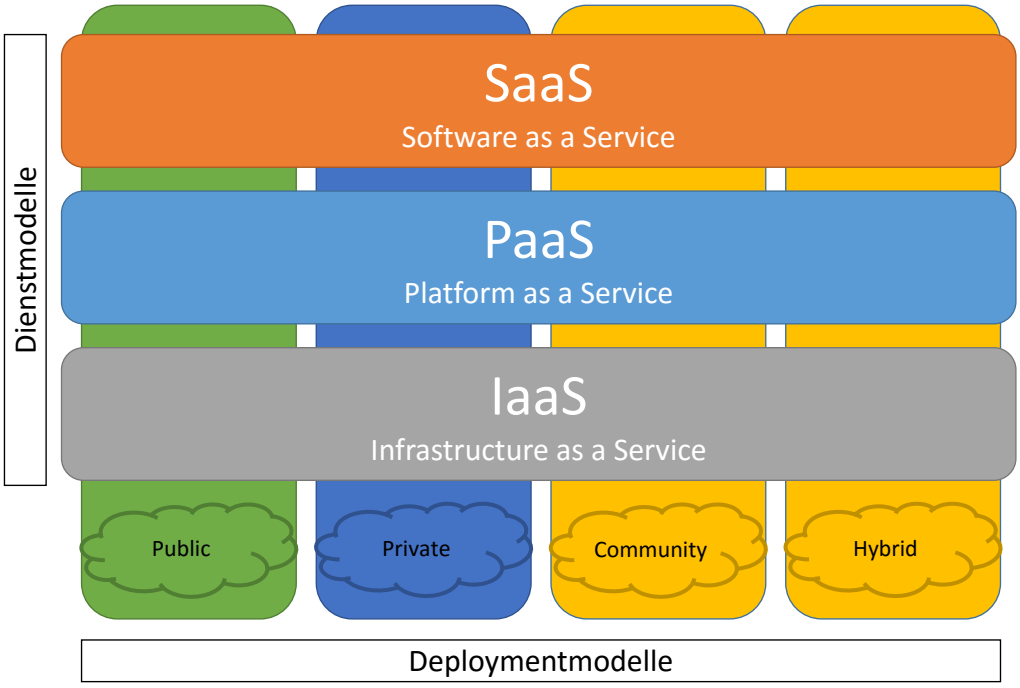

Abbildung 2.6: Ein zusammenfassender Überblick die zwei wichtigsten Bestandteile des Cloud Computings: den Dienstmodellen und Deploymentmodellen.

Zum Cloud Computing gehören zwei wesentliche Modelle, die einerseits den Dienst in einer Form bereitstellen und andererseits die Form der Cloud definieren wie in Abbildung 2.6 zu sehen. Diese beiden Modelle werden im folgenden genauer erläutert.

2.4.1 Dienstmodelle

Bei den Diensten, die eine Cloud anbieten, lässt sich im wesentlich zwischen drei Formen unterscheiden [MG12]. Die unterste Schicht (IaaS) hat ein noch sehr geringes Abstraktionsniveau. Die Abstraktion steigt mit jeder Schicht immer weiter an bis der Punkt erreicht ist, an dem nur noch Anwendungen bereitgestellt werden. Die drei Kernschichten sind in Abbildung 2.6 dargestellt.

Infrastructure as a Service (IaaS) ist die unterste Schicht. Der Cloudanbieter bietet virtuelle Maschinen mit gewünschten Konfigurationen (Anzahl der CPUs, RAM Größe etc.) an. Im Ausnahmefall könnten dies auch echte Maschinen sein. Der Nutzer hat volle Kontrolle über die Maschine. Das Bezahlmodell entspricht "Pay-per-use", es wird nur die tatsächliche Nutzung bezahlt.

Plattform as a Service (PaaS) ist eine Ebene über IaaS, bei der es nicht mehr möglich ist komplette Maschinen (physisch oder virtuell) zu erhalten, stattdessen wird die Möglichkeit geboten Archive bereitzustellen, die vom Provider gehostet werden. Die bekannteste Variante ist das Deployen von Java WAR Archiven im Tomcat. Dabei liefert der Nutzer bereits ein lauffähiges Programm/Anwendung, die dann nur noch gehostet wird.

Software as a Service (SaaS) ist die höchste Abstraktionsstufe. Hierbei muss sich der Nutzer um keine Aspekte wie Maschine oder Hoster mehr kümmern. Der Nutzer verwendet nur noch eine Software, die ihm zur Verfügung gestellt wird. Ein bekanntes Beispiel sind hierbei die Google Anwendungen wie zum Beispiel Gmail.

Neben den vorgestellten drei Kernmodellen sind noch verschiedene Abwandlungen mit neuen Begriffen entstanden. Als Beispiel gibt es eine leichte Abwandlung von PaaS zu Container as a Service (CaaS). Der Fokus richtet sich auf die Bereitstellung von Containern. Dies ist vor allem bei Containerisierungen sehr interessant[2]. Ein wichtiger Faktor, der mit den Dienstmodellen einhergeht, ist das Bezahlprinzip. Hier kann der Anbieter je nach Aufstellung seiner Dienste verschiedene Möglichkeiten anbieten, wie er seinen Service abrechnen kann. Wie bereits erwähnt kann er beispielsweise dem Nutzer die reine Nutzungszeit anrechnen oder auch die Auslastungsmenge, Ressourcenverwendung oder die Nutzung in Blöcken anbieten.

2.4.2 Deploymentmodelle

Neben den verschiedenen Diensten einer Cloud gibt es noch Modelle für die Art des Deployments, also wie die Clouds bezüglich Standort aufgestellt sind. Wie in Abbildung 2.6 zu sehen ist, gibt es vier Modelle.

Public Cloud Das sind jene Clouds, die für alle Nutzer frei zugänglich sind und keine direkten Beschränkungen haben. Die Form ist am weitesten verbreitet und zu dieser zählen die bekannten Anbieter Amazon AWS, Google Cloud und Microsoft Azure.

[2] http://www.searchdatacenter.de/tipp/Container-as-a-Service-Kurzueberblick-ueber-die-drei-wichtigsten-CaaS-Anbieter

Private Cloud Dies sind Clouds, die in einem abgeschlossenen Bereich gehostet werden und nicht frei zugänglich sind. Typischerweise werden solche Clouds in Firmenrechenzentren gehostet und sind auch nur im firmeninternen Netz verfügbar. Durch den beschränkten Zugriffsbereich hat diese Cloudform eine höhere Sicherheit, da alle Daten nur im eigenen Netz vorhanden sind.

Community Cloud Bei der Community Cloud werden die Daten zwischen Organisationen und/oder Firmen getauscht. Das bedeutet, dass die Cloud Daten nicht nur in einem Netz zur Verfügung stehen, sondern in einem Zusammenschluss aus mehreren Netzen. Nur ausgewählte Partner und Gruppen erhalten somit Zugriff auf die Cloud.

Hybrid Cloud Wie der Name schon andeutet besteht die Cloud aus einer Kombination von verschiedenen Cloudarten. Dabei können Public und Private Clouds kombiniert werden und je nach Bedarf oder Anforderungen zwischen diesen verteilt werden. Manche Dienste laufen auf der privaten Cloud, wohingegen andere auf einer Public Cloud laufen können.

2.5 Schemas

Ein wichtigen Baustein bei den Service nimmt die Strukturierung, Definition und die Festlegung des Kommunikationsformates ein. Dafür ist die Definition eines geeigneten Schemas entscheidend. In dem Schema wird festgelegt, wie die Struktur aussieht und welche Typen verwendet werden können. Im nachfolgenden werden ein bekannte Varianten vorgestellt, die eine Definition eines Schemas ermöglichen.

2.5.1 Varianten zur Schemadefinition

Schemas können mit verschiedenen Mitteln und Ansätzen definiert werden. Dafür können eigene Sprachen, Tags oder Annotationen genutzt werden. Im folgenden werden einige Methoden aufgeführt.

Text Die einfachste Möglichkeit ein Schema zu definieren oder festzulegen, ist die Nutzung von einfachen textuellen Elementen. Die Elemente können selber konzipiert oder nach Regeln und Definitionen aufgebaut werden.

XML Eine der bekanntesten Methoden zur Definition eines Schemas ist die Nutzung von XML. Dabei kommt das XML Schema zum Einsatz kurz XSD (XML Schema Definition). Diese dient zur Strukturierung der XML Dokumente und bietet viele Datentypen an. Darüber hinaus lassen sich eigene Datentypen definieren, vorhandene erweitern, beschränken oder an Bedingungen knüpfen. Ein Nachteil von der Verwendung von XML ist die Entstehung von sehr großen Dateien, da zu einem öffnenden Element stets ein schließendes hinzukommt. Es gibt Möglichkeiten wie die Nutzung von XML-basierter Kommunikation komprimiert werden kann.

WSDL Die Servicebeschreibung mithilfe von WSDL gehört eigentlich zur XML-Variante, da WSDL mit XML geschrieben wird. Jedoch hat sich das WSDL als Standard für firmenrelevante Anwendungen etabliert. Die Definition ist durch klare Linien und Strukturen

gegliedert und aufgebaut. Auch sind die Erweiterungsmöglichkeiten durch den gesamten W*-Standard sehr groß [WHS11].

JSON Schemas über das JSON-Format zu definieren nimmt an Beliebtheit zu. Durch die intensive Nutzung von Webanwendungen und mobilen Applikationen ist die Bekanntheit von JSON gestiegen. Aus dem Browser sind Anfragen mit JSON nicht mehr wegzudenken. Für JSON gibt es momentan noch keinen festgelegten Standard. Eine bekannte Definitionsstruktur für JSON kommt von Apache Avro. Hier werden ähnlich zu den XSDs atomare Typen vorgegeben, die mithilfe von Strukturen, Listen und Mengen beliebig erweitern und konzipiert werden können.

Ein weiterer wichtiger Punkt im Zusammenhang mit den Schemas ist die Evolution, also die Veränderungen eines Schemas über einen Zeitraum. Für jede Änderungen wird dabei eine neue Version für das Schema festgelegt. Aus den Versionen ergibt sich ein Verlauf an dem eine Transformation für alte Versionen ansetzen kann. Das ganze wird als Schema-Evolution bezeichnet.

2.6 OPC Unified Architecture (OPC UA)

Nachdem die Containerisierungen und Cloud Computing Punkte erläutert wurden, ist es wichtig echte Maschinen in das gesamte Netzwerk der Services zu integrieren. Dafür gibt es bereits Schnittstellen wie Daten von Maschinen abgegriffen werden. Es gibt eine OPC-Spezifikation für Maschinen, die ein industrielles M2M-Kommunikationsprotokoll (Machine-to-Machine) definiert. Dies ist die OPC UA. Mit der neusten Spezifikation ist neben der Datenübertragung nach außen auch eine Übertragung zur Maschine möglich. Die OPC UA orientiert sich dabei an der Server-Client-Architektur, dies bedeutet, dass ein Server die Daten/Anbindung der Maschine bereitstellt und der Client mit diesem interagieren kann. Die neuen Erweiterungen von OPC UA umfassen dabei die Redundanz, Verbindungsüberwachung und eine Pufferung der Daten. Dadurch wird eine verlustfreie und konsistente Verbindung ermöglicht.

Protokolle Bei dem OPC UA Protokoll kann zwischen zwei Varianten gewählt werden. Einerseits steht das TCP Protokoll zur Verfügung, um Daten binär zu transferieren. Das binäre Protokoll hat den Vorteil, dass es zwischen verschiedenen Teilnehmern genutzt werden kann. Weiterhin ist die Performance bei der binären Übertragung sehr hoch. Die zweite Möglichkeit ist die Übertragung mittels HTTP als Webservice. Das Format hierfür ist ist XML-basiert via SOAP.

2.6.1 Server/Client

Der Aufbau für den Server und Client ist in Abbildung 2.7 zu sehen. Das Grundgerüst bilden die Base Services, die abstrakte Beschreibung der verfügbaren OPC Dienste darstellen. Diese werden mithilfe des Transportlayers mit den zuvor vorgestellten Protokoll kommuniziert. Aufbauend auf den Grunddiensten können spezielle Modelle aufgesetzt werden, die beispielsweise das Informationsmodell erweitern oder spezielle Dienste je nach Gerät oder Hersteller benötigt werden. Dies macht die gesamte Architektur sehr flexibel.

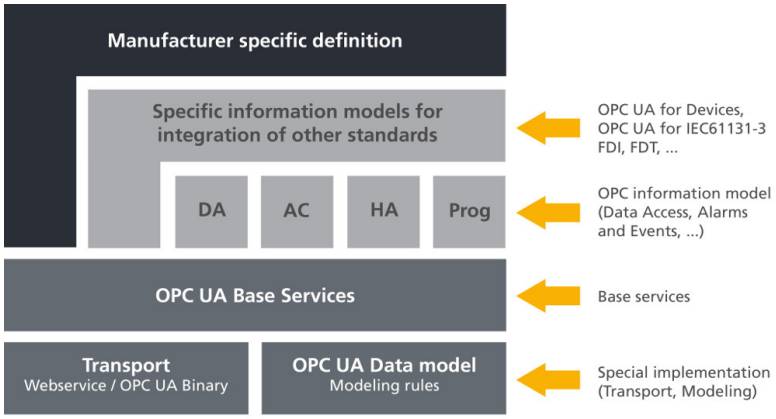

Abbildung 2.7: Darstellung der gesamten Struktur von OPC UA. Die unterschiedlichen Layer kümmern sich dabei um ihre eigenen Teilbereiche. [Tec18a]

Das Informationsmodell besteht aus einem Netzwerk aus Knoten (nodes). Diese Knoten können verschiedene Daten übertragen: normale Nutzungs-, Meta- und Diagnosedaten. Zum Abrufen der Daten muss der gewünschte Knoten mit einer entsprechenden Schnittstelle angegeben werden. Mithilfe eines Intervalls kann die Übertragungsrate festgelegt, in dem die Daten geschickt werden. Dadurch können die Maschinendaten flexibel abgefragt werden, die über einen OPC UA-Server verfügbar sind. Zusätzlich bietet die Spezifikation die Möglichkeit Daten der Maschine nicht nur zu lesen, sondern auch Daten in die Maschine zu schreiben. Die Funktion des schreibenden Zugriffs wird im Rahmen dieser Arbeit jedoch nicht benötigt. Die Kommunikation wird dabei nach dem Publish-Subscribe Verfahren durchgeführt.

2.7 Deployment

Mit den nun definiertem Aufbau der Services und der Kommunikationsstrukturen müssen diese aufgesetzt und gestartet werden. Die Container müssen auf einer Cloud deployed und entsprechend konfiguriert werden, damit diese miteinander kommunizieren können. Im nachfolgenden werden einige Deploymentarten aufgelistet und beschrieben.

Container Die schnellste und einfachste Form eines Deployments. Mithilfe von der Containerisierung lassen sich Anwendungen schnell und stets im identischen Zustand starten und direkt verwenden. Es ist nur wenig zusätzliche Infrastruktur vorhanden und auf das notwendigste reduziert, was die Anwendung benötigt.

VMs Das Deployment über virtuelle Maschinen ist die allgemeinere Form verglichen zur Containervariante, da hierbei ein komplettes Betriebssystem gestartet werden muss. Pro VM ist dabei ein vollständiges System zu starten, was zu einer hohen Redundanz an Daten und zu hohem Leistungsverbrauch führt Vorteil dieser Methode ist die sichere Umgebung, da das System direkt vorkonfiguriert und an die Ansprüche angepasst werden kann und davon dann ein Abbild erzeugt wird.

Skript Diese Form des Deployments erfordert im Vorfeld einige manuelle Arbeit, da per Skript vieles festgelegt werden kann. Beispielsweise können benötigte Dateien oder Bibliothek heruntergeladen und verarbeitet werden. Anschließend werden alle Teilkomponenten verknüpft und die Subsysteme per Befehl gestartet.

Je nach Ausgangssituation und Anwendungsfall muss zwischen einer Deploymentart gewählt werden mit der optimale Resultate erzielt werden können. Die genaue Art des Deployments wird in dieser Arbeit jedoch nicht weiter untersucht werden. Es soll zum Verständnis der gesamten Architektur beitragen.

2.7.1 Automatisierung

Eine wichtige Komponente im Bezug zum Deployment ist der mögliche Automatisierungsgrad des gesamten Prozesses. Dazu zählen Begriffe wie Continous Testing, Continous Integration und Continous Deployment. Dabei nimmt der Teil der Automatisierung der zuvor genannten Begriffe mehr und mehr zu [HRN06]. Idealerweise wird durch einen Commit in ein Versionsverwaltungssystem wie Git oder SVN die Pipeline angestoßen und automatisch Tests durchgeführt, anschließend das Paket zusammengebaut und auf einer Zielinstanz aufgesetzt. Bekannter Vorreiter hierfür ist zum Beispiel Amazon, wo mehrere tausend Deployments automatisiert an einem Tag durchgeführt werden. Dies hat den Vorteil, dass die Releases und dessen Zyklen für ein Unternehmen zügiger und leichter ablaufen.

3 Verwandte Arbeiten

Dieser Teil der Arbeit beschäftigt sich mit den verwandten Arbeiten und Themen in Hinblick auf die Automatisierung von Services und dessen Verknüpfungen mithilfe von Cloudkomponenten. Das Kapitel wird dabei in einen Stand der Wissenschaft und einen Stand der Technik gegliedert, um die verschiedenen Bereiche zu separieren. Der wissenschaftliche Anteil muss dabei nicht zwingend prototypische Umsetzungen enthalten, sondern kann Konzepte oder Ansätze darstellen. Beim technischen Teil werden gezielt verfügbare Applikationen und Plattformen untersucht und mit den Anforderungen der Arbeit verglichen.

3.1 Stand der Wissenschaft

In diesem Abschnitt werden die verfügbare wissenschaftlichen Ansätze mit den Anforderungen der Arbeit verglichen.

3.1.1 Spezifikation eines Services

Ein wichtiger Teil für die Automatisierung eines Herstellungsprozesses nimmt die Definition und Spezifikation eines Services ein. In diesem Abschnitt werden verschiedene Möglichkeit zur Beschreibung von Services dargestellt.

WSDL

Ein bekanntes Beispiel ist der öffentliche Standard für Webservices: Web Service Description Language (WSDL) [CCMW01; CMRW07]. Es ist eine Sprache zur Beschreibung von Nachrichten und Services in plattform-, programmiersprachen- und protokollunabhänginger Form. Dabei gibt die Beschreibung Antworten auf die Fragen *Wie* und *Wo* mit dem Service kommuniziert werden kann. Durch den gesamten Standard für Webservices, der durch die W3C veröffentlicht wurde, bietet sich ein sehr breites Spektrum an Möglichkeiten, Definition und Spezifikation für die Interaktion und Kommunikation von Services an. Zudem werden ebenso Punkte wie Sicherheit, Bedingungen und Aushandlungen an Informationen durch den WS* Standard ermöglicht [CCMW01]. WSDL wird mithilfe von XML dargestellt und aufgebaut. Zusammen mit dem Simple Object Access Protocol (SOAP) wird WSDL oft für das Internet verwendet. Dadurch wird die Form der Nachrichten strikt vorgegeben.

Ein Service wird in WSDL dabei mit folgenden Grundelementen definiert, die einzelnen XML Tags entsprechen: types, message, interface, binding, endpoint und service. Dabei wird die Nachrichtenart, Aufbau der Nachricht, eine Menge an möglichen Operationen, etc. genau definiert. Die Funktionalitäten eines Services werden dabei abstrakt beschrieben, können bei Bedarf jedoch auch

konkret mithilfe von Details zu der Servicebeschreibung definiert werden. Ein wichtiger Kernaspekt ist auch die Wiederverwendbarkeit von Teilbeschreibungen eines Services in verschiedenen Bereichen, die über Verlinkungen mit URLs referenziert werden [CMRW07].

TOSCA

TOSCA steht für *Topology and Orchestration Specification for Cloud Applications* von OASIS. TOSCA ist eine Sprache zur Beschreibung von Services und Komponenten und deren Verknüpfung zum Betrieb in der Cloud. TOSCA selbst ist keine Implementierung für die Sprache, sondern nur die Spezifikation und Definition [OAS13; Zim16]. Die Kernaspekte von TOSCA sind dabei:

- Beschreibung von Services und Beziehungen innerhalb einer Topologie
- Beschreibungen von Orchestrierungen und dem entsprechenden Management
- Erweiterbarkeit durch anbieterspezifische Teile
- Templates für Wiederverwendbarkeit in unterschiedlichen Umgebungen
- XML als Beschreibungssprache

Der Aufbau eines Service Templates in TOSCA ist in Abbildung 3.1 dargestellt. Dabei werden alle benötigten Informationen eines Services für die Cloud definiert. Dies soll zu einer reibungslosen und einfachen Migrationen zwischen unterschiedlichen Cloud-Anbietern führen. Das Service Template selbst besteht aus Teilkomponenten wie den *Node types*, *Relation types*, *Plans* und dem *Topology template* selbst [Zim16].

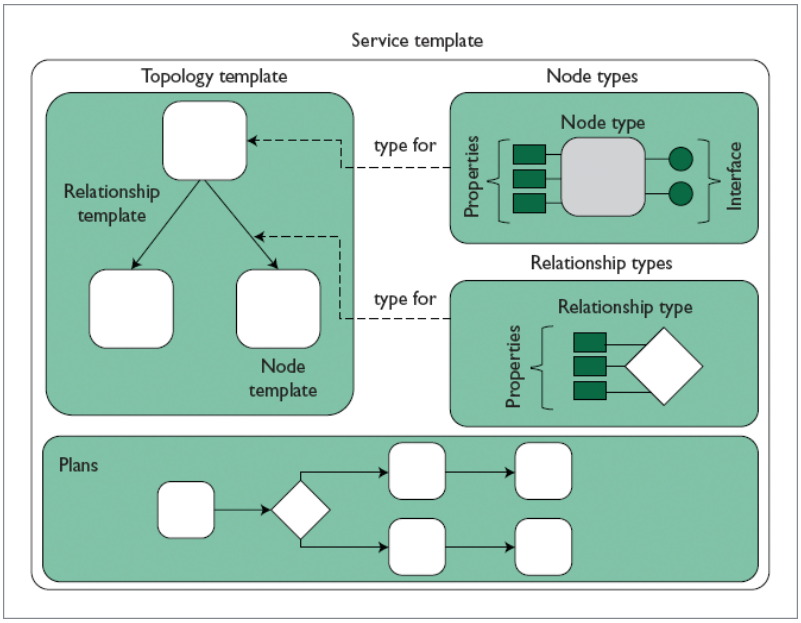

Abbildung 3.1: Aufbau eines Service Templates mit den zentralen Teilkomponenten. [BBLS12]

Node type Abstrakte, wiederverwendbare Definition von Komponenten wie zum Beispiel einem Server oder einer Softwarekomponente. Innerhalb dieser können weitere Eigenschaften, beispielsweise RAM Größe, und auch mögliche Operationen (Interfaces) definiert, die ausgeführt werden können.

Relationship type Dienen der Beschreibung von Abhängigkeiten, Interaktionen und Verbindungen der nodes. Diese sind ebenso wiederverwendbar und lassen sich zusätzlich mit Eigenschaften beschreiben.

Topology template Besteht selbst aus den Teilen: *Relationship template* und *Node template*. Diese entsprechen dabei einem konkreten Element von den abstrakten types. Die Topologie des Services wird dabei als gerichteter Graph beschrieben.

Plan Um den nun angefertigten Service verwalten und managen zu können, werden Abläufe benötigt. Diese werden oft mithilfe der Beschreibungssprachen BPMN oder BPEL definiert, können jedoch auch beliebige andere Sprachen sein [Zim16].

Neben den vorgestellten Aufbau eines Service templates gibt es noch viele weitere Bausteine in der TOSCA Spezifikation. Dazu zählen zum Beispiel *Artefakte*, die eine benötigte Ressource für ein Deployment darstellen bspw. eine Tomcat Instanz. Ein anderer Baustein ist die Darstellung von Einschränkungen und Bedingungen eines Services. Node types können über die Eigenschaften *Requirements, Capabilities* entsprechend erweitert werden, um diese auf den genauen Anwendungsfall begrenzen zu können [OAS13]. Durch das große Spektrum an Möglichkeiten, macht es die gesamte Spezifikation einerseits vielfältig, auf der anderen Seite wird dadurch jedoch die Einfachheit und Adaptierbarkeit reduziert.

OpenTOSCA Das Institut für Architektur von Anwendungssystemen (IAAS) hat eine Open Source Implementierung für die TOSCA Spezifikation bereitgestellt. OpenTOSCA stellt ein gesamtes Ökosystem dar bestehend aus: einem TOSCA Container (TOSCA Laufzeitumgebung), einem grafischen Modellierwerkzeug (Winery) und einem Portal für die Anwendungen (Vinothek) [BBH+13]. Auf die Teilsysteme Vinothek und Winery wird nicht genauer eingegangen (siehe [BBH+13] für detaillierte Informationen).

Die Architektur (siehe Abbildung 3.2) besteht im wesentliche aus drei Teilen: der Implementierungsengine (IAE), der Planengine (PE) und einem Controller, der das ganze steuert. Zusätzlich zu den drei Kernkomponenten können noch weitere Daten wie Informationen zum Model, genaue Instanzdaten oder den Endpunkten gespeichert werden. Wenn die Komponenten weiter definiert werden, kommen unterschiedliche Bus-Komponenten hinzu, die miteinander kommunizieren und agieren. Mithilfe einer definierten API können Tools mit dem Container arbeiten oder Analyseinformationen bereitstellen. Das ganze wird innerhalb einem Container, in dem sogenannten *CSAR*-Format, bereitgestellt. Hierbei handelt es sich nicht um Docker Container. Dieser Container kann dabei genutzt werden, um einen Service auf einer Cloudplattform zu starten. Alle benötigten Operationen und Instanziierungen von Cloud-Instanzen werden dabei so wie in dem Plan angegeben vollautomatisiert durchgeführt. Am Ende kann beispielsweise auf die Instanz über eine öffentliche IP-Adresse zugegriffen werden. Dies ist ähnlich zu einem Docker Container

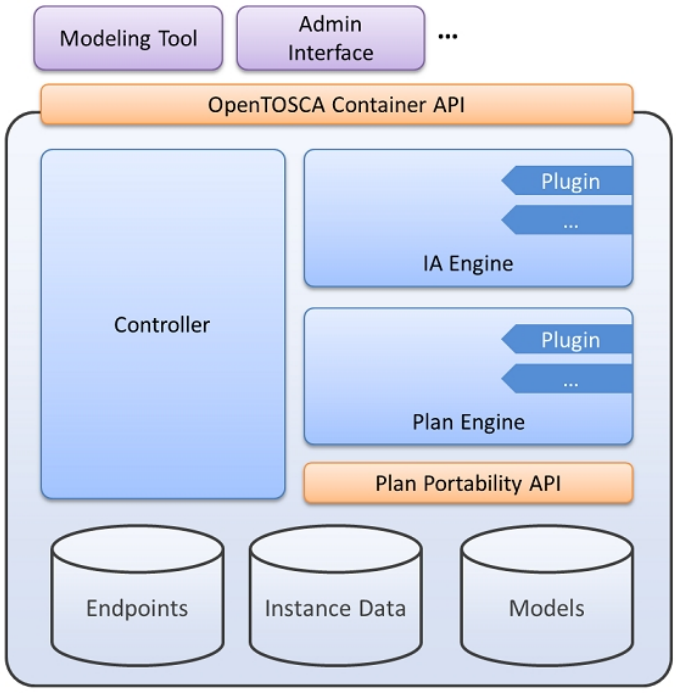

Abbildung 3.2: Architekturdarstellung von OpenTOSCA. [Arc18]

(siehe Kapitel 2). OpenTOSCA wird aktiv auf GitHub[1] weiterentwickelt und gewartet, befindet sich aktuell jedoch noch im Forschungsstatus.

Neben den beiden Standards WSDL und TOSCA sind keine weiteren verbreiteten und standardisierten Definition vorhanden. Daher ist die Wahl zwischen beiden begrenzt.

3.1.2 Cloud Manufacturing

Beim Cloud Manufacturing geht es um die Übertragung der bekannten Cloud Computing Prinzipien (siehe Abschnitt 2.4) auf die Produktion. Das Ziel ist es die Unternehmen durch die Digitalisierung virtuell zu vernetzen und damit eine *virtuelle Produktion* zu erschaffen [BTV14]. Dadurch sollen Faktoren wie Fertigungsstandort, Flexibilität der Netzwerke und auch auftragsspezifische Anforderungen schneller und leichter umgesetzt werden. Zusätzlich werden die verfügbaren Daten effektiver genutzt und die Effizienz von Maschinen und Produktionsprozessen gesteigert.

Dabei sind einige neue Terme und Begriffe im Produktionsumfeld entstanden. Die *Smart Factory* beschreibt die optimale Verbindung von verschiedenen Bereichen einer Fabrik, um Daten und Information zur richtigen Zeit am richtigen Ort zu verbinden. Neben Arbeitsplätzen, Maschinen, Werkzeugen und anderen Hilfsmitteln sollen allen Komponenten einer Fabrik optimale Bedingungen für den Betrieb geschaffen werden. Die *Mixed Reality* steigert ihre Popularität immer weiter. Mixed Reality steht für die Verknüpfung der Realität mit informationstechnischen Daten und Veranschaulichungen. Als Beispiel kann die Automobilherstellung genannt werden, bei der

[1] https://github.com/OpenTOSCA/container

mithilfe von VR-Brillen Informationen in Echtzeit zu Bauteilen des Fahrzeugs angezeigt werden. Das können Maße der Bauteile, Motorisierung oder andere relevante Informationen. Dies ist bereits aus der Spieleindustrie oder den Smartphones bekannt, weit verbreitet und findet langsam den Weg in den Industriebereich.

Der wichtigste Bestandteil für diese Arbeit ist zudem die Verwertung von kontinuierlichen Maschinendaten, die im Laufe einer Bearbeitung erzeugt werden. In vielen Bereichen werden diese Informationen zwar erzeugt, jedoch nicht weiter verarbeitet oder analysiert [BTV14]. Dadurch werden potentielle Optimierungen eines Prozesses übersehen oder mögliche Defekte durch z. B. Früherkennung nicht berücksichtigt. Das bedeutet für Firmen Gewinnverlust und/oder Kostenerhöhung.

3.1.3 Sicherheit und Schutzmechanismen

Bei der Sicherung von Daten, der Kommunikation und allgemeinen Anwendung kann das bekannte *OSI-Schichtenmodell* (Open Systems Interconnection) als Orientierung genutzt werden [HS96]. Das OSI-Modell besteht aus 7 Schichten, wobei die erste Schicht (physische Schicht) sich mit echter Hardware, wie zum Beispiel einem LAN-Kabel, beschäftigt. Mit jeder zunehmenden Schicht nimmt die Abstraktion zu. Von grundlegenden Übertragung von Bits zu konkrenten Daten in Schicht 7, der Applikationsschicht. Wenn nun die Schichten des OSI-Modells als Referenz genommen, lassen sich pro Schicht einzelne Sicherheits- und Schutzmechanismen integrieren. Im simpelsten Fall werden die Daten innerhalb der verschiedene Protokolle getarnt, um möglichen Angreifern die Daten nicht sichtbar darzubieten [HS96]. Die ersten vier Schichten sind dabei für das Netzwerk relevant und die letzten 3 Schichten werden vom Betriebssystem verwaltet. Um für Sicherheit zu sorgen, gibt es unterschiedliche Verfahren und Techniken. Daten und Kommunikationskanäle können verschlüsselt werden, es kann mit Identitätsschlüsseln gearbeitet werden, um eine Validierung der Informationen zu ermöglichen oder die Daten möglichst intelligent verstecken. Dies soll dazu beitragen, dass der Angreifer das Gefühl vermittelt bekommt keine relevanten Daten in der Kommunikation zu finden. Je mehr Sicherungsschichten eine Anwendung hat, desto schwieriger ist es für einen Angreifer verwertbare Daten abgreifen zu können. Sobald es ihm gelingt eine Schicht zu umgehen, bleiben trotzdem weitere Schichten zwischen ihm und den Rohdaten. Zudem bleibt ein Nutzer oder Administrator stets eine wichtige Sicherheitskomponente in solch einem System. Dieser spielt oftmals die erste Schnittstelle zwischen den Rohdaten und den Sicherungsebenen.

In der Softwareentwicklung spielt die Sicherung der tiefsten Schichten dabei eine eher untergeordnete Rolle, da kein Kontakt zu Hardware besteht. Hier liegt der Fokus eher auf der Sicherung von der Anwendung selbst, der Kommunikation oder der Netzverbindung. Wie in Abbildung 3.3 zu sehen kann in der äußersten Schicht mit der Ausbildung der Personen über Schulungen oder Richtlinien begonnen werden. Dies verschafft bereits erste Sicherungspunkte, um dem Angreifen den Zugang zum System zu erschweren. Dies ist bspw. durch infizierte E-Mail Anhänge bekannt, die in einer Firma geöffnet werden und anschließend das gesamte Firmennetz infiltrieren. Tiefere Schichten können bei einem Versagen der äußeren Schicht weiter eingreifen. Beispielsweise durch eine Filter, Firewalls oder Antivirus-Software. Weitere Maßnahmen zur Sicherung sind Verschlüsselung von Daten durch Passwörter oder Schlüsselpaaren. Dies soll im Falle der Datendiebstahls ein Verarbeiten der Daten unterbinden. Eine weitere wichtige Sicherungsmethode sind Backups der Daten. Durch die Ausbreitung von Erpresser-Software, die alle Daten verschlüsseln

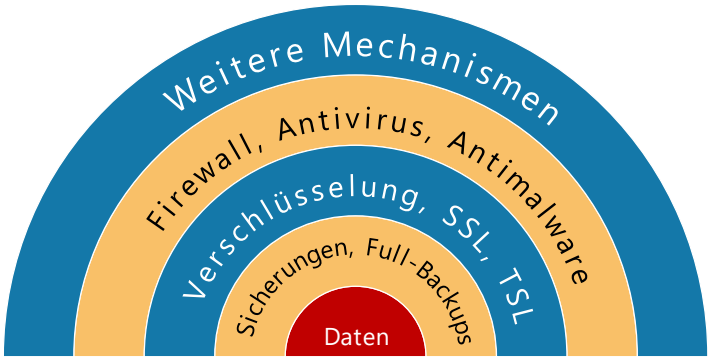

Abbildung 3.3: Darstellung von verschiedenen Schutzmechanismen der Daten. Diese sind am Zwiebelprinzip angelehnt, um den Schutz zu erhöhen.

(sogenannten Ransomware), ist eine sichere Methode zur verlustfreien Wiederherstellung seiner Daten entscheidend. Im konzeptionellen Teil der Arbeit (Kapitel 4) werden unterschiedliche Möglichkeiten untersucht, wie welche Schicht geschützt werden können. Dabei wird auch Bezug auf Tools und Technologien genommen, um den aktuellen Standards und Verfahren gerecht zu werden.

3.2 Stand der Technik

Durch Industrie 4.0 hat sich bei Firmen und Fabriken das Thema rund um Digitalisierung und Cloud Computing etabliert. Daraus resultieren Projekte, die die neuen Möglichkeit der Themen testen. Diese Projekte und verwandten Arbeiten sind vielfältig in können unterschiedlichen Bereichen betrachtet werden, angefangen von einfachen Teillösungen für bspw. das Deployment bis zu kompletten Lösungen für die Vernetzungen mehrerer Maschinen.

3.2.1 Containermanagementplattformen

Für das Management von Containern gibt es bereits eine Vielzahl an Tools, die große Teile der Steuerung von Containern übernehmen. Diese Werkzeuge werden oft als Orchestrierungswerkzeuge bezeichnet. Ziel der Tools ist es dabei das Starten, Stoppen und Managen von Containern/-Anwendungen zu übernehmen und bestmöglich zu automatisieren. Bei einigen Werkzeugen ist auch ein automatisches Provisionieren bei Cloudanbietern integriert. Alle vorgestellten Tools haben dabei verschiedene Sicherheitskonzepte integriert, um interne oder externe Kommunikation, sicheren Betrieb oder Datenverschlüsselung zu gewährleisten.

Docker Swarm Von Docker selbst ist das sogenannte *Docker Swarm* entwickelt worden mit dem Ziel verschiedene Maschinen innerhalb eines Netzwerks, einem Schwarm, leicht verwalten und nutzen zu können. Docker Swarm ist dabei eng in die bestehende Landschaft von Docker

eingebunden und macht sich Docker Engine zu nutze. Im wesentlichen besteht ein Swarm aus *Managern* und *Workern*. Die Manager leiten und stellen die Arbeitsfähigkeit des Swarms sicher, wohingegen die Worker für die konkrete Arbeit, also dem Ausführen von Container, zuständig sind. Als Kernfeatures sind dabei Punkte wie ein dezentrales Design, Konfiguration zur Laufzeit, Skalierbarkeit und Multihost Networking fundamental gewesen [Doc17]. Interessant ist zudem wie Services innerhalb eines Swarms angesprochen werden können. Durch eine integrierte Service Discovery kann dabei ein Service an jedem beliebigen Knoten des Swarms angefragt werden und die Anfrage wird zu dem Knoten durchgeleitet auf dem der Service läuft [Doc17]. Das erleichtert die Nutzung von Services, da nicht an einen genaue Knoten adressiert werden muss. Durch die Nutzung von einzelnen Zertifikaten für alle Hosts, die im Swarm beigetreten sind, ist eine entsprechende Sicherheit der Kommunikation und des Swarms von Beginn an gewährleistet.

Kubernetes Eine ähnliche Lösung wurde dabei von Google mit der Software *Kubernetes* entwickelt. Das OpenSource Projekt ist mit dem Hintergrund von Google entstanden, um eine große Anzahl an Anwendungen/Container leicht und schnell deployen, starten und verwalten zu können. Dabei standen folgende Anforderungen im Fokus: schnell und vorhersehbar Anwendungen zu deployen, eine komfortable Skalierung der Anwendungen, problemloses bereitstellen von neuen Features und Auslastungsminimierung der Ressourcen. Vor allem die automatische Skalierung spielt eine zentrale Rolle bei Kubernetes, um Container effektiv an die benötigte Auslastung anzupassen. Verglichen zu anderen Orchestrierungswerkzeugen arbeitet Kubernetes unabhängig von einer Kontrollinstanz und ist nicht an eine Umgebung gebunden. Docker Swarm setzt beispielsweise Docker als Umgebung voraus, wohingegen Kubernetes auch andere Virtualisierungssysteme nutzen kann oder auch direkt mit Ressourcen umgehen kann, die nicht in einer Containerumgebung arbeiten. In Kubernetes bezeichnen *Pods* die kleinste deploybare Einheit auf *Nodes*. Nodes sind virtuelle oder physische Maschine, die zu einem *Cluster* verbunden werden können. Kubernetes kann über verschiedene grafische Oberflächen gesteuert oder direkt über die API via Kommandozeile kommuniziert werden.

Rancher Rancher ist ein Plattform zum Managen von Containern und gleichzeitiger Überwachung von entsprechenden Maschinen und der Möglichkeit direkt VMs bei Providern provisionieren zu können. Rancher wird mithilfe eines Docker-Containers gestartet und anschließend über die Website konfiguriert. Die Plattform bietet eine große Unterstützung für unterschiedlichen Technologien wie Docker-Compose, Docker Swarm, Kubernetes, etc. an. Die komplette Umstellung der Nutzung von Kubernetes ist mit der Version 2.0 eingezogen, die erst Ende 2017 offiziell released wurde. Zuvor wurde auf die hauseigene *Cattle* Umgebung gesetzt, in der eigene Netzwerklösungen implementiert waren. Über die Oberfläche oder einer REST API ist der Nutzer in der Lage mit der Plattform zu interagieren. Komplette Anwendungsstacks lassen sich mithilfe einer Docker-Compose und einem Mausklick direkt starten. Rancher selbst kümmert sich um den Download (voraussgesetzt die Images sind in einer Docker Registry verfügbar), anschließender Auswahl eines geeignet Hosts und dem Starten der Container. Neue virtuelle Maschinen können dabei leicht über einen einzigen Kommandozeilen Befehl hinzugefügt werden. Voraussetzung ist, dass Docker auf dem System installiert ist. Dabei wird ein Rancher Agent heruntergeladen, der sich anschließend über das Internet oder Netzwerk mit dem Master-Agent verbindet. Ab diesem Zeitpunkt lässt sich das System via Rancher verwalten. Rancher abstrahiert die Komplexität der unterschiedlichen Technologien und bietet ein komfortables und einfachen System für Contai-

ner an [Ran17]. Rancher bietet zudem einen Store an, indem verschiedene Anwendungen direkt heruntergeladen und installiert werden können. Beispiele hierfür wären GitLab oder WordPress.

Cloudify Cloudify setzt den Fokus verglichen zu den vorherigen Werkzeugen nicht so stark auf Container. Vielmehr wird hier eine komplette Orchestration mit Maschinen und Systemen ausgeführt [Clo17]. Als Technologie wird dabei TOSCA (siehe Abschnitt 3.1.1) verwendet. Eine komplette Orchestration wird dabei als *Blueprint* bezeichnet. Der Fokus von Cloudify liegt vor allem in der leichten Nutzung und Verknüpfung von verschiedenen Cloudanbietern und Plattformen. Dem Nutzer soll die Arbeit der Verbindung von mehreren Clouds abgenommen werden und diese Schritte weitestgehend automatisiert werden. Dies bedeutet, dass entsprechende Knoten bei Cloudanbietern provisioniert werden können. Auch bei Cloudistry wird eine Master-Instanz als Verwaltungsmittelpunkt genutzt, die anschließend über eine UI oder die Kommandozeile konfiguriert und verwaltet werden kann. Eines der Kernfunktionalitäten von Cloudify ist zudem die Erweiterbarkeit über Plugins. Ein Plugin kann unterschiedliche Applikationen oder Abstraktion realisieren. Durch die Abstraktion wird eine Wiederverwendbarkeit erzeugt [Clo17].

Alien4Cloud Ähnlich zu Cloudify liegt der Fokus bei Alien4Cloud auf dem schnellen Deployment und der Verwaltung von Infrastrukturen (mit genauen Spezifikationen wie RAM, CPUs etc) und Clouds. Dadurch bieten sich mehr Variationsmöglichkeiten wie etwas deployed werden soll, gleichzeitig steigt damit aber auch die Komplexität für den Nutzer, da er sich um mehr Informationen Gedanken machen muss. Alien4Clouds Funktionsumfang übersteigt dabei das einfache Containermanagement. Damit ist die Zielgruppe der Nutzer auch nicht auf den einfachen User ausgelegt, sondern eher einem Entwickler oder Administrator [ALI17].

3.2.2 IBM Bluemix

Bluemix ist eine von IBM entwickelte Cloudplattform als PaaS [DVE+16], um Anwendungen flexibel nutzen zu können. Dabei ist die Palette an Funktionen von Bluemix und der IBM Cloud groß. Es reicht von Bereitstellungen der Infrastrukturen, VMs, Speicher, Netzwerk, Nutzung der großen künstlichen Intelligenzsysteme von IBM. Dabei wird der DevOps Gedanke verfolgt, um eine nahtlose Zusammenarbeit zwischen Entwicklern und Wartung zu ermöglichen. Durch eine große Unterstützung verschiedener Programmiersprachen und Technologien müssen sich die Anwendungsentwickler nicht mehr mit der Infrastruktur beschäftigen. Zudem bietet Bluemix verschiedene Tools für das Testen und „Rapid Deployment“ standardmäßig an [DVE+16]. Ein Überblick der Architektur von Bluemix ist in Abbildung 3.4 dargestellt. Da Bluemix als PaaS dient, muss sich der Nutzer nur noch um Anwendungen und Daten kümmern. Der Rest wird von Bluemix verwaltet. IBM Softwarelayer dient dabei als IaaS.

3.2.3 Virtual Fort Knox

Mit Virtual Fort Knox (VFK)[2] wurde eine Cloud-IT-Plattform vom Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA entwickelt, die es ermöglicht Geschäftsprozesse aus der

[2] https://www.virtualfortknox.de

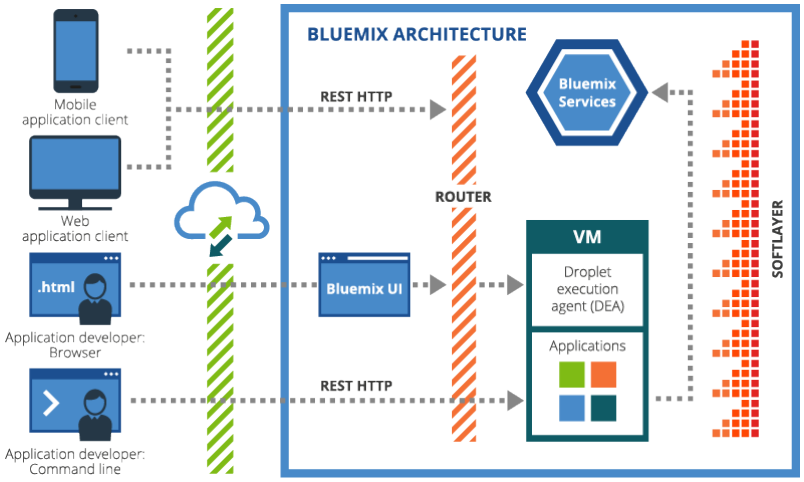

Abbildung 3.4: Überblick über die Architektur hinter IBM Bluemix. [Ins18]

Industrie abzubilden. Dabei können verschiedene Services verbunden werden, um Lösungen für genaue Kundenanforderungen zu liefern. Die Kosten und der Aufwand sollen reduziert und die Lösungen in einem Cloud Umfeld zugänglich gemacht werden. Mit Hinblick auf Industrie 4.0 und dem Trend des Cloud Computings wurde die Plattform entwickelt. Die Sicherheit und der deutsche Datenschutz haben eine wichtigen Punkt bei der Entwicklung eingenommen. Virtual Fort Knox verfolgt dabei folgende Strategie: Zeitaufwand im Minutenbereich, leichte Evaluation von Services, Marktplatz für Anwendungen und Lösungen und die virtuelle Bereitstellung, Skalierbarkeit und Erweiterbarkeit [HWSB13]. Im Zentrum der Architektur steht der Manufacturing Service Bus, der die Kommunikation von den Services selbst, aber auch den integrierten Maschinen ermöglicht (siehe Abbildung 3.5). Services und der Bus werden dabei in einer Cloud-Umgebung betrieben.

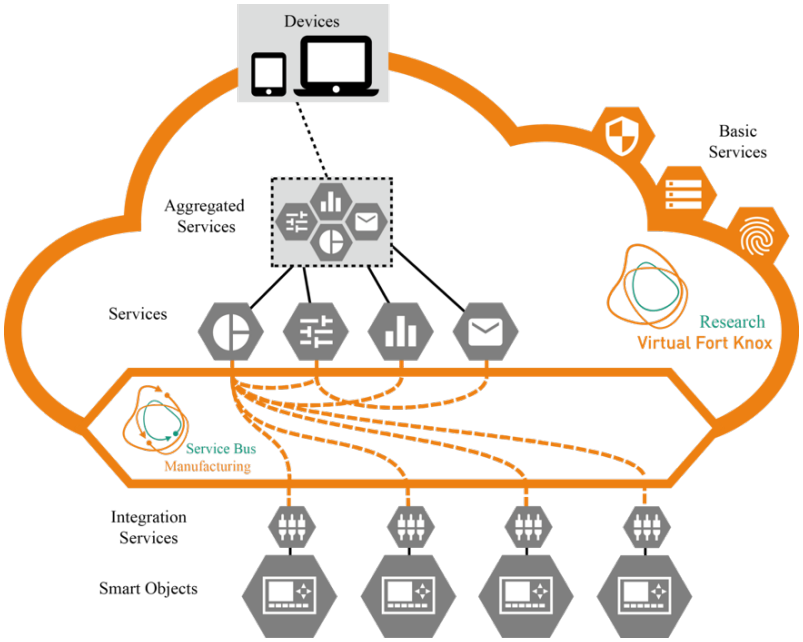

Abbildung 3.5: Überblick über die Architektur hinter VFK. [Kno18]

Die Produktionsprozesse einer Fabrik sollen durch VFK optimiert werden. Als Beispiel bietet VFK folgende Lösungen an: Nutzung von Echtzeitdaten, effiziente Materialbeschaffung und Automatisierung und Digitalisierung der Maschinen [HWSB13].

Eine Anknüpfung an die digitale Produktion ist Sense&Act [3], das ebenfalls mit dem Fraunhofer IPA erforscht wird. Das Ziel dabei ist es nach dem Aktion-Reaktion Prinzip Regeln für die Produktion definieren und erstellen zu können. Dabei liefern Sensoren (Sense) Events, die einen Zustand widerspiegeln und ein Aktor (Act) entsprechend darauf reagieren soll. Als Beispiel kann ein Material das Lager erreichen (Sensor) und anschließend soll dieses in das Warensystem eingebucht werden (Aktor).

3.2.4 Projekt piCASSO

PiCASSO ist in Zusammenarbeit mit dem Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen der Universität Stuttgart im Jahr 2013 entstanden. Es ist keine öffentliche verfügbare Version der Plattform verfügbar, sodass eine Nutzung oder Weiterentwicklung nicht ohne Weiteres möglich ist. PicASSO steht für *Industrielle CloudbASierte SteuerungsplattfOrm für eine Produktion mit cyber-physischen Systemen*[4]. Die Motivation für das Projekt war es die vielschichtige Struktur von Steuerungseinheiten zu vereinfachen, damit diese leicht miteinander genutzt werden können [LS17]. In einer Steuerungsebene werden mit den vorhandenen Istwerten mithilfe von Algorithmen Sollwerte für die darunter liegenden Ebenen berechnet. In der heutigen Produktion wird die Steuerung Top-Down durchgeführt. Die Steuerungselemente sind zumeist recht statisch und fest vordefiniert, was einem Unternehmen die Möglichkeit nimmt dynamische Verbindungen zu nutzen. Zusammengefasst war das Ziel eine Bereitstellung einer Plattform für die Steuerungseinheiten mit einer skalierbaren Rechenleistung je nach Komplexität der verwendeten Algorithmen. Dabei soll die Cloud für die einzelnen Steuerungselemente verwendet werden.

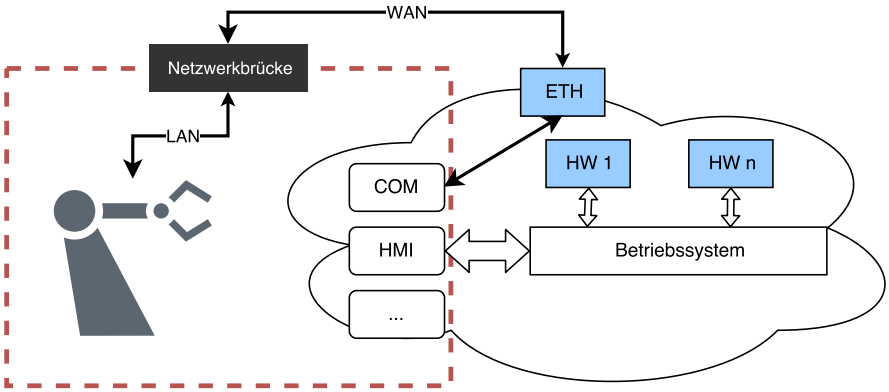

Abbildung 3.6: Ungefährer Aufbau einer piCASSO Architektur mit Maschinen.

In Abbildung 3.6 ist die Architektur mit Zusammenspiel einer Maschine dargestellt. Dabei bildet die Maschine zusammen mit einigen Services wie dem Kommunikationsmodul (COM), die in einer

[3] https://www.ipa.fraunhofer.de/de/ueber_uns/Leitthemen/industrie-4-0/anwendungen-und-projekte/sense-act.html
[4] https://www.projekt-picasso.de/

Cloud liegen, die cloudbasierte Maschine ab (roter Rahmen). Durch genau definierte Schnittstellen (LAN, WAN) findet dabei die Kommunikation statt und in diesem Punkt wird auch für die nötigen Sicherheitsvorkehrungen und Datenschutz gesorgt. Die Netzwerkbrücke zwischen der Maschine und der Cloud kümmert sich dabei auch um die Synchronisation von Echtzeit- und Nichtechtzeitdaten.

3.2.5 SmartOrchestra

Abbildung 3.7: SmartOrchestra soll alle möglichen Geräte und Maschinen generisch anbinden können. [Sma18]

Das Forschungsinstitut in Stuttgart Institut für Architektur von Anwendungssystemen (IAAS) hat mit dem Ziel eine offene und sichere Plattform für Services und Dienste inklusive Marktplatz zu entwickeln das Projekt SmartOrchestra initiiert[5]. Zudem soll es möglich Daten aus Maschinen mithilfe von Services orchestrieren zu können und so einen Prozess zu erzeugen mit dem eine definierte Aufgabe erledigt werden kann. Dabei soll SmartOrchestra einerseits ein Katalog für Services sein und andererseits eine Plattform für Verbindung von Sensordaten und Applikationen. Die Plattform soll über alle Geräte bedient und die Abhängigkeit zu Herstellersoftware minimiert werden. Die angebundenen Maschinen und Geräte sollen dabei leicht und generisch angeknüpft werden können. Dabei dient TOSCA als Grundlage der Services und deren Verbindungen (siehe Abschnitt 3.1.1) und daraus resultierend OpenTOSCA für das gesamte Ökosystem.

3.2.6 SeaClouds

SeaCloud ist ein Forschungsprojekt der EU mit dem Ziel die Probleme des Multi-Deployments bei unterschiedlichen Cloudanbietern und -plattformen richtig organisieren und verwalten zu können [BIS+14]. Dabei sollen die Cloud-Standards eingehalten werden, damit die Interoperabilität zwischen den Anbietern möglichst fehlerfrei bleibt. Ebenso nimmt die Migration von Anbieter zu Anbieter einen sehr zentralen Punkt in SeaClouds ein. Die Architektur von SeaClouds fokussiert sich neben dem Orchestrieren und Deployen überwiegend auf das Monitoring und Verwalten von laufenden Deployments. Laufende Services sollen sich nicht nur zur Deployzeit konfigurieren lassen, sondern je nach Bedarf zu gewissen Maßstäben auch zur Laufzeit. Mithilfe einer geeignet API kann mit der Plattform interagiert werden. Dabei wird *„Agility after Deployment“* verfolgt.

[5] http://smartorchestra.de

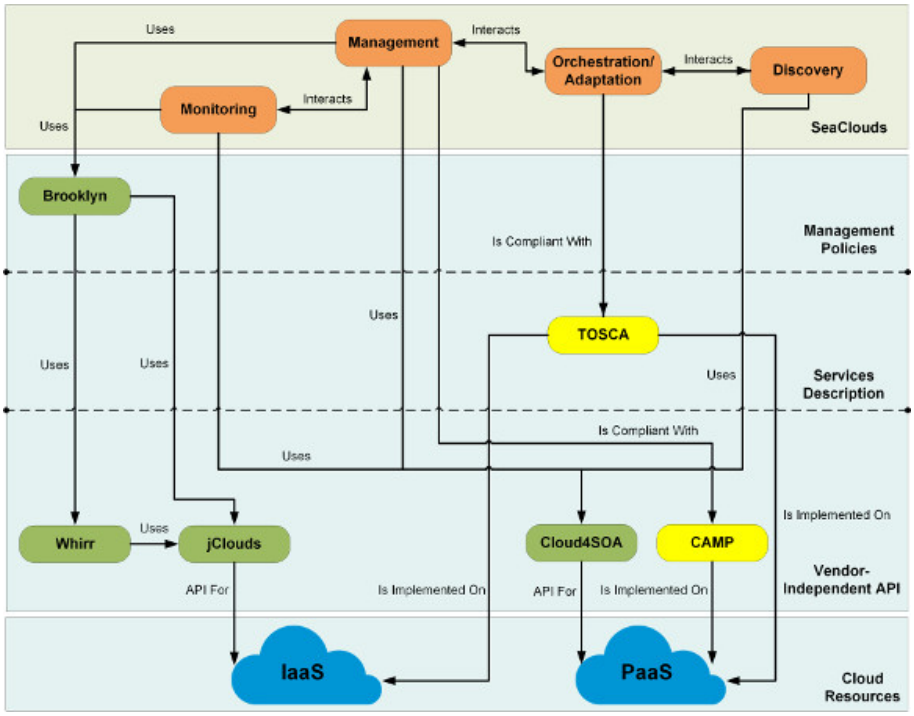

Abbildung 3.8: SeaClouds greift auf viele unterschiedliche Werkzeuge und Frameworks zurück, um die verschiedenen Cloud-Anbieter leichter verwalten und nutzen zu können. [Bro15]

Wie in Abbildung 3.8 zu sehen ist, greift SeaClouds auf eine große Anzahl an unterschiedlichen Technologien und Konzepte zurück, um mit den IaaS und PaaS zu arbeiten. Einige dieser Technologien (Whirr, Cloud4SOA) sind mittlerweile veraltet oder wurden nie konkret umgesetzt. SeaClouds abstrahiert alle Schichten zwischen den Cloud-Anbietern und dem Nutzer. Die komplette Deployment-Strategie von SeaClouds basiert dabei auf dem TOSCA Standard (siehe Abschnitt 3.1.1).

3.2.7 Adamos

Adamos steht für *ADAptive Manufacturing Open Solutions* und besteht aus einer Allianz von weltmarktführenden Unternehmen. Gegründet wurde es von DMG MORI, Dürr, Software AG und ZEISS sowie ASM PT. Das Ziel der Allianz ist es einen Standard zu schaffen, der das Wissen von Maschinenbau, IT und Produktion verbindet [Ada17]. Die zwei Kernpunkte von Adamos sind:

IIoT-Plattform Das ist die technologische Basis von Adamos für angebundene Marktplätze und bietet grundlegende Funktionalitäten als PaaS an. Dabei werden Dienste für Analyse und Verwaltung von Daten angeboten. Zudem können Produktionsprozesse firmenübergreifend erstellt werden. Die Plattform wurde als offene und skalierbare Plattform konzipiert und bietet Funktionen wie maschinelles Lernen, Echtzeit-Analysen und weitere Punkte an. Auf die Sicherheit des Systems und der Daten wird mittels modernen Standards hoher Wert gelegt.

App Factory Entspricht einer Entwicklungsumgebung in das Wissen und die Expertise der Unternehmen vereinigt werden und schafft eine schnelle Entwicklungsmöglichkeit. Dabei kooperieren Maschinenbau- und Softwareunternehmen, um eine solide Lösung für beide Bereiche zu schaffen. Bei gemeinsamen Entwicklungen können dadurch auch Kosten geteilt und damit gespart werden.

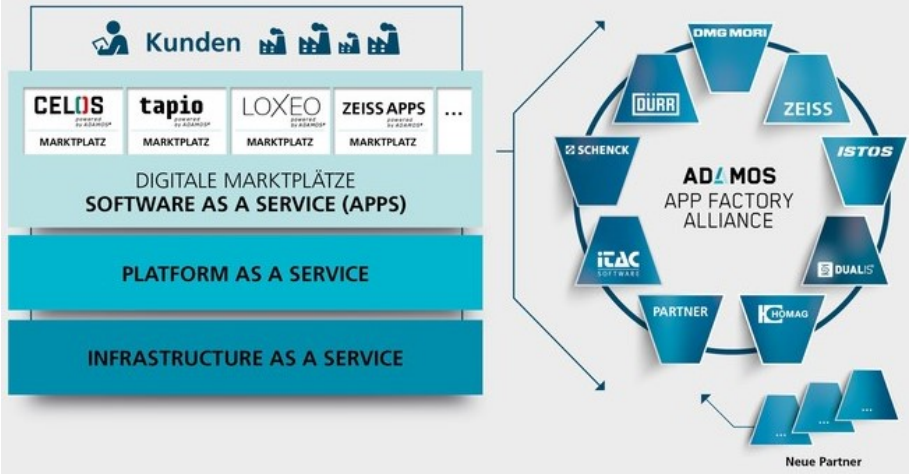

Abbildung 3.9: Die Plattform stellt dabei den Marktplatz, sowie Infrastrukturen und Basis bereit. Partnerunternehmen können dabei über die App Alliance angebunden werden und den Marktplatz entsprechend erweitern. [Gmb18]

Neben den Hauptpunkten wird ebenso Wert auf eine herstellerneutralen Plattform, einen großen Marktplatz für Apps und einen hohen Kompetenzgrad gelegt. Wie in Abbildung 3.9 zu sehen ist, können neue Partner leicht an das bisherige Portfolio angekoppelt werden und erweitern den vorhanden Marktplatz durch Wissen und neuen Apps weiter.

3.2.8 Axoom

Axoom ist ein Unternehmen mit ähnlichen Zielen wie Adamos. Maschinen sollen leicht an das System angebunde, integriert und anschließend verwaltet werden[6]. Dabei bieten sie die *IoT-Plattform* an.

In Abbildung 3.10 ist ein Überblick über die Oberfläche von Axoom zu sehen. Mithilfe der Oberfläche lassen sich Maschinen anbinden, die Plattform modular erweitern oder Echtzeitanalysen durchführen. Mithilfe von Daten und Visualisierungen sollen die Anwender den Zustand des Prozesses leicht überwachen können. Mit dem Ziel Richtung Industrie 4.0 und Kernthemen wie Smart Factory und Smart Machine werden somit ähnliche Ziele wie Adamos verfolgt. Durch ein bereitgestelltes SDK sollen Anwendungen leicht für den von Axoom bereitgestellten Store entwickelt und beigetragen werden. Es soll ein gesundes Ökosystem entstehen mit Axoom als Mittelpunkt. Axoom ist es wichtig, dass eine hohe Nachhaltigkeit der Systeme garantiert wird. Das soll durch Monitoring-Konzepte und Kontrollzyklen realisiert werden. Axoom verweist

[6] http://www.axoom-solutions.com/

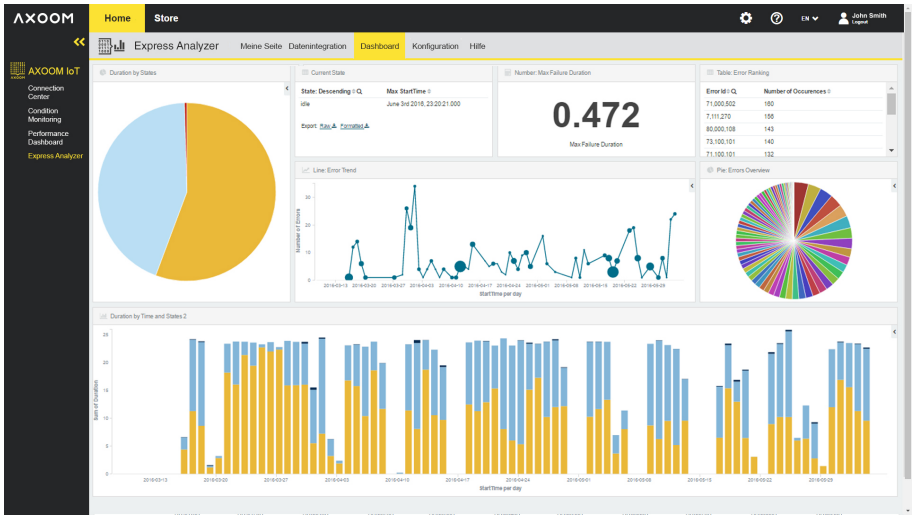

Abbildung 3.10: Eine Ansicht der Axoom IoT-Plattform. Mithilfe vieler Visualisierungen sollen Zustand von Maschinen und Produktion überwacht werden. [Tec18b]

auf Sparmöglichkeiten in verschiedenen Bereichen. Im Schnitt kann dabei ungefähr 10% eingespart werden. Die Bereiche umfassen direkten und indirekten Fertigungsbereich, Qualitätskosten, Reparatur- und Wartungskosten der Maschinen.

3.3 Zusammenfassung

In diesem Kapitel wird der wissenschaftliche und technische Stand zusammengefasst und verglichen. Zusammengehörige Themen werden dabei geclustert und nach geeigneten Eigenschaften mithilfe von Harvey Balls bewertet. Anschließend werden die Kandidaten gegeneinander abgewogen und in ihrem Funktionsumfang verglichen. Die Bewertungen basieren nicht auf Umfragen, sondern entsprechen der eigenen Einschätzung. Je nach Möglichkeit wird dabei ein Präferenzkandidat hervorgehoben.

Rein wissenschaftliche Ansätze im Themenbereich der Industrie und der Digitalisierung sind kaum zu finden. Es sind eher Teilbereiche eines verwandten Themas wie beispielsweise den Service-Definition oder Sicherungsideen einer Software zu finden, die durch anfängliche wissenschaftliche Forschungen gestartet sind. Viel öfter entstehen hier Arbeiten im technischen Bereich mit direkten Lösungen oder Ansätzen, die in Zusammenarbeiten mit Industriepartnern entstehen.

Im aktuellen Stand der Technik sind eine Vielzahl von verwandten Entwicklungen und Lösungen zu beobachten, die sich mit einem ähnlichem Themenbereich beschäftigt. Dabei liegen die Fokussierungen oftmals in anderen Bereichen als im Vergleich zu dieser Arbeit. Zudem sind die vorgestellten Anwendungen überwiegend unter Verschluss der jeweiligen Firmen und daher nicht für die Erweiterung oder Veränderung verfügbar. Darüber hinaus sind die meisten Projekte nicht öffentlich verfügbar, sodass es keine Möglichkeit gibt das Projekt genauer untersuchen und analysieren zu können. Dadurch sind die großen Projekte nicht zum weiterverwenden geeignet. Die Frameworks und Tools hingegen könnten für entsprechende Teilbereiche genutzt werden, um die Entwicklungsarbeiten zu vereinfachen und zügig ein Ergebnis zu erreichen. Nichtsdestotrotz

ist leicht zu erkennen, dass das Themengebiet aktuell ist und einen hohen Stellenwert in der Industrie einnimmt, da der Bedarf ansteigt. Die Industrie hat ein großes Interesse daran Plattformen nutzen zu können, die die Arbeitsschritte und Fertigungsprozesse automatisieren, vereinfachen und optimieren können mit dem Ziel der Kostenreduktion und Gewinnsteigerung.

3.3.1 Service-Definition

Für den ersten Vergleich, der Service-Definition, sind nur wenige verbreitete Definition verfügbar. Das hängt damit zusammen, dass bereits einige Standards verfügbar sind, und oftmals eigene Definition erstellt werden, die speziell an die Software oder das Problem zugeschnitten sind. Diese verbreiten sich dadurch nicht und bleiben in nur einem Kontext bestehen. Folgende Eigenschaften nehmen dabei einen wichtigen Standpunkt bei der Bewertung ein.

Flexibilität beschreibt, wie einfach sich Elemente verknüpfen, wiederverwenden etc lassen.

Standardkonform gibt an, ob die Definition einem Standard entspricht.

Erweiterbarkeit Wie gut lässt sich die Definition um eigene Konstrukte erweitern.

Bekanntheit Wie bekannt oder verbreitet ist die Definition.

Benutzbarkeit Ist die Definition leicht nutzbar?

Komplexität Ist die Defintion komplex und sehr vielfältig aufgebaut, was ein gesamtes Verständnis erschwert?

Akzeptanz Wie gut würde die Definition im Industrie-Kontext akzeptiert werden? (Starke Kopplung an Komplexität und Benutzbarkeit)

	WSDL	TOSCA
Flexibilität	●	●
Standardkonform	●	●
Erweiterbarkeit	●	●
Bekanntheit	●	◑
Benutzbarkeit	◑	◑
Komplexität	◑	●
Akzeptanz	◑	○

● = Erfüllt, ◑ = Teilweise erfüllt, ○ = Nicht erfüllt, – = Unbekannt

Tabelle 3.1: Bewertungskriterien für Service-Definitionen.

Die verfügbaren Definition sind beide erweiterbar und flexibel, jedoch ist ein Problem hinsichtlich der Benutzbarkeit, Komplexität und Akzeptanz zu erkennen. Die Integration eines der Definition würde für die prototypische Umsetzung aufwendig werden und anschließend Akzeptanzprobleme bekommen. Daher sind WSDL und TOSCA weniger für den Prototypen geeignet.

3.3.2 Containermanagement

Der nächste Bereich ist das Containermanagement samt Konzepten. Hier werden bewusst nur größere Softwarelösungen miteinander verglichen. Neben den Eigenschaft wird auch angegeben welche Formate die jeweiligen Lösungen unterstützen und welche zusätzlichen nützlichen Funktionalitäten inkludiert sind. Folgende Eigenschaften werden bewertet:

Skalierbarkeit beschreibt die allgemeine Fähigkeit Anwendungen skalieren zu können. Es wird kein Fokus auf automatische Skalierung nach bestimmten Ereignisse gelegt.

Einfachheit drückt aus, wie einfach die Software anzuwenden und zu erlernen ist.

Sicherheitsmechanismen gibt an, ob Sicherheitsmechanismen integriert sind, um mögliche Angreifer oder Zugriff von Dritten zu erschweren. Meist umfasst dies Kommunikation über SSL Verbindungen, Login-Mechanismen oder anderen Schutzschichten.

Wie an der Tabelle Tabelle 3.2zu erkennen ist, bieten viele der Plattformen einen großen Funktionsumfang mit den notwendigen Elementen an. Ein umfangreiches Knotenmanagement ist nur bei einem kleinen Teil vorhanden, dies bedeutet das bei einigen Plattformen die Management Funktionalität selbst übernommen werden müsste. Ebenso spielt auch die Wahl der unterstützten Technologie eine wichtige Rolle. Durch die sehr weite Verbreitung und Bekanntheit hat Docker einen großen Vorteil. Es gibt viele Foren für Problemstellungen und Tutorials für bekannte Probleme.. TOSCA hingegen ist verglichen mit Docker noch kaum verbreitet. Dadurch sticht die Plattform Rancher mit dem enthaltenen Funktionsumfang stark hervor.

	Docker Swarm	Kubernetes	Rancher	Cloudify	Alien4Cloud
Docker	●	●	●	○	○
TOSCA	○	○	○	●	●
Integrierte UI	○	◑	●	●	●
REST-Schnittstelle	●	●	●	●	●
Skalierbarkeit	●	●	●	●	○
Einfachheit	◑	◑	●	◑	○
Sicherheitsmechanismen	●	●	●	●	●
Provisionierung	◑	◑	●	◑	◑
Knoten-Management	○	◑	●	●	○

● = Erfüllt, ◑ = Teilweise erfüllt, ○ = Nicht erfüllt, – = Unbekannt

Tabelle 3.2: Bewertungskriterien der möglichen Containermanagement-Plattformen.

3.3.3 Technische Lösungen

Der letzte Blickpunkt (siehe Tabelle 3.3) sind die verfügbaren Lösung aus Industrie und Forschung. Die meisten dieser Anwendungen sind bereits verfügbar und werden von Kunden genutzt. Diese

bieten viele verschiedene Funktionalitäten an, die in der Industrie benötigt und gewünscht sind. Die Anbieter mit dem größten Umfang an Funktionalität sind Adamos und Axoom. Da diese jedoch ganze Firmen hinter sich haben, ist keinerlei Informationen der genauen Software und der entsprechenden Umsetzung vorhanden. Die anderen Anwendungen bieten nicht so einen großen Funktionsumfang an oder konzentrieren sich auf andere Bereiche. IBM Bluemix ist beispielsweise eine PaaS ohne direkte Maschinenanbindung. Virtual Fort Knox ist ein interessanter Kandidat, da hier eine Orchestrierung von Maschinen über die Cloud im Fokus steht. Jedoch ergibt sich das selbe Problem wie bei Adamos und Axoom mit fehlender Verfügbarkeit.

Die Verwendung oder Weiterentwicklung der bestehenden großen Produkte kommt durch die fehlende Verfügbarkeit der Quellen nicht in Frage. Stattdessen können die Ideen wie beispielsweise einer möglichen Marktplatzanbindung in der Konzeption verwendet werden.

	IBM Bluemix	Virtual Fort Knox	piCASSO	SmartOrchestra	SeaClouds	Adamos	Axoom
OpenSource	○	○	○	○	○	○	○
Aktiv	●	●	○	●	–	●	●
Multi-Cloud	○	○	○	○	◑	○	○
Maschinenanbindung	○	●	–	○	○	●	●
Kostenpflichtig	●	●	–	–	–	●	●
Marktplatzanbindung	●	○	○	●	○	●	●
Prozessorchestrierung	○	●	○	●	○	●	●
Monitoring	●	●	–	–	○	●	●

● = Erfüllt, ◑ = Teilweise erfüllt, ○ = Nicht erfüllt, – = Unbekannt

Tabelle 3.3: Bewertungskriterien für industrielle Produkte.

4 Konzept

Zu Beginn ist es wichtig die Anforderungen und Rahmenbedingung zu erfassen, die von ein Plattform für die Verknüpfung von Services, Maschinen in beliebigen Clouds nötig sind. Anschließend wird sich diese Arbeit auf einzelne Teile der Plattform konzentrieren und verschiedene Lösungsansätze vergleichen. Dies umfasst im größten Teil die Services und ihrem Aufbau, der Veränderungen eines Services, die Kommunikation von internen/externen Komponenten und der Sicherung der verschiedene Teile und Schichten. Die Serverarchitektur wird dabei nur kurz angeschnitten.

4.1 Anforderungen an die Plattform

Durch das Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen und der Kooperation mit verschiedenen Industriepartnern, waren bereits viele Erfahrungen zu den benötigten Funktionalitäten und Wünschen seitens der Industrie vorhanden. Zusätzlich wurden Konzepte aus den verwandten Projekten (siehe Abschnitt 3.2) verwertet.

Um ein besseres Verständnis für die einzelnen Anforderungen zu erhalten, sind die Stakeholder wichtig. Diese umfassen: *Anwender*, *Service-Entwickler*, *Entwickler* und *Industrie*. Anwender umfassen dabei alle Nutzer vom einfachen Maschinenbauer bis hin zum Administrator, der die Plattform mit seinen entsprechenden Zugangsdaten konfiguriert. Die Service-Entwickler kümmern sich um die Entwicklung und den Beitrag von Services zur Plattform. Die Plattform selbst wird von Entwicklern erweitert und gewartet. Und zum Schluss die Industrie, die sich an der Optimierung der Prozesse innerhalb der eigenen Produktion interessiert und die eigenen Maschinen in die Plattform integrieren möchte.

Multi-Cloud Wie aus dem übergeordnetem Projektnamen vom ISW hervorgeht, ist die Anbindung an mehrere Clouds fundamental. Der Nutzer soll nicht direkt dadurch beschränkt sein, dass er beispielsweise nur mit Amazon AWS oder Google Cloud arbeiten kann. Im Idealfall sollte es möglich sein die Plattform mit beliebigen Cloud-Anbietern und auch lokalen Ressourcen wie virtuelle Maschinen nutzen zu können. Dies ist relevant, wenn der Standort beim Deployment beschränkt werden muss und dadurch ein Provider genutzt werden kann, der das beste Preis-Leistungsverhältnis anbietet. Neben der Multi-Cloud Fähigkeit sollen zudem die bekannten Eigenschaften von Cloud Computing, wie in Abschnitt 2.4 erläutert, möglich sein. Dazu zählt die Skalierbarkeit von Services (im Idealfall je nach Bedarf) oder Isolation der einzelnen Services, damit diese bei Bedarf ausgetauscht werden können.

Services Der Begriff eines Service entspricht dabei einer Komponente in der Plattform, die etwas verarbeitet, überwacht, steuert oder Daten generiert. Jede Form von Arbeit, Anbindung etc. wird dabei als Service realisiert. Ein Service muss dabei stets definiert und von einem Entwickler umgesetzt werden. Services sollen unter sich und auch mit etwaigen äußeren Komponenten kommunizieren können. Da die Services von verschiedenen Entwicklern entwickelt werden sollen, müssen diese ohne großen Aufwand umzusetzen und zu integrieren sein. Dies soll beispielsweise durch eine geeignete Anleitung oder ähnliches erfolgen. Um einen Service zu entwickeln, der in Verbindung mit einem anderen Service gebracht werden soll, müssen entsprechende Informationen bereitgestellt und ersichtlich sein. Ein Service entspricht also einem Grundbaustein der Plattform mit dem der Nutzer arbeiten wird.

Externe Komponenten Wie bereits im Teil der Services erwähnt ist es wichtig, dass externe Komponenten oder bestehende Softwaremodule ohne großen Aufwand an die Plattform angebunden werden können. Dabei sollen die externen Komponenten nicht modifiziert werden müssen, sondern über eine einfache Schnittstelle an die Plattform angebunden werden (vorausgesetzt sie stellen eine externe Schnittstelle zur Verfügung). Als Beispiel kann eine aktive Datenbank genutzt werden, die mittels Nutzername und Password angebunden wird. Dies könnte ein Service in der Plattform lösen. Zu den externen Komponenten können neben dem einfachen Beispiel einer Datenbank verschiedene Anwendungen, Kommunikationsplattformen oder Generatoren in Frage kommen. Generatoren sind dabei Komponenten, die kontinuierlich Daten produzieren. Zu den externen Komponenten zählen auch Maschinen, die in der Plattform genutzt werden sollen. Mithilfe geeigneter Schnittstellen sollen Maschinendaten an ein externes Netzwerk senden und gegebenenfalls empfangen.

Prozessmodellierung Diese Anforderungen orientiert sich an bekannten Methoden wie bspw. Business Process Model and Notation (BPMN). Mithilfe von verfügbaren visuellen Elementen wird dabei ein Geschäftsprozess mit Bedingungen, Verzweigungen und anderen verfügbaren Elementen abgebildet. Der Prozess hat dabei einen Start und mögliche Endpunkte. Dabei soll ähnlich zu einer Modellierung von Geschäftsprozessen auch in der Plattform ein Prozess oder Workflow vom Anwender erstellt werden, der durch visuelle Elemente unterstützt und geführt wird. Gleichzeitig sieht der Nutzer auf einen Blick wie der komplette Prozess aussieht und an welchen Stellen bestimmte Daten verarbeitet werden. Zusätzlich bietet es sich dann an Zusatzinformationen zu den einzelnen Elementen anzuzeigen. Dazu kann zum Beispiel der Zustand eines Services zählen oder die aktuelle Position der Verarbeitung. Dies könnte durch Analysen erweitert, werden sodass beispielsweise mögliche Engpässe erkannt werden.

Nachrichtenbasierte Kommunikation Für eine möglichst lose Kopplung der Services ist es wichtig, dass die Services nicht über feste Verbindungen wie zum Beispiel Programmcode aufgebaut werden. Daher wurde zu Beginn als Anforderung eine nachrichtenbasierte Kommunikation vorgeschlagen und festgelegt. Durch die Verwendung von Nachrichten sind die Service sehr lose verbunden und müssen nur wissen zu welchen Kanälen oder Queues sie sich verbinden müssen. Die restlichen Arbeiten werden von einer Middleware abgenommen. Diese umfassen Sicherungen der Nachrichten, Verifikation über Erhalt und Verarbeitung von Nachrichten und vielen weiteren Vorteile. Gleichzeitig ist die zusätzliche Middleware ein Nachteil. Es ist eine weitere Komponente, die verwaltet und gewartet werden muss.

Marktplatz - Service-Store Nachdem Services, deren Kommunikation und eine Modellierung definiert wurde, ist es auch wichtig auf entsprechende Services zugreifen zu können. Angelehnt an den Google PlayStore oder dem Apple AppStore bietet es sich an einen eigenen Marktplatz, dem Service-Store, bereitzustellen. Dabei können je nach Anforderungen unterschiedliche Preismodelle, Kaufmöglichkeiten oder Leihmodelle integriert werden. Neben dem Service-Store der Plattform selbst (dem Zentralbezugspunkt für Services) soll es möglich sein die Quellen für Service mit eigenen zu erweitern. Angelehnt an das Modell von Adamos Abschnitt 3.2.7 bei dem Unternehmen sich in den Marktplatz integrieren konnten. Im Rahmen der Arbeit soll jedoch kein funktionsfähiger Service-Store umgesetzt werden.

Datensicherheit Einen sehr wichtigen Punkt bei Bereitstellung einer Plattform mit verschiedenen Anwendung aus einem Service-Store nimmt die Sicherheit der Daten ein. Dabei erwarten verschiedene Nutzer entsprechende Sicherheiten: der Service-Entwickler, der Plattformnutzer und der Plattformbetreiber. Die Plattform selbst sollte in einer sicheren Umgebung betrieben werden und von sich selbst aus keinen direkten Zugriff auf die Daten der Plattform, Services und der Nutzer ermöglichen. Alle Informationen sollen durch entsprechende Sicherheitsmechanismen ausreichend geschützt werden. Zudem sollen alle laufenden Services Sicherheit durch die Ausführungsumgebung gewährleisten. Der Kommunikationskanal der gesamten Umgebung sollte ebenso durch aktuelle Sicherheitsstandard geschützt werden. Wie in Abschnitt 3.1.3 beschrieben, bietet es sich an verschiedene gesondert zu schützen.

Weitere Anforderungen Neben den vorgestellten größeren Anforderungspunkten gibt es noch einige weitere, die jedoch nicht im Detail erläutert werden. Dazu zählt beispielsweise eine gute *Erweiterbarkeit* durch einen modularen und sauberen Aufbau der Plattform und der Architektur. Bei Bedarf sollen neuen Funktionalität einfach und schnell in das Projekt integriert werden können. Dabei kann das Pluginsystem von Eclipse als Ortierung in Betracht gezogen werden. Ein weiterer Punkt ist die *Benutzbarkeit* der Plattform. Da diese nicht nur von geschultem Personal oder Administratoren genutzt werden soll, müssen diese modern und selbsterklärend aufgebaut werden. Der Referenzpunkt dabei war, dass ein Maschinenbauer in der Lage sein sollte innerhalb kurzer Zeit mit der Plattform arbeiten zu können.

4.2 Services

Für die Plattform sind die Services von größter Bedeutung. Daher ist es wichtig zu definieren, wie ein Service dargestellt wird, welche Funktionalität oder Schnittstellen ein Service haben muss und wie dieser definiert wird. Das spielt eine wichtige Rolle bei der Verwendung innerhalb der Plattform, im Service-Store und auch bei den Service-Entwicklern, die den Service entsprechend spezifizieren müssen. Im nachfolgenden werden die Funktionen und Anforderungen an einen Service genauer erläutert.

4.2.1 Anforderungen

Zunächst muss definiert werden, wie ein Service aufgebaut ist. Services sollen eine modulare Einheit darstellen, die eine Arbeit leistet und Daten verarbeiten kann. Zusätzlich ist nötig, dass

einzelne modulare Bausteine miteinander verknüpft werden sollen. Dies kann mit einem Legobaukasten verglichen werden, bei der die einzelnen Steine für sich ein fertiges Element darstellen aber erst im Gesamten, also im zusammengebauten Zustand, das fertige Modell ergeben. Die Steine selbst könnten dabei nochmals durch zum Beispiel Aufkleber erweitert werden. Abbildung 4.1 stellt nun die grundlegenden Elemente einer solchen Komposition dar.

Abbildung 4.1: Die Darstellung von einem Verbund von zwei Services.

Wie zu sehen ist, sind dabei zwei Services miteinander verbunden. Durch die Verbindung wird eine Kommunikation oder Datenaustausch dargestellt. Zusätzlich ist Service B an eine Datenbank angebunden. Das bedeutet, dass Daten zu einer Datenbank übertragen oder von ihr gelesen werden. Aus diesen wenigen Teilkomponenten können wir bereits die notwendigsten Elemente für eine Beschreibung eines Services herauslesen. Dafür dient die Abbildung 4.2.

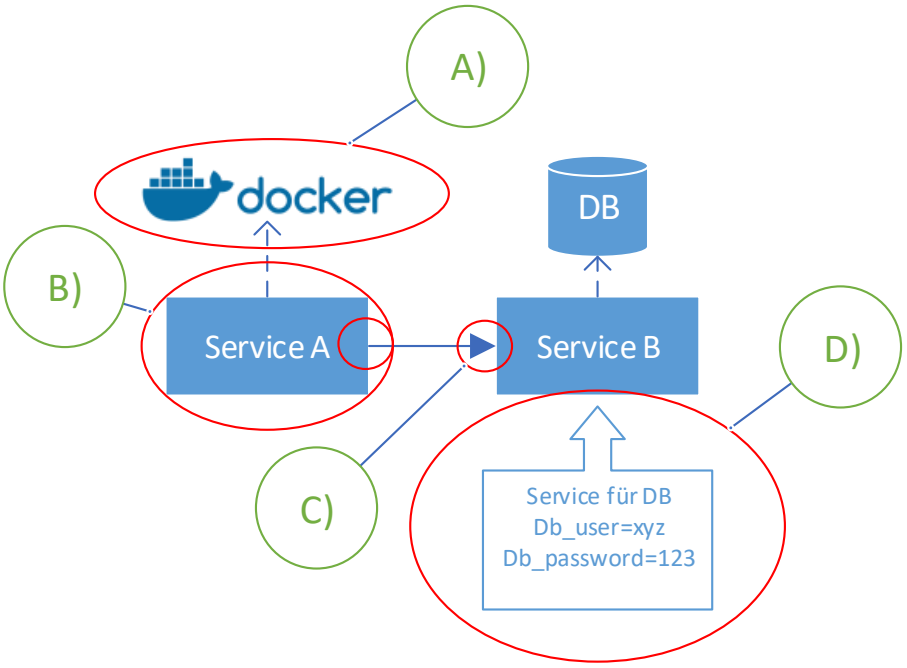

Abbildung 4.2: In dieser Service Komposition sind Informationen erweitert worden und die wichtigen Stellen durch rote Kreise hervorgehoben.

Wie zu sehen ist, sind einige Informationen im Vergleich zur Abbildung 4.1 hinzugekommen.

A) Hier ist zum Beispiel die Darstellung einer Definition für den Ort des Services dargestellt. Wenn der Service im Service-Store abgelegt oder referenziert wird, muss diese Information bei der Definition des Services enthalten sein. Es muss klar sein, wo der Service zu finden ist und wie die notwendigen Informationen geholt werden können. In Form eines Docker-Containers würde hier entweder der Image-Name festlegt werden oder falls es sich um eine andere Docker-Registry handelt, eine entsprechende Konkatenation der Registry und des

Image-Names. Es ist also wichtig zu definieren, wo der Service zu finden ist. Zusätzlich muss berücksichtigt werden, dass es sich um eine möglichst plattformunabhängige Definition handelt. Es wäre zu umständliche für jeden Service die konkrete Laufzeitumgebung definieren zu müssen.

B) stellt die allgemeinen Informationen zu einem Service dar. Darunter können beispielsweise der Name des Services fallen, der Autor oder ein Datum. Die Informationen eines Services sind notwendig, um diesen identifizieren, suchen und verstehen zu können. Der Name des Services sollte am besten aussagekräftig sein. Da mehrere Services miteinander kommunizieren müssen, sollten entsprechende Schnittstellen vorhanden sein.

C) Hier ist die Kommunikation mithilfe eines Pfeils dargestellt, der auf einer Seite eine Pfeilrichtung hat. Dies soll ausdrücken, dass der eine Service Daten bereitstellt und sie nach außen gibt (Service A) und ein anderer Service diese empfangen kann (Service B). Anders ausgedrückt sollte ein Service über Eingänge und Ausgänge verfügen über die er die Daten übermitteln kann. Interessant ist nun in welche Richtungen die Kommunikationen stattfinden könnten. Dies wird in Abschnitt 4.2.2 genauer ausgeführt.

D) Das letzte Teilstück ist in Verbindung zum Service B zu sehen. Durch die Verknüpfung an eine Datenbank ist es wichtig wo die Datenbank zu finden ist und wie eine Verbindung aufgebaut werden kann. Trivial gelöst könnte diese Information im Quellcode hinterlegt sein, dadurch würde der Service jedoch nicht mehr dynamisch genutzt werden können. Daher wäre es vorteilhaft, wenn die Informationen im Nachgang bei dem Deployvorgang ausgewertet werden würden. Dass heißt, dass der Nutzer beim Verknüpfen in der Lage sein soll einen Service mit zusätzlichen Informationen zu bestücken. Wie zu sehen ist, können hier Informationen wie der Datenbank Benutzername und das Password eingetragen werden. Wenn dieser Gedankengang erweitert wird, könnte es sein, dass der Service-Entwickler über diesen Weg Informationen an der Orchestrator bereitstellen möchte. Diese Informationen sind in diesem Fall nur lesbar und nicht vom Nutzer veränderbar.

Durch die vorgestellten Elemente innerhalb einer Service-Orchestration, erreicht die Definition für einen Service eine große Flexibilität und Dynamik.

4.2.2 Kommunikationsrichtungen

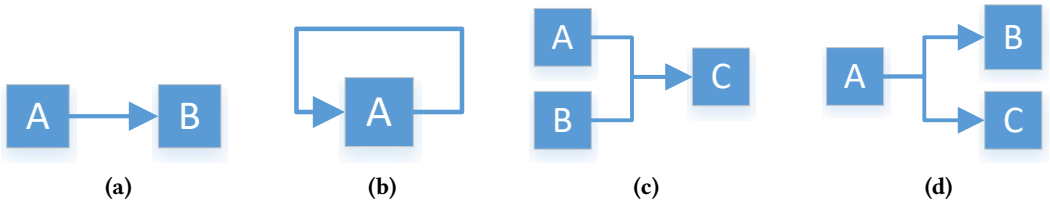

Abbildung 4.3: Darstellung der möglichen Kommunikationsrichtungen und Konstellation.

In Abbildung 4.3 werden die möglichen Kommunikationskonstellation zwischen Services dargestellt. Im einfachsten Fall Abbildung 4.3a kommuniziert Service A mit B, wie bereits zu Beginn erläutert wurde. Es könnte jedoch auch ein Szenario auftreten bei der die Ausgabe eines Services

wieder für sich selbst genutzt werden soll. Dies wird in Abbildung 4.3b über eine zyklische Verbindung dargestellt. Der Ausgang wird mit dem Eingang desselben Services verknüpft. Dies könnte dann nützlich sein, wenn die Daten beispielsweise für die Selbstoptimierung genutzt werden und so ein Algorithmus oder eine maschinelles Netzwerk optimiert wird. Werden Nachrichten zu mehreren Services gesendet, muss dies entsprechend betrachtet werden. Abbildung 4.3c zeigt, dass Service C mehrere Datenquellen zu einem Eingang empfangen kann. In Abbildung 4.3d wird das Gegenstück dargestellt, wo Service A aus einem Ausgang an mehrere Services senden möchte. Zu beachten ist, dass in Abbildung 4.3 davon ausgegangen wird, dass die Eingänge und Ausgänge mit den selben Daten arbeiten.

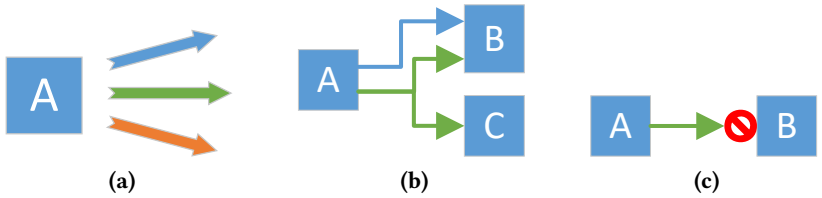

Abbildung 4.4: Services können in der Lage sein verschiedene Daten zu übertragen.

Veranschaulicht wird dies mit Abbildung 4.4. Ein Service ist in der Lage unterschiedliche Daten zu versenden. Die unterschiedlichen Datenarten sind dabei durch verschiedene Pfeilarten in Abbildung 4.4a dargestellt. Werden Services nun verknüpft müssen die entsprechenden Eingänge und Ausgänge verbunden werden, sodass die Kommunikationspartner die Daten empfangen und verarbeiten können. Wie in Abbildung 4.4b zu sehen ist, kann Service B dabei beide Datenformate verarbeiten. Kommt nur der Fall vor, dass die Daten nicht verarbeitet werden können (siehe Abbildung 4.4c), so muss dies entsprechend unterbunden oder anderweitig gelöst werden. Einige Ansätze wie so etwas lösbar wäre, werden in Abschnitt 4.5 beschrieben.

4.2.3 Definition

Nachdem in den vorherigen Abschnitten die Anforderungen für einen Service hervorgehoben wurden, müssen diese nun in einer geeignet Form definiert und festgehalten werden. Im Abschnitt 3.1.1 wurden einige Spezifikationen vorgestellt mit denen Services definiert werden können. Wie in Abschnitt 3.3 dargestellt, sind die meisten verfügbaren Spezifikationen hinsichtlich der Benutzbarkeit und der Akzeptanz schlechter einzuschätzen. Für die Form eines Service werden folgende Eigenschaften benötigt: Allgemeine Informationen, Service-Speicherort, Details zu Eingängen und Ausgängen, sowie Zusatzinformationen. Durch die Flexibilität von WSDL und TOSCA wäre es möglich diese Informationen in die Form zu bringen. Die Komplexität beider Definitionen macht es jedoch umständlich die Services in die Form zu zwingen. Dies muss auch in Anbetracht der Service-Entwickler beachtet werden, da diese die Service Definitionen bereitstellen sollen. WSDL hat hier nochmals einen Nachteil, da es komplett in XML mit den ganzen Namespaces aufgebaut ist. Dadurch wird solch eine Definition nicht gut lesbar. Um die Lesbarkeit zu erhöhen, kann eine separate Darstellung für den Service genutzt werden. TOSCA hat den Vorteil, dass das YAML-Format verwendet werden kann. Dabei werden durch Einrückungen Elemente gruppiert. Dies spart unnötige Zeichen und erhöht die Lesbarkeit. Jedoch besteht bei TOSCA das größte Problem der sehr komplexen und vielfältigen Strukturen. Es würde Potential mit sich bringen, die Service Definition in dem vorhandenen TOSCA Standard umzusetzen. Jedoch bietet

es sich für die prototypische Umsetzung nur bedingt an, da schnell Resultate gewünscht sind. Daher wird in dieser Arbeit vorerst (siehe Kapitel 7 für mögliche Optimierungen) auf eine eigene Service-Definition gesetzt. Die Definition nutzt das YAML Format, da es gut lesbar und leicht zu schreiben ist. Die Definition wurde im Laufe der Entwicklung überarbeitet und angepasst, damit alle Anforderungen mit der Definition umgesetzt werden können. Das bedeutet, dass hier nur an die notwendigsten Informationen gedacht wurde und Erweiterungen möglich sind. Dadurch wird kein extra Tool für die Erstellung benötigt, da dies händisch erledigt werden kann.

Listing 4.1 Abstrakte Definition eines Services mit allen möglichen Eingaben.

```
Service-Bezeichner:
 Metadata:
   Title: string
   Description: string
   Author: string
 Image:
   Registry: string (optional)
   Name: string
   Tag: string (optional)
 Properties:
   - Property-Bezeichner:
      Name: string (eindeutig)
      Description: string (optional)
      DefaultValue: string (optional)
      Required: boolean (optional, default=false)
      ReadOnly: boolean (optional, default=false)
      Type: (optional, [int(64), float(64), string (default), boolean, port])
 Input:
   - Port: Port
 Output:
   - Port: Port
```

In Listing 4.1 wird die vollständige Definition für einen Service dargestellt. Wie zu sehen ist, wird zu Beginn ein beliebiger Bezeichner für den Service gewählt. Das ist nötig, um eventuell mehrere Service innerhalb einer Datei definieren zu können. Dies soll Zeit sparen und auch die Verarbeitung vereinfachen.

Metadata Für die allgemeinen Informationen dient dabei das Feld Metadata. Darin enthalten ist der Service-Name (Title), eine Beschreibung (Description) und der Autor (Author). Dies könnte in Zukunft noch durch weitere wichtige Informationen wie zum Beispiel einer Version erweitert werden.

Image Für den Service-Speicherort wird Docker als Technologie verwendet. Dies hat den Grund, dass Docker auf allen Plattformen lauffähig ist und die Container vom Service-Entwickler erstellt werden können. Dadurch bleibt Plattform plattformunabhängig und es muss nur die Docker-Umgebung bereitgestellt werden. Zusätzlich bieten sich durch die vielen bereits vorhandenen Container ideale und einfache Startbedingungen an. Durch die Popularität von Docker wird viel Support geboten und die Entwicklung kann dadurch verkürzt werden. Mithilfe der Docker-Registry und Docker-Hub bietet sich direkt ein öffentlicher Speicherort für die Services an, die heruntergeladen und verwendet werden können. Bei Bedarf besteht trotzdem noch die Möglichkeit

private Docker-Registries bereitzustellen. Daher werden unter *Image* mit den Attributen Name, Registry und Tag alle benötigten Informationen bereitgestellt.

Properties Hier werden die Zusatzinformationen eines Services definiert. Dies ist eine Liste, die wie bei dem Service mit einem Bezeichner eingeleitet werden. Eine Eigenschaft muss dabei einen eindeutigen Bezeichner in dem Service besitzen. Dies ist vor allem im Hinblick auf die Nutzbarkeit in der Oberfläche bedacht. Würden hier zwei Eigenschaften mit dem selben Bezeichner stehen, könnte der Nutzer die Information nur schwer oder überhaupt nicht verarbeiten. Zusätzlich kann eine Beschreibung hinzugefügt werden. Der Service-Entwickler kann mit *DefaultValue* einen Standardwert vorgeben, um dem Anwender das Konfigurieren zu vereinfachen. Ebenso hat der Service-Entwickler die Möglichkeit Eigenschaften als benötigt zu markieren oder nur einen lesbaren Wert zu definieren. Dies soll zum Beispiel bei einer Datenbankanbindung den Nutzer dazu zwingen die Datenbankadresse, etc. anzugeben. Ohne die entsprechende Konfiguration soll das Deployment nicht gestartet werden können. Ein Gegenstück zu diesem Szenario wäre, wenn der Service-Entwickler die Datenbankadresse festlegt (read-only) und diese dem Nutzer nur zur Information angezeigt wird. Als letztes wird noch ein Typ für die Eigenschaft ausgewählt. Hier wurden Standardtypen genommen und zusätzlich mit dem *port* erweitert. Dies ist vor allem für den Zugriff einer Website relevant.

Input/Output Die letzten Eigenschaften eines Services sind die Eingänge (Input) und Ausgänge (Output). Da ein Service mehrere Eingänge/Ausgänge haben, werden diese als Liste definiert. Um die Definition zu vereinfachen, wurde dafür ein eigener Typ festgelegt (siehe Listing 4.2).

Listing 4.2 Abstrakte Definition eines Ports für die Eingänge/Ausgänge eines Services.

```
Port:
  Id: string (eindeutig)
  Schema-Id: (optional, wenn Schema angegeben)
  Schema: (optional, wenn Schema-Id angegeben)
```

Ein sogenannter **Port** muss eine eindeutige ID besitzen und entweder ein komplettes Schema beschreiben oder auf die Schema-ID referenzieren. Ein Schema beschreibt dabei das genaue Datenformat, dass an diesem Port übertragen wird. Eine genaue Beschreibung findet sich in Abschnitt 4.2.4. Damit ist die Definition eines Services vollständig und flexibel aufgebaut.

4.2.4 Datenformat

Da Services nun Schnittstellen haben über die sie miteinander verknüpft und somit Daten austauschen können, ist es wichtig ein einheitliches Datenformat festzulegen. Die Service-Entwickler müssen informiert sein, welche Daten sie empfangen und verarbeiten können. Dies muss entsprechend im Vorfeld definiert werden. Durch die vorzeitige Definition und Festlegung haben es andere Service-Entwickler leichter sich an die entsprechenden Schnittstellen zu orientieren und ihre Services entsprechend der Schnittstelle zu programmieren. Auch soll eine einheitliche Datendefinition die Duplikate für ein und dieselben Daten reduzieren. Dies soll dadurch gelöst werden, dass die Formate über das Netzwerk abrufbar sind und im Idealfall mit einer geeigneten

Oberfläche angezeigt werden können. Mithilfe der Oberfläche soll die Verwaltung und Suche nach Schemas vereinfacht werden.

Nun stellt sich die Frage, welches Format für so eine Definition in Frage kommen könnte. In XML ist dies bereits durch XSDs bekannt und diese werden auch weitflächig verwendet. Selbst für den Maschinenkontext gibt es einen Pool an Datenquellen vom OPC-Verband[1]. Diese stellen bereits eine Vielzahl an unterschiedlichen Datentypen bereit. Die Nachteile, die durch die Nutzung von Schemadefinition in XML einhergehen, wurde bereits diskutiert. Um die Nachteile zu reduzieren, gibt es bereits Vorverarbeitungen, die die Daten komprimieren und die Datengröße reduzieren. Die Vorteile sind ganz klar durch die bereits vorhandene Datenbasis von OPC und der großen Verbreitung von XML gegeben. Alternativ würde sich im Webumfeld die Verwendung von JSON empfehlen, da die meisten Webanwendung mittels JSON kommunizieren. Hier bietet Apache mit Apache Avro[2] bereits eine komplette Umgebung mit Schemadefinition und Datenserialisierung an. Das Schema wird dabei ähnlich zu XSD mit Namespaces definiert und kann dadurch referenziert werden.

Listing 4.3 Beispiel eines Schemas für ein Auto mit Feldern für Marke, PS und der Farbe.

```
{
      "namespace": "example.cloudistry",
      "type": "record",
      "name": "Auto",
      "fields": [
      {"name": "marke", "type": "string"},
      {"name": "pferdestaerken", "type": ["int", "null"]},
      {"name": "farbe", "type": ["string", "null"]}
      ]
}
```

In Listing 4.3 ist ein Beispiel Schema im Avro-Format zu sehen. Alle Einträge werden JSON typisch mittels Schlüssel-Wert-Paaren definiert. Durch die Definition der Schemas kann eine Validierung in der Implementierung durchgeführt werden, da bekannt ist, welche Form die Daten haben. Durch den Wunsch einer Webanwendung und zusätzlich der Präferenz gegen die Verwendung von XML, da dies im Bereich des Maschinenbaus nicht so gern gesehen wird, ist die Entscheidung zu der Schemadefinition auf JSON Basis mit Apache Avro gefallen.

Schema-Evolution

Im Laufe der Lebenszeit eines Services kann davon ausgegangen werden, dass mehrere Updates für den Service zur Verfügung gestellt werden. Mit Updates können dabei neue Funktionalitäten, Erweiterungen oder Beschränkungen zur vorherigen Version des Services hinzukommen. Damit verbunden kann sich das Schema der Daten verändern, die vom Service empfangen oder gesendet werden. Diese Änderungen müssen erfasst und gesichert werden. Dieses Vorgehen wird als Schema-Evolution bezeichnet. Dabei wird nicht nur der Service mit verschiedenen Versionen versehen, sondern auch das Schema kann versioniert werden. Dadurch lässt sich der Verlauf eines Schemas nachverfolgen und zusätzlich können Rückwärtskompatibilitäten angeboten werden.

[1] https://opcfoundation.org/UA/schemas/1.02/Opc.Ua.Types.xsd
[2] https://avro.apache.org/

Ein Beispiel für die Schema-Evolution ist in Abbildung 4.5 zu sehen. Von *V1 -> V2* wird der Name des Autos zur Marke geändert. Anschließend wird noch das neue Feld Modell hinzugefügt. Verwender von alten Versionen ignorieren hierbei das neue Feld und arbeiten wie gewohnt weiter. Löschen von Feldern ist dabei kritischer als das Hinzufügen von Daten, da Services von den Daten abhängig sein könnten und ohne Alternative nicht mehr richtig arbeiten. Je nach verwendetem Schema Framework können auch Bedingungen für den Verlauf eines Schemas festgelegt werden. Beispielsweise könnte eine Schemaveränderung nur dann als valides Update zugelassen werden, wenn die Kompatibilität zur alten Version gewährleistet ist und die Typen bspw. konvertierbar sind. Durch solche Mechanismen wird eine automatische Verwaltung für Services realisiert, die nicht sehr gewartet werden müssen.

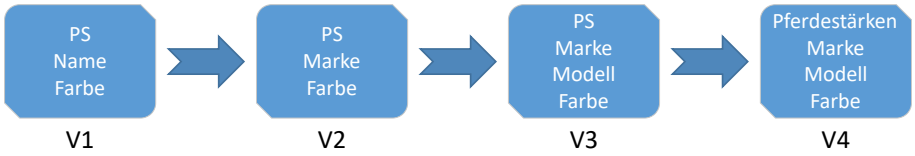

Abbildung 4.5: Beispiel für einen Evolutionsverlauf für das Schema für ein Auto.

Neben regelmäßigen Updates (Funktionen, Sicherheit) kann es vorkommen, dass bei der Veröffentlichung eines Schemas ein Fehler unterlaufen ist. Dieser Fehler bedarf einer zügigen und außerplanmäßigen Korrektur durch ein Update. Falls das Schema noch von keinem anderen Service verwendet wird, verursacht dies keine Probleme. Sobald das Schema jedoch schon verwendet wird, kann die alte Version nicht einfach verändert werden. Auch hier bietet es sich an durch die Evolution des Schemas eine Fehlerkorrektur einzubinden. Alte Versionen bleiben bestehen und Entwickler haben die Möglichkeit sich der Korrektur anzupassen.

4.3 Plattform-Architektur

In diesem Abschnitt werden grundlegende Konzepte zum Aufbau der Architektur der gesamten Plattform dargestellt. Hierfür müssen verschiedene Bereiche spezifiziert werden, die für die Verwaltung der Container, der Kommunikation und auch den Bezug zur Multi-Cloud, also verschiedenen Cloudanbietern, ermöglichen.

Containermanagement Da für die Services Docker als Containerformat verwendet wird, um leicht eine Plattformunabhängigkeit zu schaffen und das Deployment von Service so leicht wie möglich zu halten, ist es wichtig ein geeignetes Containermanagementsystem zu verwenden. In Abschnitt 3.2.1 wurden bereits verschiedene Plattformen und Frameworks vorgestellt mit denen sich Container verwalten lassen. Wie auch in der Bewertung ersichtlich ist, ist eine eher manuelle Verwaltung mithilfe von Docker Swarm zu aufwendig. Durch die Unterstützung der verschiedenen Bewertungskriterien ist Rancher hervorgestochen. Um sich einen großen Teil bei der Umsetzung eines Containermanagementsystems zu sparen, wird Rancher für diesen Bereich eingesetzt werden. Das Management der Container selbst soll dabei einerseits zentral von der Plattform aus gesteuert werden (inklusive der verfügbaren Knoten), kann aber auch zusätzlich durch die Rancher eigene Oberfläche noch granularer untersucht werden. Dies soll jedoch eher den Ausnahmefall darstellen und eher für die Entwickler oder Administratoren von Relevanz

sein. In Abbildung 4.6 ist die Architektur von Rancher dargestellt. Zum Verwalten der Container kann Rancher intern verschiedene bekannte Technologien nutzen, von Docker Swarm bis hin zu Kubernetes. Als Zwischenschicht zu den Hosts stellt Rancher eigene Dienste für den Speicher, Netzwerk, etc. bereit. Das heißt, dass hier nochmals eine Abstraktion stattfindet, um den Nutzer die Arbeit zu erleichtern.

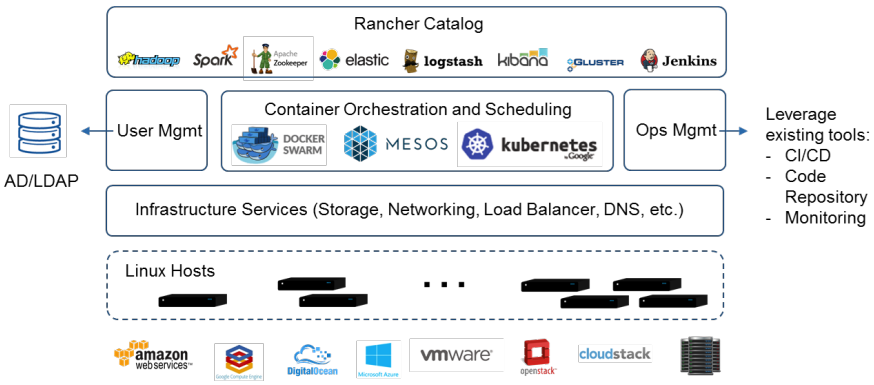

Abbildung 4.6: Darstellung der Rancher Architektur. [Ran17]

Neben dem Containermanagement bietet Rancher auch die Integration von verschiedenen Clouddanbietern wie Amazon AWS, Google Cloud, OpenStack oder auch lokale Ressourcen an (siehe Abbildung 4.6 unten). Mit der Nutzung von Rancher wird die Realisierung einer Multi-Cloud Umgebung ermöglicht, da hier gezielt je nach Bedarf ein entsprechender Anbieter genutzt werden kann.

Messagingsystem Nachdem nun die Services definiert und die Container verwaltet werden ist es wichtig zu entscheiden, wie die Services kommunizieren. In Abschnitt 2.3 wurde bereits erläutert, dass für eine lose gekoppelte Kommunikation Messaging als Kommunikationsmedium wichtig ist. Durch RPC oder andere Technologien würde die Kommunikation schnell ins Stocken kommen. Die einzelnen Aspekte der Vor- und Nachteile zwischen Messaging und anderen Möglichkeiten wird in einer anderen Arbeit genauer erläutert. Bei der Messaging Middleware gibt es dabei nur wenige, die sich etabliert haben und bekannt sind. Dazu zählen Apache Kafka, RabbitMQ und Apache Apollo (ActiveMQ). Durch die Performance, der Architektur und Verbreitung wird Apache Kafka als Messaging Middleware genutzt. Der Fokus lag dabei stets auf der effektiven Nutzung und Verarbeitung von Maschinendaten. Durch die Menge an produzierten Daten von Maschinen ist es deshalb von Bedeutung eine hochperformante[3] und stabile Messaging Basis zu verwenden. Wie in Tests gezeigt wurde[4], ist die Transferrate (ohne Node-Cluster zu nutzen) von Kafka weitaus höhere als der anderen Frameworks. In Abbildung 4.7 ist ein Beispielarchitektur für eine Kafka Konfiguration zu sehen. Im Normalfall werden dabei die ZooKeeper und Broker entsprechend skaliert, um Ausfallsicherheit und Zuverlässigkeit zu erreichen (mehr Information dazu in der Partnerarbeit). Die Producer schicken die Nachrichten an die Broker und diese verteilen die Nachrichten anschließend an die registrierten Consumer. Die Broker werden vom ZooKepper verwaltet und gesteuert.

[3] http://cloudurable.com/blog/what-is-kafka/index.html
[4] http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/

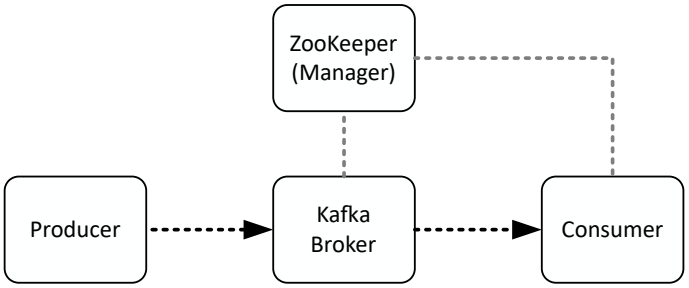

Abbildung 4.7: Beispielaufbau einer Apache Kafka Architektur.

Im Rahmen der Recherche nach einer möglichst optimalen Integration des Schemas bzw. einer effizienten Übertragung ist das Framework **Confluent**[5] aufgefallen. Confluent verbindet dabei Kafka als Messaging Basis und erweitert diese um viele weitere Funktionalitäten. Beispielsweise wurde hier idealerweise die Schema-Kompatibilität mit Apache Avro bereits eingesetzt und in das gesamte Ökosystem integriert. Dadurch können Producer und Consumer mit dem Schema arbeiten. Die Wahl des Messaging Frameworks ist durch die bereits vorhandene Integration von Apache Avro im gesamten Ökosystem für die Entscheidung von Avro bestärkt worden. Zusätzlich können Nachrichten blockiert werden, die dem Schema in einer Topic nicht entsprechen. Dadurch wird eine zusätzliche Sicherheitsschicht aufgebaut. Neben der Integration in die Messaging Umgebung existiert eine sogenannte Schema-Registry. In dieser werden alle verfügbaren Schemas gesammelt und versioniert. Die Veränderungen eines Schemas kann über die Versionierung zeitlich verfolgt und nachvollzogen werden. Confluent wurde zudem so aufgebaut, dass Erweiterungen mithilfe von Plugins möglich sind. Hier kann ideal an Anknüpfungsmöglichkeit über verschiedene Komponenten nachgedacht werden (siehe Abschnitt 4.4). Damit bietet Confluent ideale Anknüpfungspunkte mit den vorgestellten Anforderungen zu den Services in dieser Arbeit. In Abbildung 4.8 ist der grundlegende Aufbau von Confluent dargestellt. Dabei werden verschiedene Möglichkeiten angeboten, wie mit Confluent gearbeitet werden kann. Wenn benötigt, können Oberflächen angebunden, Daten überwacht oder Analysen durchgeführt werden. Confluent bietet mit Kafka Connect eine Möglichkeit Datenquellen anzubinden. Diese werden als Sink oder Source bezeichnet, je nachdem ob Daten konsumiert oder veröffentlicht werden.

4.3.1 Aufbau

Nach der Auswahl von Rancher für das Containermanagement und Confluent für das Messagingsystem mit Schemaintegration kann nachfolgend ein konkreter für die Plattform dargestellt werden.

Wie in Abbildung 4.9 dargestellt ist, steuert die Plattform die Erzeugung und das Starten der Container über Rancher und konfiguriert gleichzeitig die Kommunikationskanäle von Kafka. Dabei werden entsprechende Topics angelegt und gleichzeitig mit den Services verbunden. Dies geschieht über die REST-Schnittstelle von Confluent. Die gestarteten Container in Rancher werden dabei mit den entsprechenden Topics konfiguriert und sind anschließend dazu fähig untereinander zu kommunizieren. In dem Beispiel ist eine Maschine mithilfe eines Services angebunden. Dieser

[5] https://www.confluent.io/

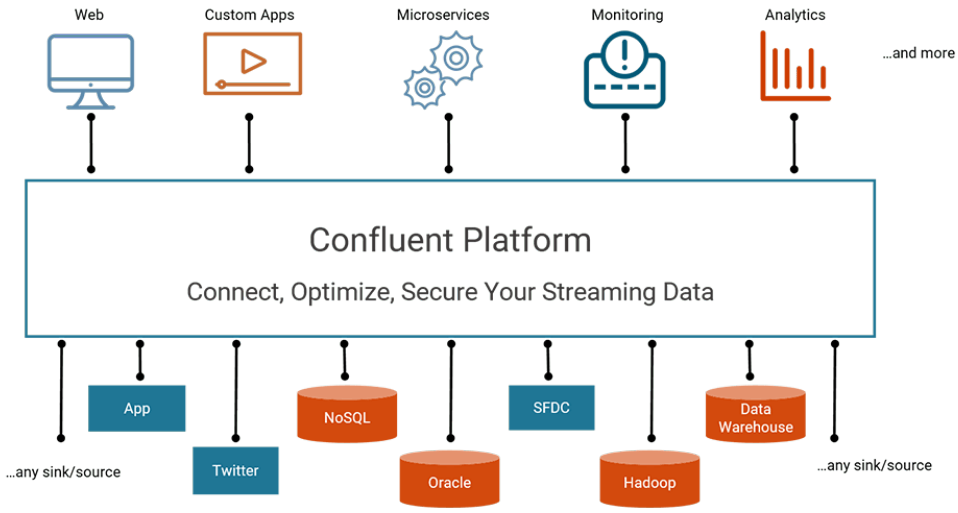

Abbildung 4.8: Darstellung der Confluent Architektur. [Con18]

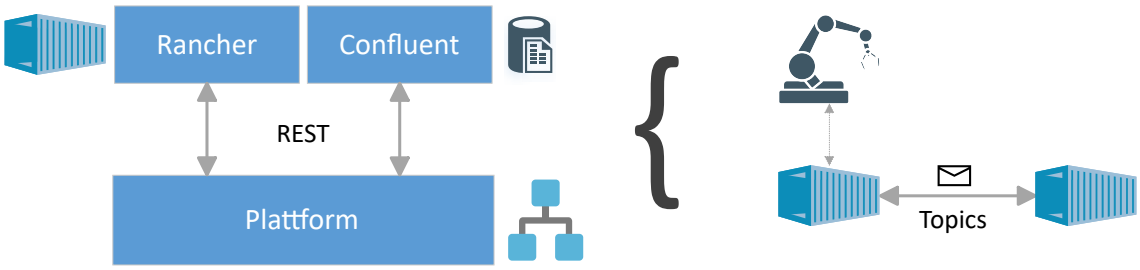

Abbildung 4.9: Der grundlegende Aufbau der Plattform arbeitet eng zusammen mit dem Rancher und Confluent.

Service startet einen OPC UA-Client und verbindet sich anschließend mit dem OPC UA-Server der Maschine. Nach dem Starten der Container dürfen diese nicht manipuliert werden, um eine sichere Laufzeitumgebung zu erzeugen. Durch das Konzept der Docker-Container wird dies bereits gewährleistet. Um eine Neukonfiguration der Docker-Container zu ermöglichen, muss ein Mechanismus konzipiert werden. Dies wird in Abschnitt 4.5 diskutiert. Im nachfolgenden Abschnitt wird der Aufbau des Backends erläutert. Die Frontend-Konzeption ist nicht Teil dieser Arbeit.

Backend-Aufbau

Das Backend übernimmt dabei die Arbeit der Verknüpfung, Steuerung und zur Überwachung der gesamten Services und deren Kommunikation. Die grobe Architektur wird dabei in Abbildung 4.10 definiert. Der Aufbau des Backends ist möglichst modular und erweiterbar aufgebaut. Dadurch sollen mögliche Erweiterungen, Verbesserung oder Modifikationen von Nutzern einfach und ohne großen Aufwand möglich sein.

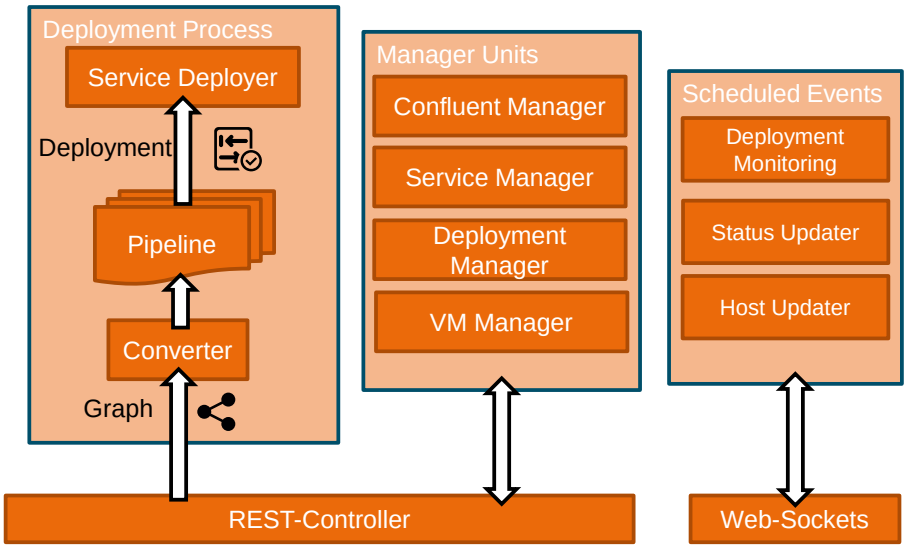

Abbildung 4.10: Die wichtigsten Komponenten für das Backend der Plattform. Die Hauptkomponenten umfassen dabei die Management, Deployment und Statusnachrichten.

Diese besteht aus drei Hauptkomponenten. Dies sind der Deploymentprozess, die Verwaltungseinheiten und die Events. Die aktive Kommunikation zum Frontend findet über eine REST-Schnittstelle statt. Hier werden alle Aktionen des Nutzers ausgewertet. Eine passive Kommunikation findet dabei über Web-Sockets statt. Passiv steht für zyklische Informationen, die dem Client ohne Anfrage bereitgestellt werden. Dies soll vor allem für Punkte wie Status-Updates und Überwachungen genutzt werden. Beispielsweise sendet der Server in zeitlichen Intervallen Informationen zum Zustand der Clouds an den Client.

Die Managementkomponente gliedert sich nach den benötigten Teilbereichen. Das ist zum Beispiel ein Confluent-Manager für die Steuerung und Nutzung von der Confluent Plattform. Weiterhin kommen Manager wie für das Deployment, den Services oder der virtuellen Maschinen hinzu.

Den wichtigsten Teil nimmt dabei die Komponente Deploymentprozess ein. Über das Frontend wird ein Prozess erzeugt und anschließend an das Backend übertragen. Dies kann als Startpunkt und Trigger für den Deploymentprozess gesehen werden. Der übertragene Graph wird dabei in eine interne Repräsentation überführt und persistiert. Anschließend wird es über eine Pipeline mit verschiedenen Modulen verarbeitet, die für das Deployment relevant sind. Beispielsweise könnte sich dabei ein Modul um die korrekte Zuweisung einer Cloud kümmern. Die Pipeline wird im nächsten Abschnitt genauer erläutert werden.

Konvertierungspipeline Um die Transformation von einem vom Nutzer erzeugten Graphen hin zu einem Deployment zu vereinfachen und zu strukturieren, wird eine sogenannte Pipeline eingeführt. Eine Pipeline besteht dabei aus mehreren Modulen, die einen Teilbereich der Transformation ausführen. Dadurch sollen die Module relativ klein gehalten und die Zuständigkeiten pro Modul weitestgehend reduziert werden. Zusätzlich ist durch die Modularisierung der Pipeline eine leichte Parallelisierung möglich. Module, die keine Abhängigkeit zu einem anderen Modul haben, können parallel ausgeführt werden. Bei einer Abhängigkeit ist dies nicht möglich und daher muss das Modul warten. Die Pipeline arbeitet dabei mit einem Kontext der für alle Module zugreifbar

ist. Dadurch hat jedes Modul die Möglichkeit Zwischenzustände zu sichern. Anschließend wird aus dem gesamten Kontext das Deployment generiert.

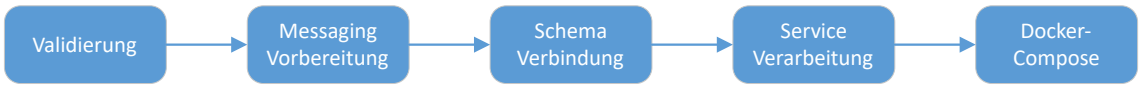

Abbildung 4.11: Konvertierungspipeline

In Abbildung 4.11 sind dabei die wesentlichen Module der Pipeline dargestellt, die während der Konzeptphase aufgekommen sind. Die Module sind von der der Service-Definition ausgehend extrahiert worden. Die wichtigsten Module umfassen dabei folgende Bereiche:

Validierung Dieser Teil der Pipeline muss zu Beginn ausgeführt werden. Das Modul prüft die Service-Definition, Konfigurationen und Parameter auf Validität und Konsistenz. Dies soll verhindern, dass in nachfolgenden Schritten Probleme entstehen, die nicht korrigierbar sind.

Messaging Vorbereitung In diesem Schritt werden alle benötigten Schritte für die Kommunikation und dem Messaging vorbereitet. Dazu zählt beispielsweise die Erzeugung der benötigten Topics in Kafka.

Schema Verbindung Die zuvor erstellten Topics werden nun bezüglich des Schemas der Eingänge und Ausgänge des Service verbunden und festgelegt. Nach diesem Schritt ist die Kommunikation dahingehend fertig, dass die Kommunikationspartner die übertragenen Daten verstehen und verarbeiten können.

Service Verarbeitung Hier werden alle weiteren Informationen der Service-Definition verarbeitet wie zum Beispiel die Properties des Services. Je nach Property kann nach Bedarf ein eigenes Modul definiert werden.

Docker-Compose Generierung Zum Schluss wird mit allen vorangegangenen Schritten eine vollständige und valide Docker-Compose Datei erzeugt. Die entstandene Docker-Compose kann im Anschluss direkt für das Deployment verwendet werden.

Nach der Konvertierung muss die fertige Docker-Compose nun deployed werden. Idealerweise bietet Rancher auch direkt eine Unterstützung für Docker-Compose Dateien, sodass das Deployment direkt möglich ist.

4.4 Interne/Externe Komponenten

Nachdem die Plattformen und die wichtigsten Teilkomponenten konzipiert wurden, müssen verschiedene Komponenten in die Plattform integriert werden. Dabei wird zwischen internen und externen Komponenten unterteilt. Interne Komponenten sind dabei solche, die selbstständig entwickelt und programmiert werden. Der fertige Service wird dann über einen Service-Store der Plattform zur Verfügung gestellt. Das heißt, dass die Integration des Services direkt über die Plattform stattfindet. Externe Komponenten hingegen umfassen Anwendungen oder andere Plattformen, die bereits in Betrieb sein können. Dazu kann beispielsweise eine externe Datenbank oder eine andere Messaging Plattform zählen, in der Daten verarbeitet werden. Die laufenden

Instanzen können nicht nochmals in der Plattform aufgesetzt werden, da dies extra Aufwand ist oder der Zugriff auf die externen Komponenten nur über verfügbare Schnittstellen möglich ist. Stattdessen soll eine flexible und leichte Integration möglich sein. Diese beiden Fälle werden in den nachfolgenden Abschnitte bearbeitet und eine Möglichkeiten zur Integration mit der Plattform konzipiert werden.

4.4.1 Interne Komponenten

Bei den internen Komponenten müssen Entwickler selber dafür sorgen, dass der Service entsprechende Daten empfangen und senden kann. Dafür sind die Inputs/Outputs in der Service-Definition vorgesehen. Dies ist notwendig, da der Entwickler den neuen Service an die Kommunikationsstrukturen der Plattform anbinden muss. Dem Entwickler die Arbeit mit der Anbindung an das Messaging Framework komplett selbst zu überlassen, gestaltet sich dabei als schwierig. Er müsste die entsprechenden Schnittstellen von Kafka verwenden und alle benötigten Adressen korrekt verarbeiten, um eine Verbindung zu dem gesicherten Kommunikationskanal aufbauen zu können. Eine Möglichkeit ist es über eine gute Dokumentation und Anleitung den Entwicklern die Arbeit zu erleichtern. Dies wird aber weiterhin einen hohen Aufwand bei der Entwicklung der Services bedeuten. Deshalb wurde dieser Ansatz nicht weiter verfolgt.

Um die vorgestellten Nachteile zu umgehen ist dabei der Ansatz einer eigenen Bibliothek zur einfachen Integration entstanden. Die Bibliothek soll dabei alle wesentlichen Elemente des Messaging Kontextes kapseln und dem Service-Entwickler nur die relevanten Informationen zur Verfügung stellen. Diese Bibliothek trägt den Namen Service-Core. Da der Service-Entwickler nur die Daten an definierten Eingängen und Ausgängen empfangen oder senden möchte, bietet es sich an genau hierfür Schnittstellen anzubieten.

Listing 4.4 Die beiden Kernfunktionen des Serivce-Cores zum Senden und Empfangen von Nachrichten.

```
function SendToOutput(outputId: string, data: SchemaFormattedData)
      FindSender(outputId).send(data)

function ConsumeFromInput(inputId: string, callback: Callback<SchemaFormattedData>)
      CreateConsumer(inputId).start(callback)
```

In Listing 4.4 sind die Schnittstellen als Pseudocode dargestellt. Der Aufwand für den Entwickler wird durch die kompakten Schnittstellen reduziert und dadurch eine Integration vereinfacht. Ein Problem an der Schnittstelle ist, dass der Service-Entwickler dafür sorgen muss, dass die zu übertragenden Daten im korrekten Schema vorliegen. Dafür muss der Service-Entwickler auf die verfügbaren Bibliotheken von Confluent oder Avro zurückgreifen, um seine Daten in das entsprechende Schemaformat umzuwandeln. Diese Bibliotheken könnten auch direkt im Service-Core angeboten werden, damit diese nicht extra vom Service-Entwickler zusammengesucht werden müssen. Ein weiterer Nachteil des Service-Cores stellt die Technologie dar. Da die Plattform programmiersprachenunabhängig ist, muss der Service-Core für die zu unterstützenden Sprachen umgesetzt werden. Neben der zusätzlichen Umsetzung ist auch eine Einschränkung seitens Kafka bzw. Confluent vorhanden. Die Technologien sind nicht in allen verfügbaren Sprachen umgesetzt und nur für einen geringen Anteil existiert eine entsprechende Unterstützung.

4.4.2 Externe Komponenten

Unter diese Kategorie fallen Anwendungen, die bereits aktiv sind oder nicht von außerhalb angepasst werden können. Dazu zählen beispielsweise laufende Datenbanken, existierende Messaging Plattformen oder auch zum Beispiel einfach REST-Services, die genutzt werden sollen. Dies bedeutet, dass eine Möglichkeit konzipiert wird, mit dem diese Datenquellen an die Plattformen angebunden werden ohne entsprechende Änderungen an den laufenden Komponenten vornehmen zu müssen. Zu dieser Kategorie zählen auch Maschinen. Diese werden im Anschluss separat behandelt.

Um externe Komponenten anzubinden, könnte für jede benötigte Art von Anbindung, bspw. JDBC-Datenbank, SQL-Datenbank, etc., ein eigener Service geschrieben wird. Dabei wird jeder spezielle Service entsprechend konfiguriert, um die Daten an die Eingänge/Ausgänge zu leiten. Jedoch ist dies eine sehr aufwendige Lösung, da für viele verschiedene Fälle eine eigene Lösung bereitgestellt werden müsste. Für kurzfristige Anbindung, stellt dieser Lösungsansatz einen brauchbaren Weg dar.

Eine andere Möglichkeit bietet Confluent mit *Kafka Connect* an. Dies ist ein Bibliothek, um externe System direkt an das Messaging einbinden zu können.

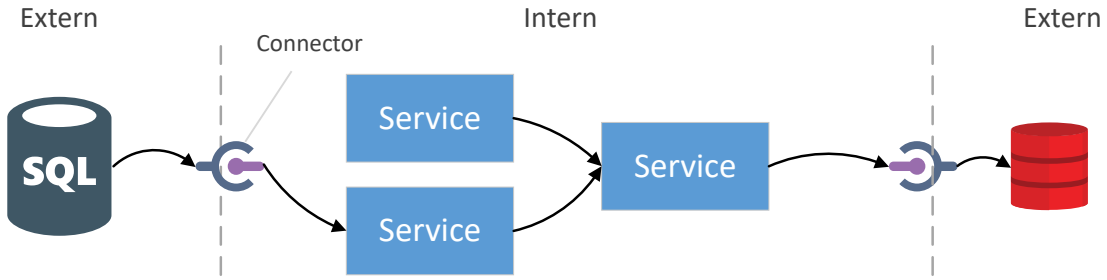

Abbildung 4.12: Mögliche Integration von Kafka Connect, um externe Komponenten ohne hohen Aufwand anbinden zu können.

Dabei wird zwischen Sink- und Source-Connectors unterschieden. In Abbildung 4.12 ist ein mögliches Szenario dargestellt. Links handelt es sich dabei um einen Source-Connector, der die Daten aus einer SQL-Datenbank importiert. Dort wird mit den Daten gearbeitet und anschließend über den Sink-Connector (rechts) in eine Oracle Datenbank übertragen. Die Connectoren entsprichen dabei in etwa dem Ausgang und Eingang eines Services. Durch ein Plugin System können zudem neue Connectors hinzugefügt werden, falls die vorhandenen nicht ausreichen sollten. Zu den bereits verfügbaren Konnektoren zählen zum Beispiele einfache Datei-Imports, Datenbanken, Amazon S3, FTP und viele mehr. Es sind sogar ausgefallene Konnektoren wie zum Beispiel einem Github Feed Connector vorhandenen. Hier würde es sich anbieten den Teil des Frameworks von Confluent mit in die Plattform zu integrieren, um die bereits vorhandenen Elemente für externe Komponenten nutzen zu können. Da dies jedoch losgelöst von der Plattform ausgeführt wird, muss ein entsprechender Service für die Konfiguration eines Sink- oder Source-Connectors hinzugefügt werden. Ergänzend dazu können eigene Integration entweder mittels eigenem Service oder einem neuen Plugin für Kafka Connect realisiert werden.

Schemadefinition für externe Komponenten Bei den internen Komponenten ist die Definition des Schemas für die einzelnen Eingänge und Ausgänge trivial. Wenn die Integration von externen Komponenten betrachtet wird, muss definiert werden wie und wo das Schema festgehalten wird. Da nicht im Vorfeld bekannt ist welches Format die Daten haben, muss diese Konfiguration dynamisch möglich sein. Am Beispiel von einer Datenbankschnittstelle können dabei beliebige Datenformate entgegen kommen.

Eine Möglichkeit ist die Definition in der Oberfläche über dynamische Felder. Das heißt, dass der Nutzer im Vorfeld wissen muss, was für Daten von der genauen Schnittstelle abgefragt werden. Durch einen entsprechenden programmierten Konvertierungsservice können die Daten bspw. aus der Datenbank in das Schemaformat umgewandelt und anschließend dem Prozess zur Verfügung gestellt werden. Bei einem Fehlerfall muss hier entsprechend reagiert werden. Wenn der Nutzer ein falsches Format angibt, darf der Service die Daten nicht weiterreichen, da sonst der komplette Prozess fehlerhaft verläuft. Um diesen Fall entgegen zu kommen, könnte auch eine Testverbindung aufgebaut werden, um das korrekte Format validieren zu können.

Eine weitere Möglichkeit wäre es direkt bei der Schnittstelle anzufragen und zum Beispiel ein Datenexemplar anzufordern. Dadurch hat der Nutzer die Möglichkeit die Daten zu sehen und in ein gewünschtes Schema umwandeln zu können. Gemeinsam mit einer geeignet Oberfläche, die ein einfaches zuordnen ermöglicht, könnte sich der Nutzer so das Format zusammenstellen. Diese Möglichkeit funktioniert wahrscheinlich nicht bei allen Quellen, da möglicherweise ein entsprechender Service im Vorfeld gestartet werden muss, um Zugriff zu erhalten. Bei Datenbanken könnte das Schema abgefragt und dieses anschließend verarbeitet werden.

Maschinenanbindung

Maschinen laufen in ihrer geschlossen Umgebung und haben nur in seltenen Fällen eine direkte Anbindung an ein offenes Netzwerk. Oftmals wird stattdessen über separate Netzwerkstrukturen, die Kommunikation mit der Maschine ermöglicht. Dies ist auch wichtig, da die Maschinen in einer sicheren Umgebung arbeiten müssen und nicht von äußeren Einflüssen manipuliert oder angegriffen werden dürfen. Da Maschinen von verschiedenen Herstellern produziert werden, existieren viele unterschiedliche Lösungen wie Daten von den spezifischen Maschinen abgefragt werden. Um diesen Aspekt zu vereinfachen wird zur Kommunikation mit der Maschine die OPC UA Anbindung genutzt. Hierbei handelt es sich um ein Hardwaremodul, dass an Maschinen angeschlossen angeschlossen wird. Anschließend können die Daten in einem definierten Format über einen OPC UA-Server zur Verfügung abgefragt werden. Gleichzeitig ist es durch den OPC UA-Server möglich Daten an die Maschine zu senden. Dadurch könnte die Maschine konfiguriert oder gesteuert werden. Die Steuerung der Maschine wird in dieser Arbeit nicht erarbeitet.

Der Aufbau eines OPC UA-Servers und Clients ist in Abschnitt 2.6 erläutert. Um die OPC UA-Schnittstelle in dieser Plattform nutzen zu können, muss der Client integriert werden. Dafür bietet es sich an einen generischen Service zu erstellen, der sich dynamisch konfigurieren lässt. Wie in Abschnitt 4.4.2 beschrieben, kann hier über die OPC UA Schnittstelle direkt auf die verfügbaren Datenformate zugegriffen werden. Das zur Verfügung gestellte Format ist dabei hierarchisch aufgebaut, sodass es leicht in ein benötigtes Schema überführt werden kann. Daher bietet es sich an die verfügbaren Daten über eine flexible Zuordnung unterstützend über die Oberfläche anzubieten. Über diese kann der Nutzer schnell festlegen, wo welche Daten ausgegeben werden

sollen. Dies bedeutet, dass der generische Service erst in der Oberfläche definiert wird und die Ausgänge erst zu diesem Zeitpunkt festgelegt werden. Dadurch hat der Nutzer die Möglichkeit zu entscheiden, welche Daten er von der Maschine nutzen möchte.

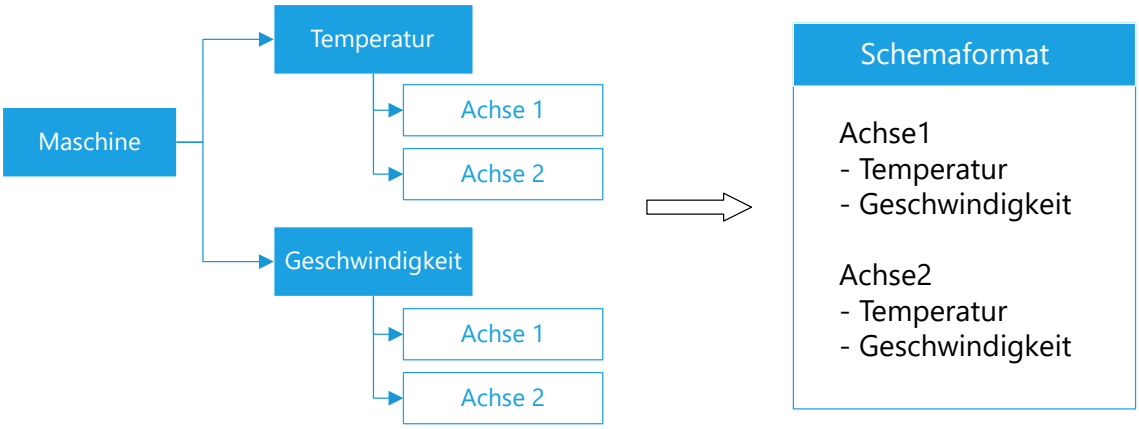

Abbildung 4.13: Flexible Zuordnung des Knoten-Kanten-Formats vom OPC UA-Server in das benötigte Schemaformat. Hierbei werden die Achsen pro Ausgang benötigt.

Ein Beispiel ist in Abbildung 4.13 dargestellt. Die Daten vom OPC UA-Server wurden beispielsweise in Temperatur und Geschwindigkeit gegliedert, bei denen die Subknoten Werte pro Achse enthalten. Der benötigte Service würde hierbei jedoch eine Gliederung pro Achse benötigen. Hier kann der Nutzer dann die jeweiligen Werte so zuweisen, dass das Schema jeweils Achsen enthält mit den entsprechenden Werten für Temperatur und Geschwindigkeit. Die weitere Ausarbeitung der genauen technischen Integration und der visuellen Darstellung des Maschinenservices wird nicht weiter betrachtet, da es nicht Teil dieser Arbeit ist.

4.5 Dynamik der Services

Neben dem Betrieb der Services spielt auch die dynamische Veränderung eine wichtige Rolle. Services sind wie Anwendungen, die nicht in in einem Zustand verweilen, sondern stetig verbessert, korrigiert und erweitert werden. Auf diese kontinuierliche Dynamik muss entsprechend reagiert werden. Der Begriff Dynamik im Zusammenhang von Services kann verschiedene Facetten einnehmen. Es kann für die Konfiguration und Veränderungen eines Services im laufenden Betrieb, einer Veränderung durch den Service-Entwickler oder auch die dynamische Konfiguration vor dem Deployment stehen. Diese Punkte werden in den nächsten Abschnitten genauer erläutert und gegebenenfalls mögliche Konzepte abgeleitet.

4.5.1 Aktualisierungen der Services

In der Laufzeit eines Services kommt es regelmäßig zu Änderungen, die durch Fehlerbehebungen, Updates oder Optimierungen seitens der Service-Entwickler auftreten. Daher muss damit gerechnet werden, dass sich die Definition und das Abbild des Services stets verändern kann. Es ist keine praktikable Lösung, wenn der Service-Entwickler für jede Änderungen einen neuen Service in die Plattform einpflegen müsste. Stattdessen kann mit Versionen und einem Aktualisierung-Logbuch

gearbeitet werden, in der die zeitlichen Veränderungen eines Services dokumentiert werden. Dadurch ist der Service immer unter dem gleichen Eintrag im Service-Store zu finden und gleichzeitig wird eine Aktualisierung für den Nutzer ermöglicht. Mithilfe einer geeigneten Kennzeichnungen können Services mit verfügbaren Aktualisierungen im erstellten Prozessgraphen hervorgehoben werden, damit der Nutzer je nach Bedarf reagieren kann.

Neben dem Verwalten der Versionen eines Services, ist es wichtig zu wissen wie ein Service genau aktualisiert wird. Als Orientierung können dabei die bestehenden Möglichkeiten bei Anwendungen und Containern genutzt werden. Der kritische Aspekt dabei ist, wie mit einer Ausfallzeit des Services umgegangen wird. Die Ausfallzeit steht hierbei für die Zeit, in der der Service nicht in der Lage ist die Daten in dem laufenden Prozess zu verarbeiten. In einem laufenden Prozess kann ein kompletter Stillstand eines Services (auch wenigen Sekunden) möglicherweise gravierend sein, wenn Werkstücke verarbeitet werden.

Anwendungsaktualisierung Diese Art der Aktualisierung entspricht dabei der klassischen Variante, wie es auf jedem Betriebssystem zu finden sind. Eine Anwendung wird dabei beendet oder im laufenden Betrieb aktualisiert, wo jedoch stets ein Neustart der Anwendung notwendig ist. Das Problem hierbe ist, dass ein relativ langer Zeitraum entsteht, bei dem die Anwendung im Stillstand ist. Dadurch ist diese Art der Aktualisierung nur bedingt für die Plattform geeignet.

Containeraktualisierung Bei den Containern sind die Aktualisierung anders gehandhabt, da diese bereits mit dem Anspruch auf das flexible Starten und Stoppen ausgelegt sind. Je nach verwendeten Technologien und Mechaniken kommt es dabei jedoch oft vor, dass eine aktualisierte Instanz des Containers gestartet wird und erst dann mit dem originalen Container getauscht wird, sobald dieser vollständig und fehlerfrei gestartet ist. In diesem Fall existiert ein Stillstand nur in Sekundenbruchteilen. Jedoch könnten hierbei Probleme auftreten, wenn die beiden Instanzen zur gleichen Zeit laufen. Dann könnte es vorkommen, dass die Services die Daten doppelt verarbeiten.

Containeraktualisierung mit dynamischer Konfiguration Eine Möglichkeit die Problematik mit gleichzeitig laufenden Container zu umgehen, besteht aus einer zeitverzögerten Konfiguration. Der neue Container wird beim Starten nicht vollständig konfiguriert, sondern erhält erst beim Wechsel der Container die restlichen Konfigurationen. Dadurch würden mögliche Fehlerzustände durch nicht auftreten.

4.5.2 Servicekonfiguration

Ausgehend aus den Gedanken zur Aktualisierungen von Services sind Herausforderungen bezüglich der Konfiguration des Services entstanden. Es hat sich die Frage gestellt wie ein Service in der Verbindung mit der Containertechnologie möglichst flexibel konfiguriert werden kann. Aus dem Containerprinzip geht hervor, dass gestartete Container nicht mehr verändert werden sollen und diese einen konstanten Zustand widerspiegeln.

Eine mögliche Art wie Services mit den nötigen Informationen (auch den Topics) konfiguriert werden, ist die Verwendung von Umgebungsvariablen. Diese können von außen initial gesetzt werden und stehen dann dem Container zur Verfügung. Die Eingabe der Umgebungsvariablen ist nur initial möglich. Nach dem Start des Containers lassen sich diese nur noch durch einen

Neustart des Containers neu setzen. Ein Vorteil ist jedoch, dass die Umgebungsvariablen direkt in einer Docker-Compose Datei eingetragen werden können. Dieses Vorgehen kann in der Transformationspipeline der Plattform umgesetzt und die Services entsprechend konfigurieren werden. Der Nachteil ist jedoch, dass bei etwaigen Änderungen der Konfiguration der Service immer in gewisser Weise „neu“ gestartet werden muss. In diesem Prinzip stellt sich die einfache Umsetzung und Verwendung dem etwas starren Konzept der Veränderung gegenüber.

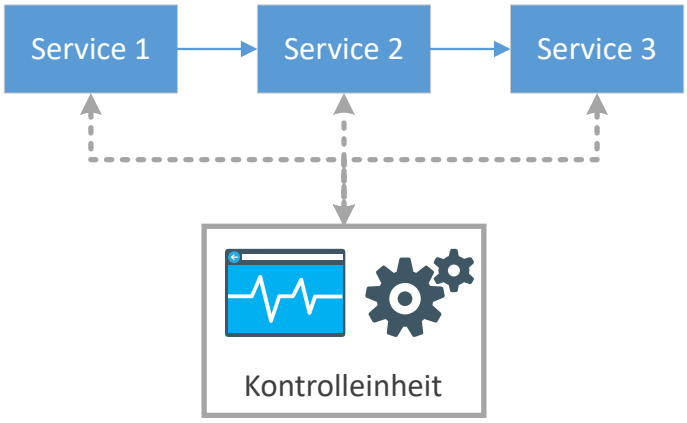

Abbildung 4.14: Konfiguration von Services über eine Kontrolleinheit (Control Bus Pattern).

Eine weitere Möglichkeit, die aus dem Kontext der Message Oriented Middleware entsprungen ist, ist das entsprechende Pattern *Control Bus*. In Abbildung 4.14 ist das Pattern entsprechend dargestellt. Über eigene Kommunikationskanäle für jeden einzelnen Service wird eine dynamische Konfiguration ermöglicht. Dadurch kann eine dynamische Veränderung der Topics für Eingänge und Ausgänge ermöglicht werden, wodurch das Problem der Stillstandzeiten gelöst werden kann. Als weiterer Vorteil ergeben sich zur Laufzeit beliebige Konfiguration und Anpassungen an den Service. Dieses Vorgehen könnte auch dahingehend erweitern werden, dass der Service-Entwickler über neue Felder in der Service-Definition Anpassungen ermöglicht. Dabei kann es sich um Konfigurationsparameter oder Funktionen handeln, über die sich der Service speziell anpassen lässt. Dadurch kann ein erneutes Starten der Services verhindert werden, da über die Kontrolleinheit eine neue Konfiguration des Services getriggert werden kann.

Neben der direkten Konfiguration, die durch den Nutzer ausgelöst wird, würde sich durch den Kontrollkanal ein dynamisches Konfigurieren anbieten. Je nach Nachricht kann sich der Zustand des Services verändern. Die Kontrolleinheit hat dadurch die Möglichkeit vollautomatisch Handlungen abzuleiten. Dadurch können Fehlerzustände entsprechend behandelt und der Kommunikationsfluss durch die Kontrolleinheit gesteuert werden.

4.5.3 Dynamischer Schemagenerator

Ausgehend von den Ideen mit der Zusammenstellung eines neuen Schemas aus vorhandenen Strukturen des OPC UA-Servers (siehe Abschnitt 4.4.2), lässt sich dieses Konzept auch auf beliebige Eingänge und Ausgänge übertragen. Um dem Nutzer die Möglichkeit zu geben nicht kompatible Eingänge und Ausgänge zu verbinden, kann dieser durch dynamische Zuordnung des Schemas von den Ausgängen die Kompatibilität für den Eingang erzeugen. Dies lässt sich an einem einfach Beispiel darstellen. Wenn ein Service als Ausgang Temperaturen in Celsius und Fahrenheit

liefert, jedoch der Eingang eines anderen Services nur Temperatur in Celsius annimmt. Durch entsprechendes dynamisches Zuordnen durch den Nutzern, könnte dieser die Kompatibilität herstellen. Dadurch würde die gesamte Plattform dynamischer und flexibler werden, da die starre Inkompatibilität durch dynamische Zuordnung aufgelöst werden kann.

4.6 Sicherheitsschichten

Neben der Konzeption der Rahmenbedingung und Strukturen von Service und Plattform nimmt die Sicherheit der einzelnen Bestandteile eine wichtige Rolle ein. Daher müssen entsprechende Schutzmechanismen vorhanden sein, um die Sicherheit zu gewährleisten. Im Artikel der SPS[6] wurden Sicherheitsaspekte des Forschungsprojekts PiCasso (siehe Abschnitt 3.2.4) untersucht. Der Fokus lag dabei auf den verschiedenen Sicherheitsmechanismen zwischen Betreiber und Integrator und den daraus resultierenden Einschränkungen der einzelnen Parteien. Im wesentlichen wurde zwischen einer End-to-End Verschlüsselung und einer ständigen Überwachung des Datenverkehrs verglichen. Bei der End-to-End Verschlüsselung war die Sicherheit höher, jedoch konnte der Datenverkehr nicht mehr untersucht werden, wohingegen die ständige Überwachung der transferierten Daten einen entsprechenden Einblick ermöglicht. Um die Komplexität der Sicherheitsaspekte zu reduzieren, wird wie in Abschnitt 3.1.3 beschrieben die Plattform in Teilelemente zerlegt und für jeden dieser Teile Schutzmechanismen diskutiert. Durch die unterschiedlichen Sicherungen der Ebenen entsteht ein zusätzlicher Schutz, da ein Angreifer nicht mit den gleichen Mitteln Zugang zu den unteren Schichten erhält.

4.6.1 Cloud-Ebene

Die Möglichkeiten der Sicherheit im Bereich der Cloud sind auf den verfügbaren Mechanismen der entsprechenden Cloud-Provider begrenzt. Das heißt, dass die Integration von eigenen Firewalls oder Sicherheitsfeatures in die Cloudumgebung nur bedingt möglich ist. Jedoch bieten die Provider bereits hohe Sicherheitsstandards an und halten diese auch stets auf aktuellem Stand. Es lassen sich verschieden granulare Regeln für die Sicherheit definieren. Daher reicht es in dieser Ebene aus die Features der Anbieter zu verwenden, um die Sicherheit zu gewährleisten.

4.6.2 Server-Ebene

Server können mit den verfügbaren Mitteln wie Firewalls, Antiviren-System und anderen Anwendungen geschützt werden. Zusätzlich kann der Zugriff durch Passwörter oder Zertifikate geschützt werden, damit Unbefugte nicht einfach mit dem Server kommunizieren können. Es besteht auch die Möglichkeit die Kommunikation zum Server so zu kapseln, dass nur erlaubte Zieladressen mit dem Server kommunizieren können. Das Ziel bei der Sicherung des Servers ist es das Eindringen von Fremden zu unterbinden und damit möglichen Schaden zu vermeiden. Um im Falle eines Schadens den alten Zustand wiederherstellen zu können, bieten sich ein automatisiertes Backup-System an. Zwischen den Backup-Strategien kann ebenso differenziert werden. Je

[6] http://www.sps-magazin.de/?inc=artikel/article_show&nr=122277

nach Möglichkeiten können dabei komplette Sicherungen durchgeführt oder nur inkrementelle Änderungen der Daten aufgezeichnet werden.

4.6.3 Container-Ebene

Die Sicherung der laufenden Services in ihren Containern ist ein wichtiger Bestandteil der Sicherheit. Dies hängt damit zusammen, dass verschiedene Anbieter ihre Services über den Service-Store anbieten können und somit eine Sicherheit für den Anbieter sowie dem Nutzer der Services gewährleistet sein muss. Docker bietet bereits standardmäßig ein Sicherheitskonzept durch die Containerisierung an. Durch die starke Isolation der Container vom Host und anderen Containern ist bereits eine Grundsicherheit gewährleistet. Zusätzlich werden Möglichkeiten geboten die Zugriffe der Applikationen in einem Container zu steuern[7]. Neben den vorhandenen Sicherheitsmechanismen in Docker müssen die Docker Images sicher gebaut werden und keine unnötigen sicherheitskritischen Parameter oder Umgebungsvariablen enthalten. Im Allgemeinen muss die mögliche Angriffsfläche eines Containers so klein wie möglich gehalten und die möglichen Punkte für Angriffe reduziert werden. Auf dem Markt existieren Lösungen um die Sicherheit eines Containers erhöhen zu können. Als Beispiel kann hier das Twistlock-Framework[8] genannt werden, die eine erweiterte Sicherung der Container anbieten. Dabei untersucht das Framework die bestehende Containerstruktur und zeigt mögliche sicherheitskritische Defizite in einem Dashboard an.

4.6.4 Nachrichtenkanal-Ebene

Bei der Sicherung des Nachrichtenkanals kann eine komplette Verschlüsselung der Nachrichten stattfinden oder über Überwachung die Kommunikation untersucht werden. Dabei kann auf vorhandene Techniken der Verschlüsselung von Datenpaketen zurückgegriffen werden. Standardmäßig werden die Topics und die Kommunikation bei Kafka bereits verschlüsselt. Neben den herkömmlichen Möglichkeiten zur Verschlüsselung der Kommunikation lassen sich auch weitere Punkte zur Sicherheit addieren. Dazu kann beispielsweise die Lebenszeit der Topics und Nachrichten gesteuert werden. Nachrichten können nach Verarbeitung entweder direkt gelöscht oder noch für bestimmte Zeit gesichert werden, um die Daten zu analysieren. Das gleiche Muster kann auf den gesamten Topickanal übertragen werden. Die Topic kann nur dann gelöscht werden, wenn der zugehörige Prozess gelöscht wird. Eine weitere Möglichkeit ist es nach zyklischen Intervallen zu schauen. Ist ein bestimmte Zeit vergangen, könnte eine Topic gelöscht werden. Damit kann die Sicherheit für kritische Daten erhöht werden. Dies müsste jedoch durch Rekonfiguration des Prozess einhergehen, da neue Topics angelegt und die Eingänge/Ausgänge umgelegt werden müssten.

4.6.5 Sicherheit mit Bezug zu externen Komponenten

Der letzte Part, der in die Sicherheitsschichten hinzugenommen werden kann, sind die externen Komponenten. Hier bietet sich jedoch eine extra Sicherung nur schwierig an, da keinerlei Ver-

[7] https://www.docker.com/sites/default/files/WP_IntrotoContainerSecurity_08.19.2016.pdf
[8] https://www.twistlock.com/

änderungen an den externen Komponenten durchgeführt werden kann. Das heißt, dass nur die Verbindungsstellen zwischen den externen Komponenten und der Plattform gesichert werden können. Durch Kafka Connect und den bereits beschriebenen Sicherheitsmechanismen durch Docker und des Nachrichtenkanals, wird bereits ein sicheres Umfeld erzeugt. Eine separate Sicherung ist nicht von Bedarf.

5 Umsetzung

Nach dem vorgestellten Konzept wird im diesem Teil der Arbeit auf die Umsetzung eingegangen. Dazu zählen die verwendeten Technologien, Programmiersprachen, Werkzeuge und Bibliotheken. In Zusammenarbeit mit einer anderen Arbeit wurden die spezifizierten Anforderungen eingearbeitet. Mit der Plattform ist es mögliche Services zu hinterlegen, in einem Prozess abzubilden und anschließend zu deployen. Nachdem die Service deployed wurden, kann der Status über eine Oberfläche überwacht werden. Es können eigene (interne) Services über eine Bibliothek bereitgestellt werden. Neben den internen Services können auch externe Service oder Maschinen an die Plattform angebunden werden. Die umgesetzte Plattform trägt dabei den Namen **Cloudistry**.

5.1 Gesamtarchitektur

Wie in der Konzeption schon dargestellt, wird die Plattform in ein Frontend und Backend Teil gegliedert und nach der Server-Client-Architektur aufgebaut. Zusammen mit den Frameworks Confluent und Rancher wurden Anforderungen für das Deployment und die Kommunikationsstrukturen vereinfacht. Die gesamte Architektur ist dabei wie folgt aufgebaut.

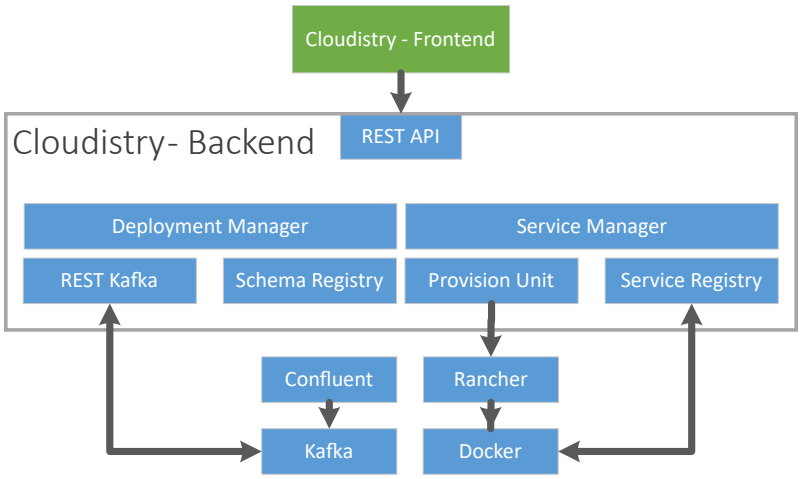

Abbildung 5.1: Darstellung der gesamten Architektur von Cloudistry. Dabei wird die Kommunikation der internen Komponenten mit den externen Frameworks dargestellt.

Die Kommunikation zwischen dem Backend und Frontend findet dabei über eine REST-Schnittstelle statt. Das Backend besteht aus zwei wesentlichen Blöcken: den Services und dem Deployment. Durch den Service-Manager werden alle benötigten Informationen rund um den Service verarbeitet. Das Deployment wird dabei von einem eigenen Manager verwaltet und gesteuert. Dabei ist eine enge Verbindung zu Confluent notwendig, da Topics erstellt und diese in

den Services hinterlegt werden. Bei der Umsetzung wurde vorausgesetzt, dass bereits jeweils eine Instanz von Rancher und Confluent vorhanden ist. Die Rancher Konfiguration wird dabei direkt beim Deployen von Cloudistry festgelegt. Die Konfiguration von Confluent hingegen kann im laufenden Betrieb der Plattform (am besten nach dem ersten Start) eingetragen werden.

Als Programmiersprache für das Backend wurde Kotlin[1] verwendet. Die Sprache hat im Vergleich zu Java viele Vorteile und Optimierungen erhalten. Zusätzlich bietet die vorhandene Kompatibilität mit Java und allen bestehenden Bibliotheken ein solides Grundgerüst. Zusammen mit dem Spring Boot[2]-Framework konnte eine schnelle und modulare Umsetzung geschaffen werden. Zudem bietet das Framework gute Integrationsmöglichkeiten von Datenbanken und einer komfortablen Definition von REST-Schnittstellen. Dadurch konnte der entsprechende Server bereits früh genutzt und getestet werden. Für das Verwalten von Bibliotheken und die Erzeugung der Artefakte wurde das Tool Gradle[3] verwendet.

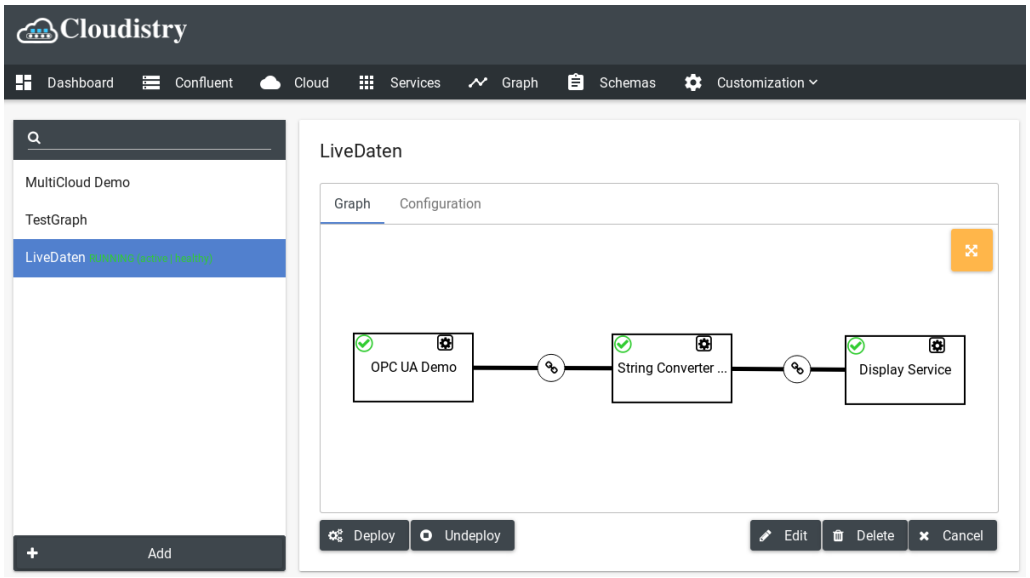

Abbildung 5.2: Screenshot der grafischen Oberfläche von Cloudistry.

Das Frontend von Cloudistry wurde mit Angular 4[4] und PrimeNG[5], als GUI Framework, umgesetzt. Wie in den Anforderungen definiert wurde, lag der Fokus bei der Erstellung der Oberfläche auf einer einfachen und selbsterklärenden Darstellung. Der Nutzer sollte ohne eine lange Einarbeitung dazu befähigt sein Prozesse zu erstellen und den Status überwachen zu können. In Abbildung 5.2 ist die Oberfläche mit einem geöffneten Prozess zu sehen. Der Prozess ist bereits deployed und der Status wird über die Icons (grüner Haken) auf den einzelnen Services dargestellt. Zusätzlich wird die konjugierte Zustandsanzeige in der seitlichen Ansicht für einen gewählten Prozess angezeigt. Die einzelnen Reiter werden im folgenden erläutert.

[1] https://kotlinlang.org/
[2] https://projects.spring.io/spring-boot/
[3] https://gradle.org/
[4] https://angular.io/
[5] https://www.primefaces.org/primeng/

Dashboard Hier werden die wichtigsten Informationen aller Reiter anzeigt. Dazu zählen die Zustände der verfügbaren Cloud-Ressourcen, der deployten Prozesse und weitere nützliche Informationen. Dies wird mithilfe von Visualisierungen wie Kreisdiagrammen auf einen Blick dargestellt.

Confluent Hier wird das Messaging-Framework konfiguriert. Es müssen Daten wie die Adressen für Broker, Schema-Registry und ZooKeeper eingetragen werden. Das bedeutet, dass eine laufende Instanz vorhanden sein muss und hier die Informationen für die Verbindung hinterlegt werden.

Cloud In diesem Reiter können Cloud-Instanzen hinzugefügt und entfernt werden. Diese Knoten werden für das Deployen der Prozesse genutzt und können bei Bedarf als Ziel gewählt werden.

Services In diesem Reiter können Services untersucht, hinzugefügt, editiert oder gelöscht werden. Dies kann als Schnittstelle für die Service-Entwickler genutzt werden, um neue Services an vorhandene Anbinden zu können. Alle verfügbaren Services in dieser Ansicht stehen dem Graphen zur Verfügung.

Graph Hier werden die Prozesse für die Verarbeitung erstellt, gestartet und überwacht. Daher ist das der Hauptarbeitsbereich von Cloudistry. Mithilfe von verschiedenen visuellen Unterstützungen wird dem Nutzer so viel Information wie nötig übermittelt.

Schemas Die Schema-Registry wird in diesem Reiter über eine weitere Oberfläche dargestellt. In dieser lassen sich verfügbare Schemas anzeigen, erweitern und neue anlegen.

Customization Diese Reiter ist nur für die Konfiguration der Oberfläche und überwiegend zu Testzwecken genutzt. Für die Produktivität ist dieser Reiter ohne Bedeutung.

5.2 Services

Services werden in der YAML-Notation (siehe Abschnitt 4.2) vom Service-Entwickler definiert. Um für die prototypische Umsetzung der Plattform Services leicht integrieren zu können, wurde der Service-Core, also die Bibliothek für den Service-Entwickler, in Kotlin geschrieben, um die Service in der Programmiersprache Java oder Kotlin zügig einbinden zu können.

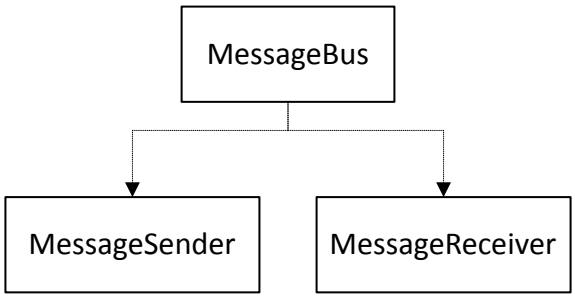

Abbildung 5.3: Aufbau des Service-Cores für die Verwendung in der Serviceentwicklung.

Die Grundkomponenten des Service-Cores sind in Abbildung 5.3 dargestellt. Der Entwickler nutzt den MessageSender und den MessageReceiver als Schnittstelle für die Kommunikation mit

Kafka. Wie die Namen es bereits ausdrücken entsprechen diese dem Senden und Empfangen von Daten.

Listing 5.1 Beispielimplementierung eines Services für die Konvertierung von einem String-Wert in einen Boolean-Wert.

```
private val booleanSender = MessageBus<String, Boolean>().createSender("boolean")

private val receiverBus = MessageBus<String, String>()
receiverBus.createReceiver("string", {
      it.toBoolean().let { booleanSender.send(null, it) }
})
receiverBus.start()
```

In Listing 5.1 ist eine Beispielimplementierung mit der Verwendung des Service-Cores zu sehen. Der MessageBus legt Sender und Empfänger an und kümmert sich anschließend um die valide Konfiguration. Zum Zeitpunkt der Erzeugung von Sender/Empfänger, wird der gewünschte Eingang/Ausgang ausgewählt.

MessageBus Stellt das Fundament der Bibliothek dar, in der die notwendigen Informationen für das Messaging Framework und aus der Service-Definition automatisch verarbeitet werden. Der MessageBus wird nicht direkt vom Service-Entwickler genutzt. Er greift stattdessen auf die nachfolgenden Elemente zu.

MessageSender Der Sender arbeitet dabei nur dann, wenn die entsprechende send(..) Methode aufgerufen wird. Zu diesem Zeitpunkt wird die gewünschte Nachricht serialisiert und an die Topic gesendet.

MessageReceiver Im Vergleich zu dem Sender arbeiten die Empfänger in eigenen Threads Das bedeutet, dass die Empfänger kontinuierlich laufen und auf das Empfangen von Nachrichten warten. Nach dem Empfangen wird die definierte Callback-Methode aufgerufen. In Listing 5.1 entspricht der Callback der Konvertierung des Werts in einen Boolean-Wert und dem anschließenden Senden.

Wie zu sehen ist, lässt sich ein Service mithilfe der verfügbaren Bibliothek schnell umsetzen. Die Schnittstelle, die für die Kommunikation mit den Topics notwendig ist, wurde sehr schmal und leichtverständlich gehalten. Dadurch können sich die Service-Entwickler auf die Entwicklung der Funktionalität ihres Services konzentrieren ohne sich viele Gedanken um Kommunikation machen zu müssen.

5.2.1 Interne/Externe Komponenten

Die internen Komponenten lassen sich mit dem vorgestellten Service-Core sehr leicht umsetzen. Dadurch sind im Rahmen der Arbeit einige Beispielservices für Testzwecke entstanden. Beispielsweise wurde ein Service mit einer kleinen Oberfläche mittels eines HTTP-Servers erzeugt oder ein Service der zufällige Integer und Float-Werte generiert. Durch die Integration ist das für interne Komponenten einfach.

Bei den externen Komponenten kann dies nur bedingt verwendet werden. Bestehende Komponenten können nicht verändert oder manipuliert werden. Stattdessen muss hier ein eigener

Service entwickelt werden, der die Daten mithilfe des Service-Cores aufnimmt und verarbeitet. Die vorgestellte Verwendung von Kafka Connect, wie in Abschnitt 4.4.2 beschrieben, konnte in der prototypischen Entwicklung nicht umgesetzt werden. Daher wurden die externen Komponenten wie zum Beispiel eine Datenbank mithilfe eines eigenen Services umgesetzt.

Maschinenkomponenten werden mithilfe eines generischen Services hinzugefügt. Dieser lässt sich zur Designzeit während der Erstellung des Prozesses in der Oberfläche konfigurieren, um die verfügbaren Daten des OPC UA-Servers verwenden zu können. Zu diesem Zeitpunkt der Implementierung wird davon ausgegangen, dass der Anwender die genauen Strukturen des Servers kennt. Im Prototypen wurden die Daten des OPC UA-Server dabei nicht direkt in ein komplettes Schema umgewandelt. Stattdessen werden die Daten des Services genommen und in einem nachgelagerten Konvertierungsservices in primäre Datentypen umgewandelt. Für den Prototyp und Tests war diese Umsetzung ausreichend.

5.3 Konvertierungspipeline

In Kapitel 4 wurde bereits vorgestellt, dass eine Pipeline für die Verarbeitung des Prozess aus dem Graphen zu einem konkreten Deployment mithilfe von kleinen Modulen leichter zu bewerkstelligen ist. Dabei wurde die Pipeline wie folgt aufgebaut. In den ersten Modulen werden die Eingabewerte auf Validität geprüft. Die Prüfungen umfassen die Validierung von benötigten Werten und anschließender Prüfung auf Inkompatibilität. Anschließend wird der Zustand für die Pipeline initialisiert mit dem in den nachfolgenden Modulen gearbeitet wird. Als nächstes werden die benötigten Topics für alle verbundenen Eingänge erzeugt und mit den zugehörigen Sendern verknüpft. Zum Schliss wird der Zustand persistiert und die fertige Docker-Compose Datei in weiteren Modulen generiert.

Listing 5.2 Beispielausgabe nach dem Durchlaufen der Konvertierungspipeline.

```
1  version: '2'
2  services:
3    ae959179-a19c-4f56-80b5-fd4d81762432:
4      image: cloudistry/ma-base-service-consumer-test1
5      hostname: ae959179-a19c-4f56-80b5-fd4d81762432
6      environment:
7        CLOUDISTRY_INPUT_1: ae959179-a19c-4f56-80b5-fd4d81762432
8        MASTER_BROKER_CONNECTION: broker1.Confluent:9092
9        MASTER_SCHEMA_URL: http://schema-registry.Confluent:8081
10   b59bbb96-3095-4eec-956b-1bceae573a5f:
11     image: cloudistry/ma-base-service-producer-test1
12     hostname: b59bbb96-3095-4eec-956b-1bceae573a5f
13     environment:
14       CLOUDISTRY_OUTPUT_1: ae959179-a19c-4f56-80b5-fd4d81762432
15       MASTER_BROKER_CONNECTION: broker1.Confluent:9092
16       MASTER_SCHEMA_URL: http://schema-registry.Confluent:8081
```

In Listing 5.2 ist eine generierte Ausgabe aus der Konvertierungspipeline zu sehen. Diese besteht aus zwei Service (Zeile 3 + 10) die eine entsprechende Konfiguration durch das *environment* in den Container injiziert bekommen. Eingänge, Ausgänge und Eigenschaften des Services werden dabei über die Präfixe *CLOUDISTRY_INPUT_, CLOUDISTRY_OUTPUT_, CLOUDISTRY_PROP_* in den Umgebungsvariablen abgefragt. Die Umgebungsvariablen *MASTER_BROKER_CONNECTION*

und *MASTER_SCHEMA_URL* dienen für die Kommunikation mit Kafka und werden intern im MessageBus verwendet.

5.4 Mögliche Erweiterungen

Nicht alle beschrieben Konzepte (siehe Kapitel 4) wurden in der prototypischen Umsetzung integriert. Durch die zeitliche Beschränkungen wurden nur die notwendigen Komponenten integriert, die für einen lauffähigen Prototypen beigetragen haben. Dabei können Erweiterungen wie eine dynamische Konfiguration der Service zur Laufzeit, ein Mandantensystem, eine vollständige Provisionierung, etc. hinzugefügt werden.

Die verfügbaren Sicherheitsmechanismen von Docker, Rancher und Confluent sind für den Prototypen ausreichend und wurden nicht mit weiteren Mechanismen erweitert. Docker bietet mit der Isolation der Container bereits eine sehr starkes Sicherheitsmerkmal. Rancher bietet durch entsprechende Loginmechanismen und SSL-Verschlüsselungen, ebenfalls wie Confluent, eine sichere Umgebung an. Bei extrem kritischen Daten und Prozessen muss für extra Schutzmechanismen gesorgt werden.

6 Validierung

In diesem Kapitel wird die umgesetzte Plattform validiert und anhand von Szenarien geprüft. Dies soll dazu dienen, die im Konzept vorgestellten Ideen auf ihre Plausibilität zu prüfen. Es wird auch ein Test mit Maschinen durchgeführt und bildet ein vereinfachtes Szenario eines Produktionsprozesses ab.

6.1 Manuelle Tests

Während der Umsetzung wurde eine Continuous Deployment-Pipeline aufgebaut mit integrierten automatischen Tests. In diesen Tests wurde dabei stets die Funktionalität der Plattform überwacht. Diese Tests prüfen den Deployment Prozess, die Korrektheit der Pipeline in verschiedenen Szenarien und damit einhergehend die Verarbeitung der Service Strukturen. Dadurch konnte bereits zu einem frühen Zeitpunkt eine solide Basis für die Funktionsfähigkeit der Plattform sichergestellt werden. Bei erfolgreichem Durchlauf der Tests wurden die Docker-Container gebaut und die laufende Instanz (auf einem Server) aktualisiert.

Zur Validierung der Plattform wurden auch Beispielservices entwickelt, um die Kommunikation, die Schnittstellen und das Deployment zu testen. Dabei sind folgende Service implementiert worden:

Generatorservice Dieser Service generiert in zyklischen Intervallen Integer/Float Werte und liefert diese an die entsprechende Ausgänge, insofern ein Verbraucher angebunden ist. Dieser Service entspricht dabei einem exemplarischen Producer.

Displayservice Dieser Service entspricht einem Consumer und soll das Empfangen der Daten visuell darstellen. Der Service startet dabei einen HTTP-Server an einem definierten Port und zeigt mithilfe von D3.js ein dynamisches Liniendiagramm der empfangenen Werte an. Dieser Service hat passende Eingänge für den Generatorservice.

Logservice Im Vergleich zum Displayservice zeigt der Logservice keine Werte an, sondern gibt diese vereinfacht in der Konsole aus. Dieser Service dient dabei eher für das Debuggen.

Eine Testkonstruktion ist dabei in Abbildung 6.1 dargestellt. Hier liefern zwei Generatorservices Daten an einen Displayservice. Der obere Service liefert Integer-Werte und der untere Service liefert Float-Werte. Über die Kante kann festgelegt werden, welche Eingänge und Ausgänge verbunden sind und kommunizieren können. Die Aktualisierung der Service wurde über die vorgestellte Container-Aktualisierung umgesetzt. Das heißt, dass parallel zum laufenden Container ein neuer gestartet wird. Bei erfolgreichem Start wird die laufende Instanz mit dem neuen Container getauscht. Die Aktualisierung der laufenden Services hat bei diesem Beispiel problemlos funktioniert. Der Displayservice zeigt die empfangenen Daten, wie in Abbildung 6.2 zu sehen ist, an.

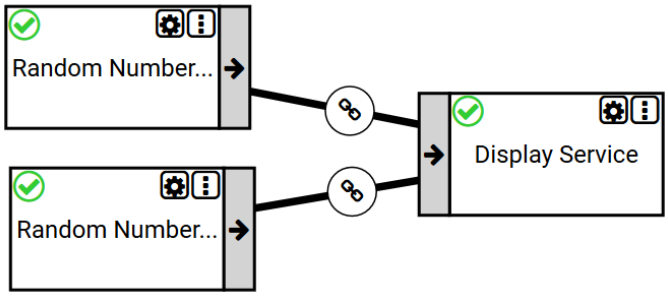

Abbildung 6.1: Testprozess mit zwei Producern und einem Consumer.

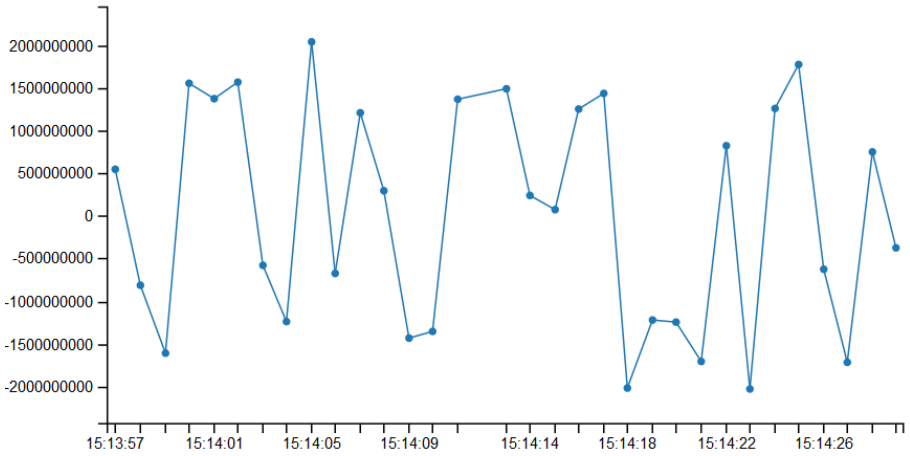

Abbildung 6.2: Darstellung der empfangen Integer-Werte von dem Producer.

Um die Multicloud Fähigkeit zu testen, kann pro Service oder für den gesamten Prozess festgelegt werden, auf welchem Knoten dieser bevorzugt deployed werden soll. Dies wurde dann manuell validiert, indem auf dem entsprechenden Knoten die laufenden Docker-Container inspiziert und so die Services geprüft wurden.

6.2 Test mit Maschinendaten

Um die Plattform an einem Aufbau mit einer Maschinenanbindungen zu testen, wurde eine Produktionskette mit Sensoren und Kameras aufgebaut. Das Ziel des Fließbandaufbaus ist die frühzeitige Erkennung von fehlerbehafteten Werkstücken. Die fehlerhaften Teile können anschließend so früh wie möglich aus der Produktionskette entnommen werden, um Zeit und Verarbeitungsaufwand zu sparen. Durch Verwertung der entstehenden Daten kann eine Optimierung der Sensoren durchgeführt werden. Damit ist die Reihenfolge der Sensoren gemeint in der die Werkstücke geprüft werden. Es könnte beispielsweise vorkommen, dass die häufigsten Fehler bei der Farbe auftreten. Dann würde es am Effizientesten sein, wenn dieser Sensor zu Beginn des Laufbandes das Werkstück überprüft. Die benötigten Service in dieser Produktionskette (siehe Abbildung 6.3) wurden dabei von einem potentiellen Plattformnutzer entwickelt. Genutzt wurde der Service-Core, um die Services schnell und einfach einbinden zu können. Einige Services

wurden dabei in der Programmiersprache R geschrieben, sodass der Service-Core mithilfe eines Wrappers eingebunden wurde, um den bestehenden Code nicht stark anpassen zu müssen.

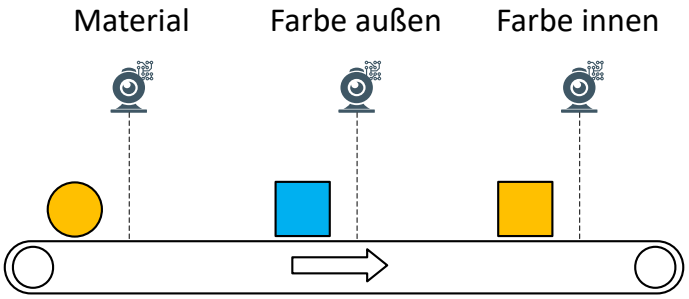

Abbildung 6.3: Aufbau der Beispielproduktionskette. Das Werkstück wird an drei Stellen untersucht.

In Abbildung 6.3 wird der Aufbau der Produktionskette dargestellt. Bauteile passieren über ein Laufband verschiedene Sensoren. Durch vorherige Festlegung der korrekten Grenzwerte werden die Bauteile überprüft. Weicht ein Teil bei einem Sensor ab, wird es aus der Kette genommen. Ein mechanischer Schieber befördert das Werkstück vom Fließband.

Abbildung 6.4: Aufnahme des Modells, das für Ausstellung genutzt wird. Das Material wird über das Gewicht bestimmt, die Farben mithilfe von Kameras.

Die aufgebaute Modellstation ist in Abbildung 6.4 zu sehen. In der Mitte befinden sich die Werkstücke, die zum Testen über das Band laufen und analysiert werden. Die komplette Technik befindet hinter der Plexiglasabdeckung unterhalb des Fließbandes. Die Daten werden dabei in einer Datenbank gesammelt und anschließend analysiert. Durch eine anschließende Selbstoptimierung verbessert sich dadurch die Detektion von fehlerhaften und fehlerfreien Werkstücken.

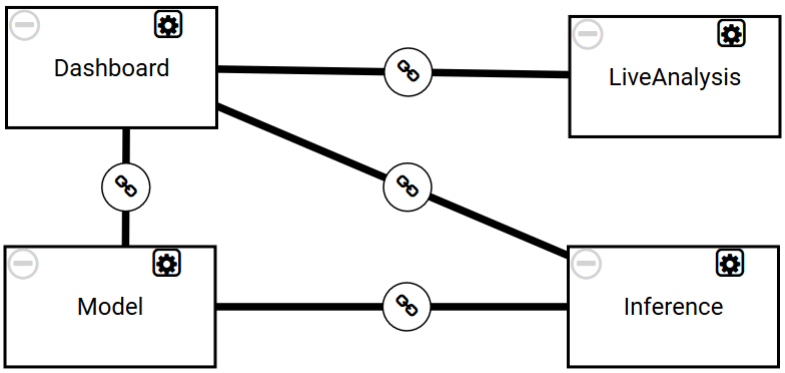

Abbildung 6.5: Der Analyseaufbau in Cloudistry für die Analyse der Daten.

Die gesammelten Daten und der aktuelle Zustand des Prozesses werden in einem Dashboard dargestellt. In Abbildung 6.5 ist der Aufbau für die Analyse zu sehen. Bei mechanischen Sensoren wurden die Daten mithilfe eines OPC UA-Servers bereitgestellt. Dadurch konnte der vorhandene Maschinenservice in diesem Szenario getestet werden. Die Daten wurden hierbei in Intervallen von 0,1 Sekunden abgefragt und verwertet. Die Geschwindigkeit des Fließbandes ist im Hinblick auf Präsentationen langsam eingestellt worden. Daher ist die Intervallrate mehr als ausreichend.

Die Integration des Service-Cores und Nutzung der entsprechenden Schnittstellen ist dem Service-Entwickler einfach gefallen. Durch die vorhandene Dokumentation der einzelnen Komponenten war die Nutzung klar und verständlich. Das Abstraktionslevel von den dahinter liegenden Kommunikationsstrukturen wurde als positiv empfunden. Es war nur bei den Services, die in R programmiert wurden, Unterstützung nötig.

7 Zusammenfassung

Die Wandlung der Firmen durch Industrie 4.0 bringt ein großes Potential für die Fertigung und Optimierung der bestehenden Prozesse. Mithilfe der Digitalisierung können die verschiedenen Bestandteile der Betriebe überwacht, verbunden und analysiert werden. Jedoch wird dies durch die verschiedenen Hersteller der Maschinen und Komponenten erschwert, da es an einer einheitlichen Schnittstelle mangelt. Aufgrund der fehlenden Kompatibilität wird eine zentrale Stelle benötigt, an der die Maschinen angebunden und miteinander verknüpft werden können. Durch anschließende Überwachung und Analyse der Prozesse kann eine Optimierung ermöglicht werden. Dies kann die Verteilung der Arbeitsschritte umfassen, frühzeitig Defekte der Maschinen erkennen oder eine Kontrolle der Werkstücke ermöglichen. Das Ziel ist die Minimierung der Produktionskosten, den Gesamtprozess zu verbessern und damit ein günstigeres Produkt erzeugen zu können.

In diesem Bereich herrscht momentan ein Defizit. Wie in Abschnitt 3.3 vorgestellt, wird von den Herstellern für Ansätze und Lösungsideen für die Problematik geworben, jedoch ist derzeit keine konkrete Lösung verfügbar. Weiterhin ist eine Validierung der verfügbaren Ansätze nicht möglich, da diese entweder nur auf Kontakt angeboten werden oder noch nicht verfügbar sind. Die verfügbaren Angebote bieten auch keinerlei Möglichkeit den Code einzusehen und so möglicherweise Erweiterungen oder Verbesserungen vorzuschlagen. Neben der Integration von Maschinen und der Verbindung in einen Prozess bieten die Plattformen kaum Unterstützung für den Einsatz in einer MultiCloud Umgebung. Die Deployments beschränken sich dabei auf firmeninterne Rechenkapazitäten oder werden in einer eigenen Cloud durch die Auswahl des Plattformbetreiber gehostet. Eine freie Auswahl und Konfiguration durch den Anwender ist somit nicht gegeben.

Gemeinsam mit einer Partnerarbeit wurde eine Plattform konzipiert und umgesetzt, mit der eine einfache Erstellung von Prozessen inklusive der Einbindung von Maschinen möglich ist. Einzelne Services oder der gesamte Prozesses lassen sich dabei gezielt auf einzelne Instanzen deployen. Diese Instanzen können dabei von Cloudanbietern wie Amazon AWS oder Google Cloud stammen oder eigene Rechenkapazitäten darstellen. In dieser Arbeit wurde die Definition von Services konzipiert, eine Bibliothek für die einfache Implementierung von den benötigten Schnittstellen beschrieben und die Kommunikationsstrukturen zwischen den Services definiert. Mit dieser kann ein Service-Entwickler in kurzer Zeit neue Services für die Plattform bereitstellen. Der Fokus lag dabei auf der Integration von internen und externen Komponenten. Durch den Service-Core soll die Integration vereinfacht werden. Für die externen Komponenten wurde eine Möglichkeit vorgestellt, mit der sich durch ein Plugin-System verschiedene Datenquellen integrieren lassen.

Um die umgesetzten Konzepte zu überprüfen, wurden verschiedene Tests durchgeführt. Mithilfe von manuellen und automatisierten Tests wurde die grundlegende Funktionalität der Plattform sichergestellt und konsistent gehalten. Durch die Implementierung von Testservices wurde die eigene Schnittstelle überprüft und die Kommunikation der Services validiert. Mit einem anschließenden realen Test mit Maschinenservices und einem externen Service-Entwickler konnte die

Gesamtheit der Plattform geprüft werden. Die Integration der Services von dem externen Entwickler wurde in kurzer Zeit umgesetzt und hat sich problemlos in die bestehende Plattform mithilfe der einfachen Service-Definition und Containerumgebung integrieren lassen. Durch das anschließende Testen mit einem physischen Modell mit Sensoren und Aktoren konnte die Funktionalität der Plattform bestätigt werden.

7.1 Ausblick

Da nicht alle vorgestellten Konzepte umgesetzt und nicht alle Teilbereiche der Plattform validiert werden konnten, sind einige Punkte für den Ausblick offen. Diese können in zukünftigen Arbeiten ausgearbeitet und optimiert werden.

Standard für Service-Definition In der Recherche wurden verfügbare Definitionen für Service-Definition vorgestellt und nach ihrer Verwendbarkeit geprüft. Aufgrund des komplexen Aufbaus und der Komplexität wurde eine eigene Service-Definition aufgestellt. Jedoch würde es sich anbieten eine Standarddefinition wie zum Beispiel TOSCA zu verwenden, um den Standard zu unterstützten und die Flexibilität der Plattform zu erhöhen. Dabei könnte die Umwandlung schrittweise erfolgen und nicht die komplette TOSCA-Engine von Beginn an unterstützt werden, sondern nur die benötigten Elemente. Dadurch kann auf die Standarddefinition umgestiegen werden, sofern dies benötigt wird.

Umsetzung fehlender Konzepte Wie bereits erwähnt wurde, sind nicht alle vorgestellten Konzepte umgesetzt worden. Dazu zählen erweiterte Sicherheitsmechanismen in den verschiedenen Schichten der Plattform, Aktualisierungsfähigkeit und der dynamischen Konfiguration der Services. Um Teile der Konzepte validieren zu können, bedarf es an vielen Ressourcen, da die Sicherheit von Komponenten nur mit gezielten Angriffen gut zu testen sind. Die dynamische Konfiguration bietet eine vielversprechende Möglichkeit an, um den laufenden Prozess zur Laufzeit nach Bedarf anpassen zu können. Dabei können Betriebsparameter manuell oder automatisiert angepasst werden, um die Qualität zu erhöhen. Beispielsweise können Sensoren durch Schwellwertanpassungen sensibilisiert werden, um den Prozess besser steuern zu können.

Einsatz in echtem Firmenkontext Einer der wichtigsten Punkte ist der Test und Einsatz in einer realen Umgebung. Die vorgestellte Testplattform mit einigen Sensoren und Aktoren kann das reale Arbeitsumfeld nur zu einem kleinen Bruchteil widerspiegeln. Daher ist es wichtig, die Plattform mit realen Maschinen und in einer größeren Prozesskette zu prüfen. Die Geschwindigkeit und Datenmenge sind in diesem Szenario viel höher. Zudem kann dadurch die Kundenrelevanz, Nutzbarkeit und Einfachheit der Plattform bewertet werden. Ein weiterer Vorteil, der sich durch das reale Umfeld ergibt, ist die Anforderung Maschinen verschiedener Hersteller in Verbindung zu bringen. Es gilt zu prüfen, wie gut sich die Maschinen verschiedener Hersteller innerhalb der Plattform verbinden lassen und ob ein problemloser Betrieb möglich ist. Dies würde den Firmen einen echten Mehrwert liefern.

Übertragung auf andere Gebiete Neben dem Betrieb der Plattform im Maschinenkontext, lässt sich der Nutzen auf andere Bereiche übertragen. Als Beispiel kann *Connected Car* oder das *Internet of Things (IoT)* genannt werden. Durch die Kommunikation von vielen kleinen Komponenten innerhalb und außerhalb des Autos wird eine Möglichkeit benötigt, um diese zu steuern.

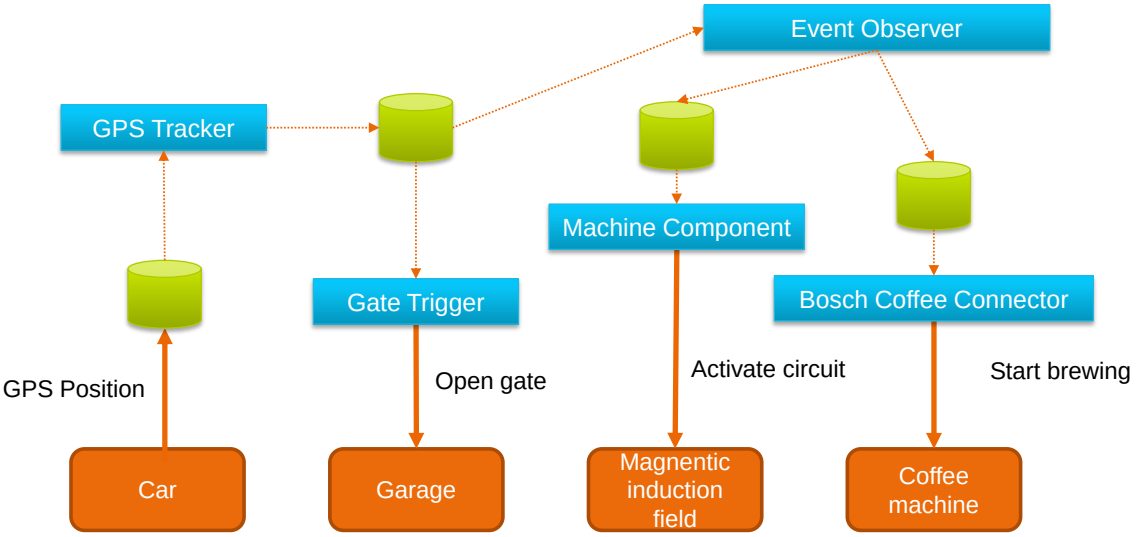

Abbildung 7.1: So könnte eine Beispiel Umsetzung für das Connected Car aussehen, bei der verschiedene Trigger durch die Position des Autos ausgelöst werden.

Ein kleines Beispielszenario ist in Abbildung 7.1 zu sehen. Dabei wird durch die Position des Autos ein Prozess in Gang gesetzt. Die Garage öffnet, das Ladefeld für das Auto wird aktiviert und anschließend wird dem Fahrer der Kaffee gekocht.

Literaturverzeichnis

[Ada17] Adamos. *Adamos Website*. 2017. URL: https://de.adamos.com/about-adamos (zitiert auf S. 36).

[AG18] M. AG. *Industrie 4.0*. 2018. URL: https://www.manz.com/de/industrie-4-0/ (zitiert auf S. 11).

[ALI17] ALIEN4Cloud. *ALIEN4Cloud Documentation 1.4.0*. Okt. 2017. URL: https://alien4cloud.github.io/#/documentation/1.4.0/index.html (zitiert auf S. 32).

[Arc18] I. of Architecture of Application Systems (IAAS). *OpenTOSCA Container - Architecture*. 2018. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/container_architecture.php (zitiert auf S. 28).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA - A Runtime for TOSCA-based Cloud Applications“. English. In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*. Bd. 8274. LNCS. Springer Berlin Heidelberg, Dez. 2013, S. 692–695. DOI: 10.1007/978-3-642-45005-1_62. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-45&engl=1 (zitiert auf S. 27).

[BBLS12] T. Binz, G. Breiter, F. Leyman, T. Spatzier. „Portable Cloud Services Using TOSCA“. In: *IEEE Internet Computing* 16.3 (Mai 2012), S. 80–85. ISSN: 1089-7801. DOI: 10.1109/MIC.2012.43 (zitiert auf S. 26).

[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: *SIGSOFT Softw. Eng. Notes* 39.1 (2014), S. 1–4. ISSN: 0163-5948. DOI: 10.1145/2557833.2557844 (zitiert auf S. 35).

[Bro15] A. e. a. Brogi. „Adaptive management of applications across multiple clouds: The SeaClouds Approach“. en. In: *CLEI Electronic Journal* 18 (Apr. 2015), S. 2–2. ISSN: 0717-5000. URL: http://www.scielo.edu.uy/scielo.php?script=sci_arttext&pid=S0717-50002015000100002&nrm=iso (zitiert auf S. 36).

[BTV14] T. Bauernhansl, M. Ten Hompel, B. Vogel-Heuser. *Industrie 4.0 in Produktion, Automatisierung und Logistik: Anwendung, Technologien und Migration*. Springer, 2014 (zitiert auf S. 9, 28, 29).

[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. 25).

[Clo17] Cloudify. *Cloudify Documentation 4.2.0*. 20. Nov. 2017. URL: http://docs.getcloudify.org/4.2.0 (zitiert auf S. 32).

[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. 25, 26).

[Con18] Confluent. *Confluent Architecture*. 2018. URL: https://www.confluent.io/ (zitiert auf S. 55).

[Cur04] E. Curry. „Message-oriented middleware“. In: *Middleware for communications* (2004), S. 1–28 (zitiert auf S. 14–16).

[Doc17] Docker. *Docker Docs v17.12*. 27. Dez. 2017. URL: https://docs.docker.com (zitiert auf S. 31).

[DVE+16] S. Daya, N. Van Duy, K. Eati, C. M. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins et al. *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach*. IBM Redbooks, 2016 (zitiert auf S. 32).

[Gmb18] A. GmbH. *Adamos Representation*. 2018. URL: https://de.adamos.com/about-adamos (zitiert auf S. 37).

[HRN06] J. Humble, C. Read, D. North. „The deployment production line“. In: *AGILE 2006 (AGILE'06)*. Juli 2006, 6 pp.-118. DOI: 10.1109/AGILE.2006.53 (zitiert auf S. 23).

[HS96] T. G. Handel, M. T. Sandford. „Hiding data in the OSI network model“. In: *International Workshop on Information Hiding*. Springer. 1996, S. 23–38 (zitiert auf S. 29).

[HWSB13] P. Holtewert, R. Wutzke, J. Seidelmann, T. Bauernhansl. „Virtual Fort Knox Federative, Secure and Cloud-based Platform for Manufacturing“. English. In: *Procedia CIRP* 7.Complete (2013), S. 527–532. DOI: 10.1016/j.procir.2013.06.027 (zitiert auf S. 33, 34).

[Inc18] D. Inc. *VM and Containers*. 2018. URL: http://rancher.com/playing-catch-docker-containers/ (zitiert auf S. 12).

[Ins15] Institut für Steuerungstechnik der Werkzeugmaschinen und Fertigungseinrichtungen. *MULTICLOUD-BASIERTE DIENSTLEISTUNGEN FÜR DIE PRODUKTION*. 1. Nov. 2015. URL: https://www.bmbf-multicloud.de/projekt/ (zitiert auf S. 9).

[Ins18] K. Insights. *IBM Bluemix - Architecture*. 2018. URL: https://www.kelros.com/ibm-cloud/ibm-bluemix/ (zitiert auf S. 33).

[Kno18] V. F. Knox. *Virtual Fort Knox - Architecture*. 2018. URL: https://www.virtualfortknox.de/media/images/GWK_Architektur.original.png (zitiert auf S. 33).

[LFWW16] F. Leymann, C. Fehling, S. Wagner, J. Wettinger. „Native Cloud Applications-Why Virtual Machines, Images and Containers Miss the Point!“ In: *WEBIST (1)*. 2016, S. 5 (zitiert auf S. 18).

[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. URL: https://doi.org/10.1007/978-3-658-18165-9_3 (zitiert auf S. 34).

[MG12] P. Mell, T. Grance. „The NIST Definition of Cloud Computing: Recommendations of the National Institute of Standards and Technology (2011)“. In: *NIST Spec Publ* 800145 (2012) (zitiert auf S. 18, 19).

[Nan15] N. Nannoni. „Message-oriented middleware for scalable data analytics architectures“. Magisterarb. 2015 (zitiert auf S. 16).

[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. 26, 27).

[Ran17] RancherLabs. *Rancher Documentation 1.6*. Dez. 2017. URL: https://rancher.com/docs/rancher/latest/en/ (zitiert auf S. 32, 53).

[Sma18] K. SmartOrchestra. *SmartOrchestra Representation*. 2018. URL: http://smartorchestra.de/ (zitiert auf S. 35).

[Tec18a] T. Technologies. *OPC / OPC UA Architecture*. 2018. URL: https://www.thorsis.com/en/industrial-automation/software/technologies/opc-opc-ua/ (zitiert auf S. 22).

[Tec18b] Techtag. *Axoom: „Wir gestalten die Zukunft der Produktion“*. 2018. URL: https://www.techtag.de/digitalisierung/digitale-pioniere/axoom-wir-gestalten-die-zukunft-der-produktion/ (zitiert auf S. 38).

[Ter18] B. Terkaly. *The Docker Ecosystem*. 2018. URL: https://blogs.msdn.microsoft.com/allthingscontainer/2016/09/15/windows-containers-getting-started-a-step-by-step-guide/ (zitiert auf S. 13).

[WHS11] R. Welke, R. Hirschheim, A. Schwarz. „Service-Oriented Architecture Maturity“. In: *Computer* 44.2 (Feb. 2011), S. 61–67. ISSN: 0018-9162. DOI: 10.1109/MC.2011.56 (zitiert auf S. 21).

[WSCL13] E. Westkämper, D. Spath, C. Constantinescu, J. Lentes, Hrsg. *Digitale Produktion*. Berlin: Springer Vieweg, 2013. ISBN: 978-3-642-20258-2. DOI: 10.1007/978-3-642-20259-9 (zitiert auf S. 11, 12).

[Zim16] M. Zimmermann. „Konzept und Implementierung einer Komponente zur Kommunikation TOSCA-basierter Anwendungen“. Magisterarb. Universität Stuttgart, 05 Fakultät Informatik, Elektrotechnik und Informationstechnik, 28. Apr. 2016. DOI: http://dx.doi.org/10.18419/opus-9324 (zitiert auf S. 26, 27).

Alle URLs wurden zuletzt am 19. 02. 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