Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Modellierung von Data Loops in Connected Car-Umgebungen

Katharina Gönner

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Bernhard Mitschang
Betreuer/in:	Dr. Pascal Hirmer, Dr. Uwe Breitenbücher
Beginn am:	03. Mai 2021
Beendet am:	03. November 2021

Kurzfassung

Moderne Autos enthalten immer mehr Elektronik. Durch diese Veränderung können heutzutage nicht nur interne Komponenten der Autos miteinander kommunizieren, sondern auch unterschiedliche Autos untereinander Daten austauschen. Fahrzeuge, welche miteinander kommunizieren, werden Connected Cars genannt. Diese Vernetzung soll es ermöglichen, dass Autos in der Zukunft autonom agieren und fahren. Die vielen Sensoren und Steuergeräte können in diesem Zusammenhang als Things in einem Internet of Things-Szenario betrachtet werden. Dies nennt sich in diesem Fall Internet of Vehicles. Um ein solches Konzept zu realisieren, wird eine Edge-, Fog- oder Cloud-Infrastruktur benötigt, mithilfe derer die gesammelten Daten gespeichert und verarbeitet werden sollen. Diese Struktur wird im Laufe der Arbeit modelliert und im Weiteren als Data Loop bezeichnet. Dabei spielt der Datenfluss eine große Rolle. Die Daten sollen zunächst gesammelt, dann gespeichert und weiterverarbeitet werden. Da die Menge an Daten nicht nur sehr groß ist, sondern diese auch heterogen sind, sollen sie in ihrer Rohform in einem Data Lake abgespeichert werden. Anschließend findet die Verarbeitung der Informationen statt. Für diese ist der sogenannte Digitale Zwilling von großer Bedeutung, welcher genau ein physisches Produkt, in diesem Fall ein Fahrzeug, in der digitalen Welt widerspiegelt. Dieser verändert sich zusammen mit dem physischen Gegenstück, durch die gesammelten Daten. Gleichzeitig sollen Veränderungen am Digitalen Zwilling auch Auswirkungen auf den physischen Zwilling haben. Dieser gesamte Ablauf wird als Data Loop bezeichnet. Im Laufe der Arbeit werden zunächst einige Anforderungen an das Konzept der Data Loop gestellt, woraufhin der Entwurf des Konzeptes folgt. Auf dessen Basis wird eine prototypische Implementierung durchgeführt und modelliert. Das daraus resultierende Modell wird anschließend dazu verwendet alle Komponenten der Data Loop automatisch zu installieren und auszuführen. Für die Modellierung wird dabei der OASIS-Standard TOSCA herangezogen. Abschließend wird die Implementierung anhand der zuvor definierten Anforderungen evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

BPMN Business Process Model and Notation. 27

CSAR Cloud Service Archive. 26

DTA Digital Twin Aggregate. 20

DTI Digital Twin Instance. 16

DTP Digital Twin Prototyp. 19

DZ Digitaler Zwilling. 19

GDPR General Data Protection Regulation. 71

IaaS Infrastructure as a Service. 23

IoT Internet of Things. 15

IoV Internet of Vehicles. 15

JSP JavaServer Pages. 57

OTA Over-the-Air. 70

SUMS Software Update Management. 70

TOSCA Topology and Orchestration Specification for Cloud Applications. 16

UI Nutzeroberfläche. 52

UNECE United Nations Economic Commision for Europe. 70

VM Virtuelle Maschine. 27

WAR Web Application Archive. 57

1 Einleitung

Heutzutage wird in Autos immer mehr Elektronik eingesetzt. Moderne Fahrzeuge haben mehr als 100 Steuergeräte und verwalten damit die nötigen Funktionen, wie das Bremsen und die Steuerung des Fahrzeugs. Zusätzlich bestehen die Software-Komponenten der Fahrzeuge aus mehreren Millionen Zeilen an Code [Gre15]. Damit werden die Autos immer vernetzter und autonomer. Dabei kommunizieren nicht nur Teile innerhalb eines Autos, wie zum Beispiel Sensoren, miteinander. Ebenfalls kann auch eine externe Kommunikation zu anderen Autos stattfinden. Aus diesem Grund beschreiben Coppola et al. [CM16], dass der Begriff *Connected Car* unterschiedlich definiert wird. Nach Kleberger et al. gliedert sich der Begriff Connected Cars in drei Bereiche. Dazu gehören das Fahrzeug selbst, das Portal des Fahrzeugherstellers und zuletzt die Kommunikationsverbindung zwischen dem Fahrzeug und dem Portal. Während das Portal Services für das Auto bereitstellt, besteht das Auto in dieser Definition aus dem internen Kommunikationsnetzwerk und den Steuergeräten [KOJ11; NLJ08]. Das United States Department of Transportation beschreibt, dass Connected Cars eine Möglichkeit bieten, nicht nur untereinander zu kommunizieren, sondern auch mit der Infrastruktur und den Geräten der Fahrzeuginsassen. Diese Kommunikation findet über ein vernetztes kabelloses Netzwerk statt und ist sicher (im Sinne von „safe“, also ungefährlich) [Uni21a]. In ihrem Artikel ergänzen Coppola et al. [CM16] ihre Definition durch eine Verbindung zum Internet. Zusätzlich wird Fahrzeug-zu-Straße-Kommunikation aufgezählt. Dazu gehört z.B. die Kommunikation mit Werkstätten und wie bei den anderen beiden Definitionen, wird die Kommunikation zwischen Fahrzeugen in die Definition inkludiert. Zuletzt sollen Anwendungen und Funktionalitäten „Infotainment“-Inhalte für die Fahrzeuginsassen bereitstellen, wozu zum Beispiel das Streamen von Musik oder Videos gehört.

Unter Verwendung von Sensorik und Kommunikationstechnologien für Fahrzeuge, wie zum Beispiel die Kommunikation zwischen Autos, soll *autonomes Fahren* möglich werden [CM16]. Ein autonomes Auto ist laut Joe et al. *“ein selbst fahrendes Fahrzeug, welches die Fähigkeit hat die Umgebung wahrzunehmen und sich selbst ohne menschliches Eingreifen zu navigieren“* [JKK+14].

In diesem Zusammenhang stößt man auf den Begriff Internet of Vehicles (IoV). Joy und Gerla [JG17] beschreiben, dass sich die Flotte aus Fahrzeugen, welche Sensordaten messen, zu einem Netzwerk aus autonomen Fahrzeugen entwickelt. Diese Fahrzeuge tauschen Daten auch untereinander aus, mit dem Ziel, unter anderem ihre Fahrzeuginsassen sicher an ihr Ziel zu bringen. Indem Fahrzeuge miteinander kommunizieren, können Unfälle verhindert werden [PKÅ+20]. Diese Entwicklung vergleichen Joy et al. [JG17] mit der Entstehung des Internet of Things (IoT). IoT ist ein Konzept, in welchem es Objekte gibt, sogenannte *Things*. Diese sind im Leben der Menschen vorhanden, zum Beispiel Sensoren oder Smartphones. Die Things kommunizieren und können so miteinander arbeiten [AIM10]. Im IoV gehören externe und interne Sensoren des Fahrzeugs, genauso wie Nachrichten des Fahrers, zum Beispiel aus dessen Social Media, zu den Things. Gemessen werden dabei unter anderem die GPS-Position, Beschleunigung oder der Zustand des Fahrers [JG17].

Dabei stellt der Datenaustausch in diesem Szenario eine große Herausforderung dar. Die Fahrzeuge stellen nicht nur unterschiedliche Sensorik bereit, die Schnittstellen sind zudem unterschiedlich, proprietär und meist herstellerabhängig. Darum ist die Kollaboration der verschiedenen Hersteller äußerst wichtig, um autonom fahrende Autos einführen zu können.

Für die Verwirklichung eines solch komplexen IoT-Konzeptes muss modelliert werden, wie die Daten gesammelt, gespeichert und weiter verarbeitet werden. Dabei soll das ganze in einer *Edge, Fog* oder *Cloud Infrastruktur* aufgebaut werden. Außerdem sollen die Daten in einem *Data Lake* gespeichert werden.

Für die Umsetzung dieses Modells spielt der sogenannte *Digitale Zwilling* eine Rolle. Der Digitale Zwilling (DZ) ist dabei ein Modell aus zwei Teilen: der Physische Zwilling und der Digitale Zwilling [Gri19]. Eine Art des DZ ist die Digital Twin Instance (DTI). Diese spiegelt die digitale Version eines existierenden Produkts wider [Gri17]. Damit sollen Änderungen in der physischen Welt, im digitalen Zwilling widergespiegelt werden. Zusätzlich sollen auch Änderungen am DZ vorgenommen werden können, die dann eine Auswirkung auf den physischen Zwilling haben. Der DZ kann zum Beispiel dafür verwendet werden das Fahrzeug zu überwachen, zu analysieren und Software-Updates darauf zu installieren, wenn es nicht verwendet wird [Krü20]. Der Prozess, die Daten aus der physischen Welt zu sammeln, zu verarbeiten, den Digitalen Zwilling zu generieren, beziehungsweise anzupassen und dann auch Veränderungen am Digitalen Zwilling wieder in den physischen Zwilling weiter zu geben, wird als *Data Loop* bezeichnet. Ziel dieser Arbeit ist es das Konzept der Data Loop zu definieren. Anschließend soll eine prototypische Implementierung entworfen werden, welche schließlich automatisch bereitgestellt werden soll. Für die Umsetzung der Bereitstellung wird der Standard Topology and Orchestration Specification for Cloud Applications (TOSCA) herangezogen.

Gliederung

Die Ausarbeitung ist folgendermaßen gegliedert:

Kapitel 2 – Grundlagen: Zunächst beginnt die Arbeit mit der Erläuterung der wichtigsten Grundlagen. Dazu gehört die Definition verschiedener Begriffe, wie Digitaler Zwilling und Data Lake. Daraufhin folgt die Beschreibung von Cloud Computing und dem Internet of Things, gefolgt von zwei Möglichkeiten für deren Integration: Edge Computing und Fog Computing. Als letztes werden der Modellierungsstandard TOSCA und zwei Applikationen für dessen Verwendung vorgestellt.

Kapitel 3 – Verwandte Arbeiten: Im nächsten Kapitel werden die verschiedenen Komponenten der Data Loop im Hinblick auf Connected Cars betrachtet. Anschließend werden Möglichkeiten für die Modellierung und das darauf folgende Deployment von Anwendungen vorgestellt.

Kapitel 4 – Datenerfassung in Fahrzeugen: Als Nächstes folgt eine Beschreibung, welche Daten in einem Auto bereits jetzt schon gesammelt werden. Im Anschluss wird betrachtet, für was einige der gemessenen Daten im Kontext von Connected Cars verwendet werden können.

Kapitel 5 – Anforderungen an die Data Loop: Nach der Beschreibung des Anwendungsfalls folgt die Definition von Anforderungen an die zu modellierende Data Loop.

Kapitel 6 – Konzept und Modellierung einer Data Loop für Connected Car-Umgebungen: Kapitel 6 beinhaltet die Beschreibung des Data Loop Konzepts, welches im Laufe der Arbeit entworfen worden ist. Hierbei wird auf die verschiedenen Teile der Data Loop eingegangen, welche Architektur und welcher Standard für die Modellierung der Data Loop verwendet worden ist.

Kapitel 7 – Prototypische Implementierung des Data Loop-Konzepts für Connected Car-Umgebungen: Daraufhin folgt ein Kapitel über den Entwurf, die Implementierung, Modellierung und das automatische Bereitstellen eines Prototyps der Data Loop. Hierfür werden zunächst einige der benötigten Technologien beschrieben. Anschließend folgt die Beschreibung der entworfenen Prototypen für die Anwendung, bevor die Modellierung dieser mithilfe von TOSCA beschrieben wird.

Kapitel 8 – Evaluation des Prototyps: Als nächstes werden in Kapitel 8 die in Kapitel 5 festgelegten Anforderungen anhand der Implementierung des Prototyps in Kapitel 7 evaluiert.

Kapitel 9 – Zusammenfassung und Ausblick: Abschließend wird in diesem Kapitel noch einmal die gesamte Arbeit zusammengefasst. Zusätzlich folgt ein Ausblick auf mögliche weitere Arbeiten, welche auf der Data Loop aufbauen.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der Arbeit besprochen. Dazu gehört zunächst der Digitale Zwilling. Für den Entwurf der Data Loop ist daraufhin die Definition der Begriffe Cloud Computing, Internet of Things, Edge und Fog Computing und Data Lakes wichtig. Zuletzt wird der OASIS-Standard TOSCA vorgestellt, der im späteren Verlauf der Arbeit für die Modellierung und das automatische Deployment der Data Loop verwendet werden soll.

2.1 Digitaler Zwilling

Grieves [Gri19] führt das Konzept zunächst namenlos schon 2002 in einer Präsentation beim Society of Manufacturing Engineering (SME) Management Forum ein. Erst später übernimmt Grieves in einer gemeinsamen Arbeit mit Vickers dessen Bezeichnung Digitaler Zwilling (DZ) [Gri17; PVL+10]. Grieves beschreibt den DZ als Modell aus zwei Teilen. Wie in 2.1 zu erkennen, gibt es den Physischen Zwilling und den DZ. Der DZ ist hierbei ein Konstrukt aus Informationen des Physischen Zwillings. Dabei soll der DZ die selben oder bessere Informationen als sein physischer Gegenpart liefern. Der Physische Zwilling entwickelt sich nach der Beschreibung von Grieves zu einem sogenannten Smart, Connected Product System (SCPS). Diese werden durch IoT ermöglicht. Dazu gehören zum Beispiel Flugzeuge und Raumfahrzeuge [Gri19].

Es gibt verschiedene Arten von DZs [Gri17]. Zum einen gibt es den Digital Twin Prototyp (DTP). Dies ist der Prototyp des zu entstehenden Produkts. Dabei enthält es alle Informationen, die das Produkt beschreiben und die nötig sind, um dieses herstellen zu können. Dazu gehören zum Beispiel die Anforderungen, ein 3D-Modell oder auch eine Liste der benötigten Materialien. Der zweite DZ ist die DTI, dies ist eine digitale Version eines einzelnen physischen Produkts. Die DTI und das jeweilige Produkt bleiben dabei über die gesamte Laufzeit verknüpft. Dabei enthält dieser zum Beispiel ein 3D-Modell des Produkts, eine Liste aller aktuellen Komponenten, als auch der Alten. Zusätzlich werden die Produktionsschritte dokumentiert, genauso wie an dem Produkt durchgeführte Änderungen. Auch dazu aufgenommene Sensordaten sind Teil der DTI [Gri17]. Des Weiteren gibt

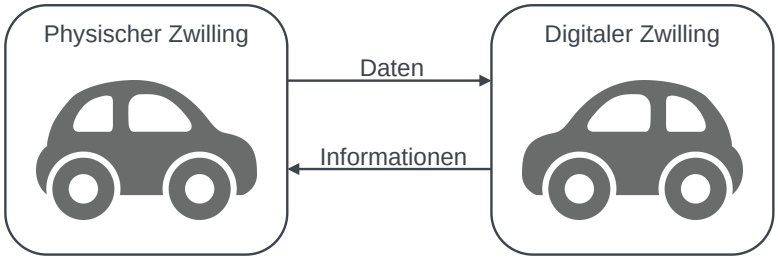


Abbildung 2.1: Modell des Digitalen Zwillings (nach dem Vorbild von: [Gri19])

es Digital Twin Aggregates (DTAs). Diese sind eine Kombination aus allen DTIs und können zum Beispiel für Vorhersagen verwendet werden. So kann man möglicherweise durch die Aufnahme bestimmter Sensordaten auf die Fehlfunktion einer Komponente schließen.

Ursprünglich ist dieses Konzept für Product Lifecycle Management (PLM) entworfen worden. [Gri19]. Der Digitale Zwilling verändert sich im Laufe des Zyklus, welcher laut Grieves et al. aus vier Phasen besteht: Kreation, Produktion, Betrieb und Entsorgung. Der Physische und der Digitale Zwilling bleiben durch die Phasen hindurch miteinander verknüpft. In der ersten Phase existiert das physische System noch nicht, da zunächst der DTP erstellt wird. Erst in der Produktionsphase werden die Physischen Zwillinge erstellt. Dabei wird eine dazugehörige DTI erschaffen. Im Betrieb besteht eine bilaterale Verbindung zwischen den beiden Zwillingen. Änderungen am Physischen Zwilling werden im Digitalen Zwilling wider gespiegelt. Diese Informationen können dann zum Beispiel für Vorhersagen über das physische System verwendet werden. Die letzte Phase, die Entsorgung, wird hierbei nicht weiter beachtet [Gri17].

Im Laufe der Arbeit wird vom DZ gesprochen. Hierbei ist im Allgemeinen dann die DTI gemeint.

2.1.1 Anwendungsfälle für Digitale Zwillinge

Grieves [Gri19] beschreibt verschiedene Anwendungsfälle für DZ. Einer davon ist das Configuration Management. Hierbei wird der DZ darüber auf dem Laufenden gehalten, was sich an seinem physischen Gegenstück verändert. Im DZ werden die Konfigurationen zu den verschiedenen Zeitpunkten widergespiegelt. Dies kann zum Beispiel in der Aktualisierung des Produkts Verwendung finden, damit immer die aktuellste Version der Software installiert ist. Wenn der DZ den Zustand eines Produkts zu jeder Zeit widerspiegelt, so kann dieser für die Überwachung verwendet werden. Zusätzlich könnte der DZ den Zustand des Physischen Zwillings beurteilen. Bei einer Fehlfunktion kann zum Beispiel eine Anpassung der Software stattfinden oder der Dienstleister wird benachrichtigt, dass das Produkt repariert werden muss. Eine weitere Anwendungsmöglichkeit ist die zuvor erwähnte Nutzung von DTAs zur Vorhersage von Systemversagen, damit bereits im Voraus Gegenmaßnahmen getroffen werden können. Des Weiteren ist die Nutzung der DZs als erweiterte Realität möglich. Hierbei soll die Arbeit am Physischen und Digitalen Zwilling gleichzeitig geschehen. Einem Menschen, welcher an einem physischen System arbeitet, sollen die vom DZ empfangenen und verarbeiteten, beziehungsweise analysierten Daten zugänglich sein. Um dies zu ermöglichen, könnten diese Daten zum Beispiel auf einer Brille für den Arbeiter visualisiert werden. Nicht zuletzt kann der DZ darin Einsatz finden, die Software des Produkts zu analysieren und Änderungen vorzuschlagen. Werden diese Anpassungen durchgeführt, wird dies im DZ dokumentiert. Ferner kann jener verhindern, dass unerlaubte Änderungen am Physischen Zwilling durchgeführt werden. Zuletzt gibt der DZ auch die Möglichkeit Rückmeldung über die Leistung des Systems zu geben.

2.2 Data Lakes

> *„Ein Data Lake ist eine skalierbare Datenmanagementplattform für Daten jeder Struktur, die in ihrem Rohformat gespeichert werden, um Analysen ohne vordefinierte Anwendungsfälle zu ermöglichen. Zudem beinhaltet ein Data Lake auch vorverarbeitete Daten für eine erhöhte Effizienz. Daten aus mehreren Quellen werden in einem Data Lake zusammengeführt und verschiedene Nutzergruppen greifen darauf zu. Ein Data Lake kann mehrere Arten von Speichersystemen umfassen. Data-Lake-Governance und Metadatenmanagement sind zentrale Bestandteile eines Data Lake.“*
>
> Giebler et al. [GGH+20]

Das Modell des DZ benötigt viele Daten und auch viel Rechenleistung [Gri17]. Das Konzept des *Data Lakes* beschäftigt sich damit, große und heterogene Mengen an Daten zu verwalten und deren Wert heraus zu arbeiten [GGH+20]. Für den weiteren Verlauf der Arbeit ist es darum wichtig, den Begriff des Data Lakes genauer zu definieren, da für diesen aktuell noch keine allgemeingültige Definition in der Literatur zu finden ist. Die erste Erwähnung eines Data Lakes macht Dixon 2010 auf seinem Blog. Hier beschreibt er, dass die Daten in einem Data Lake unverarbeitet abgelegt werden. Daraufhin können die Nutzer zum Beispiel Daten untersuchen oder Stichproben nehmen. Dabei sollen die Daten aus einer einzigen Quelle stammen [Dix10]. Im Gegensatz dazu beschreibt O'Leary in seinem Artikel, dass Data Lakes mehrere Quellen haben können [OLe14]. Nargesian et al. beschreiben ebenfalls, dass die Daten aus verschiedenen Strömen kommen können. Zusätzlich geben sie an, dass der Data Lake aus verschiedenen Speichersystemen bestehen kann. Dabei haben die Daten kein einheitliches Format und besitzen keine oder unterschiedlich strukturierte Metadaten. Zusätzlich soll er sich selbst verändern können [NZM+19]. Nicht alle stimmen mit dieser Sicht überein. Andere, wie O'Leary, bringen den Data Lake in Verbindung mit der Verwendung von Hadoop[1], nicht wie Nargesian et al. mit mehreren Speichersystemen [NZM+19; OLe14]. Dixon jedoch teilt die Ansicht, dass ein Data Lake verschiedene Speichersysteme für dessen Realisierung verwenden soll [Dix14]. Einige Beispiele sind HBase, Cassandra, CouchDB oder MongoDB [Mat17]. Im Gegensatz zu Nargesian et al. schreiben Hai et al. [HGQ16] in ihrem Artikel den Metadaten eine hohe Bedeutung zu. Ohne diese ist die Struktur der Daten nicht ersichtlich und die Daten sind nicht verwendbar. Dies beschreiben die Autoren als einen *Data Swamp*. Damit sich ein Data Lake nicht in einen solchen Data Swamp verwandelt ist Data-Lake-Governance von Nöten, die unter anderem dafür sorgt, dass die Metadaten verwaltet werden können und die Qualität der Daten gesichert werden kann [Mat17]. Eine weitere Definition spricht davon, Daten im Data Lake vor zu

[1] Zu finden unter: https://hadoop.apache.org/

verarbeiten, um eine spätere Analyse zu ermöglichen [TSRC15]. Auch Giebler et al. [GGH+20] entwickeln aufgrund der Uneinigkeit über eine Definition des Data Lakes ihre eigene Definition, welche besagt, dass:

- Dieser skalierbar ist
- Daten in ihrem Rohformat gespeichert werden und eine beliebige Struktur haben
- Auch verarbeitete Daten gespeichert werden können
- Es verschiedene Datenquellen und Nutzer gibt
- Verschiedene Speichersysteme verwendet werden können
- Eine Data-Lake-Governance und ein Metadatenmanagement existiert

Diese Definition wird in einer vereinfachten Version für die weitere Arbeit verwendet. Die vereinfachte Version des Data Lakes soll skalierbar sein, einen Metadaten-Speicher haben und unterschiedliche Speichersysteme verwenden. Die Verwaltung findet dabei über Java-Programme statt.

2.3 Cloud Computing

Das National Institute of Standards and Technology (NIST) definiert Cloud Computing wie folgt: *„Cloud Computing ist ein Modell zur Ermöglichung von ubiquitärem, zweckmäßigem, auf Abruf verfügbarem Netzwerkzugang zu einem geteilten Vorrat an konfigurierbaren Computer Ressourcen (zum Beispiel: Netzwerke, Server, Speicher, Anwendungen und Services), welche schnell zur Verfügung gestellt und mit minimalem Management-Aufwand oder Provider-Interaktion freigegeben werden können“* [MG11].

2.3.1 Charakteristiken

Cloud Computing werden mehrere Charakteristiken zugeschrieben: *On-Demand Self-Service, Broad Network Access, Resource Pooling, Rapid Elasticity* und *Measured Service* [MG11]. On-Demand Self-Service sagt aus, dass die Menge an IT-Ressourcen vom Nutzer nach eigenem Bedarf geregelt werden kann. Außerdem ist dies ohne die Interaktion mit dem Service-Provider möglich. Die nächste Charakteristik, Broad Network Access, beschreibt, dass die Ressourcen über das Netzwerk zur Verfügung stehen. Ressource Pooling besagt, dass die Ressourcen in einem großen „Pool“ zur Verfügung gestellt werden, welcher mehreren Nutzern zugänglich ist [FLR+14; MG11]. Durch Rapid Elasticity können die Kapazitäten nach Belieben angepasst werden. Da dies schnell und teils automatisch geschieht, erscheinen die Kapazitäten für den Nutzer als unendlich [MG11]. Die letzte Charakteristik ist Measured Service. Diese sagt aus, dass die verwendeten IT-Ressourcen gemessen werden [FLR+14; MG11]. Aufgrund dieser Messung ist es möglich ein *Pay-Per-Use* Modell einzuführen, bei dem der Nutzer nur das zahlt, was er auch verbraucht hat [FLR+14].

2.3.2 Service Modelle

Neben den Charakteristiken gibt es auch verschiedene Service Modelle, welche dem Nutzer zur Verfügung gestellt werden. Dazu gehören: *Infrastructure as a Service (IaaS), Platform as a Service (PaaS)* und *Software as a Service (SaaS)*. IaaS gibt dem Nutzer die Möglichkeit eigenständig Cloud Ressourcen, wie Speicher, zur Verfügung zu stellen. Damit kann ein Nutzer selbst entscheiden, welches Betriebssystem er verwendet und welche Anwendungen installiert werden. PaaS hingegen gibt dem Nutzer lediglich die Möglichkeit Anwendungen selbst auf der Infrastruktur der Cloud zu installieren. Dies können auch eigens angefertigte Applikationen sein. Das letzte Service Modell, SaaS, bedeutet, dass dem Nutzer eine bereits auf der Cloud Infrastruktur installierte Applikation bereit gestellt wird [MG11].

2.3.3 Bereitstellung

Für die Bereitstellung der Cloud gibt es ebenfalls verschiedene Modelle, zwischen denen gewählt werden kann: Die *Private Cloud, Community Cloud, Public Cloud* und *Hybrid Cloud*. Wird eine Private Cloud genutzt, so wird die Cloud Infrastruktur nur von einer Organisation verwendet. Dabei wird diese entweder durch die Organisation selbst, eine andere Partei, oder durch beide gemeinsam verwaltet. Zusätzlich kann die Cloud auf den Servern der Organisation selbst, oder woanders positioniert sein. Des Weiteren gibt es die Community Cloud, welche einer Gruppe von Nutzern zur Verfügung gestellt wird, die geteilte Interessen besitzen. Die Verwaltung ähnelt hierbei der oben beschriebenen einer Private Cloud. Die Public Cloud hingegen kann von jedem verwendet werden. Sie wird dann von einer Firma, einer anderen Partei oder mehreren Parteien zusammen verwaltet. Anders als bei den bisher vorgestellten Modellen, befindet sich die Cloud immer beim Cloud Provider. Zuletzt gibt es noch die Hybrid Cloud, welche zwei oder sogar mehr der obigen Modelle kombiniert. Die Modelle bleiben aber in sich abgetrennt [MG11].

2.4 Internet of Things

IoT ist ein Konzept, in welchem es Objekte gibt, sogenannte *Things*. Diese sind im Leben der Menschen vorhanden, zum Beispiel Sensoren oder Smartphones und kommunizieren miteinander [AIM10]. Auch Smart Lights oder Smart Switches zählen zu den IoT-Geräten, also Things [YLH+18]. Diese Things können miteinander arbeiten [AIM10]. Die Fähigkeit zur Kommunikation sorgt dafür, dass die Geräte von einem Nutzer oder anderen IoT-Geräten ferngesteuert werden können. Das Ziel dabei ist es, den Wert der Geräte für den Nutzer zu erhöhen. Dies wird ermöglicht, indem zusätzliche Funktionalitäten angeboten werden, welche ohne eine Vernetzung der Geräte nicht möglich wäre [LL15].

2.4.1 Klassische IoT-Anwendung

Ein klassisches Anwendungsbeispiel liegt im Bereich der Automation. Es behandelt die Themen Smart Home und Smart Office. Eine solche Anwendung hat zum Ziel, Energiekosten zu sparen und das Leben der Nutzer komfortabler zu machen. Hierfür werden Sensoren und Aktoren verwendet [BSS+09]. Die verwendeten Sensoren können beispielsweise die Temperatur und Helligkeit

messen, ebenso wie die anwesenden Personen bestimmen [ADS+06]. Anschließend werden die Daten dafür verwendet, die Umgebung mithilfe der Aktoren anzupassen. Als Beispiel hierfür können die Temperatur und das Licht automatisch geregelt werden [BSS+09]. Ein Beispiel für ein Smart Office System beschreiben Kastner et al. [KKJ+14]. Das Ziel des Systems ist es, den Angestellten und Besuchern eine angenehme Umgebung zu bieten und dabei Energie zu sparen. Kommt ein Angestellter an seinen Arbeitsplatz, soll das Licht und die Temperatur angepasst werden, sodass eine für ihn angenehme Atmosphäre entsteht. Damit die Umgebung den Wünschen des Angestellten entspricht, kann dieser über sein Smartphone Einstellung vornehmen. Zusätzlich soll das System den Kalender verwenden, um festzustellen, wann ein Meeting stattfindet und dann ebenfalls die Helligkeit und Temperatur anpassen. Meldet ein entsprechender Sensor, dass im Warteraum ein Besucher eingetroffen ist, soll auch hier die Umgebung angepasst werden. Damit das Ziel der Reduktion der Energiekosten erreicht werden kann, gibt es Sensoren, welche feststellen können, ob ein Raum leer ist. Daraufhin können auch hier Licht und Temperatur entsprechend angepasst werden.

2.5 Edge und Fog Computing

Edge und Fog Computing sind Möglichkeiten, IoT und die Cloud miteinander zu integrieren. Um zu entscheiden, welche der beiden Möglichkeiten am Besten für die Data Loop geeignet ist, werden diese nun beschrieben.

2.5.1 Edge Computing

Die Nutzung von Edge Computing soll Technologien die Möglichkeit geben, Daten am Rande des Netzwerkes, also am *Edge* zu verarbeiten, anstatt sie in der Cloud zu verarbeiten. Dadurch findet die Datenverarbeitung näher am Nutzer statt [SD16]. In Abbildung 2.2 ist die Architektur einer solchen Anwendung abgebildet. Diese hat drei Ebenen: Die IoT-, Edge- und die Cloud-Ebene. Die Cloud sammelt hierbei Daten aus Datenbanken oder von den, in der IoT-Ebene abgebildeten, Endgeräten. Bei den Endgeräten handelt es sich zum Beispiel um Sensoren. Anschließend werden die Daten über sogenannte *Edge Devices* an die Cloud weiter gegeben. Zusätzlich zur Weitergabe von Daten kümmern sich die Edge Devices sowohl um Anfragen der Endgeräte an die Cloud, die Datenverarbeitung und Geräteverwaltung, als auch die Sicherung der Privatsphäre [SD16]. Dabei können Smartphones, Wearables oder Spielecontroller zu Edge Devices werden [VWB+16].

2.5.2 Fog Computing

Eine weitere Integrationsmöglichkeit bietet Fog Computing. Genauso wie bei Edge Computing soll auch bei dieser Möglichkeit die Anwendung so nahe wie möglich am Nutzer platziert werden [LGL+15]. In Abbildung 2.3 ist die Architektur einer solchen Fog Computing Anwendung visualisiert. Wie bei Edge Computing befindet sich auch Fog Computing am Rande des Netzwerks. Den Endgeräten in der IoT-Ebene sollen hierbei Verarbeitungs-, Speicher- und Netzwerk-Services zur Verfügung gestellt werden [BMZA12]. Betrachtet man die obige Beschreibung der Cloud, verwenden diese und Fog Computing die selben Ressourcen. Die zwischen der Cloud und den Endgeräten eingeordnete Ebene soll zwischen diesen eine Brücke bilden. Die Fog-Ebene besteht

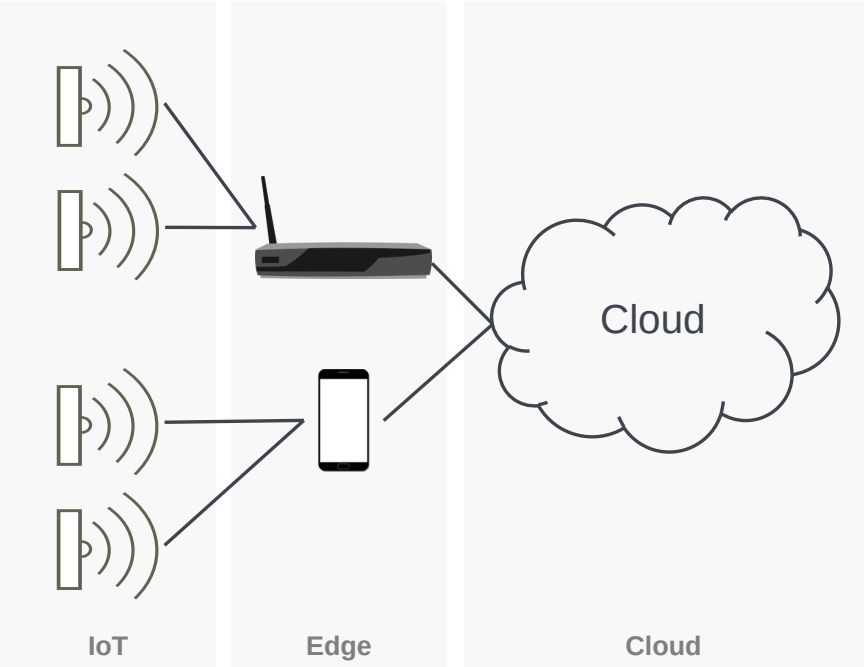


Abbildung 2.2: Architektur von Edge Computing (nach dem Vorbild von: [SD16])

aus virtuellen Servern. Sie bieten den Endgeräten aus der IoT-Ebene Speicher- und Rechenkapazität, sowie Kommunikationsmöglichkeiten. Durch ihre Position in der Edge, sind die Server nahe am Nutzer und können somit diesen einen schnelleren Service garantieren. Haben Fog-Server, zum Beispiel über Wireless Local Area Network (WLAN), mit Endgeräten kommuniziert, können diese die erhaltenen Daten mit ihren Cloud-ähnlichen Ressourcen verarbeiten. Dabei bietet die Verbindung der Fog-Server zur Cloud weitere Ressourcen zur Verarbeitung [LGL+15]. Die Motivation hinter dem Einsatz von Fog Computing sind Anwendungen, welche schlecht unter Verwendung der Cloud funktionieren. Ein solches Beispiel sind Spiele, da diese eine niedrige Latenz voraussetzen. Auch räumlich-verteilte Anwendungen, wie Sensornetzwerke oder Smart Connected Vehicle gehören zu solchen Anwendungen [BMNZ14].

Vergleicht man nun Fog und Edge Computing, so wirken diese auf den ersten Blick sehr ähnlich. Der Unterschied liegt dabei in den durch die jeweiligen Geräte zur Verfügung gestellten Ressourcen. So befinden sich auf der Fog-Ebene Server, welche Cloud-ähnliche Ressourcen bereit stellen [LGL+15]. Die Edge Devices hingegen haben nur eine begrenzte Menge an Ressourcen [DB16].

2.6 Topology and Orchestration Specification for Cloud Applications

Bei TOSCA handelt es sich um einen Standard der Organisation OASIS. Die aktuelle Version 1.0 dieser Spezifikation ist aus dem Jahr 2013 [OAS21a]. Der Standard wird verwendet, um automatische Bereitstellung und die Verwaltung von zusammengesetzten Anwendungen zu ermöglichen. Dabei wird die Struktur einer solchen Anwendung mithilfe von Topologien beschrieben. TOSCA

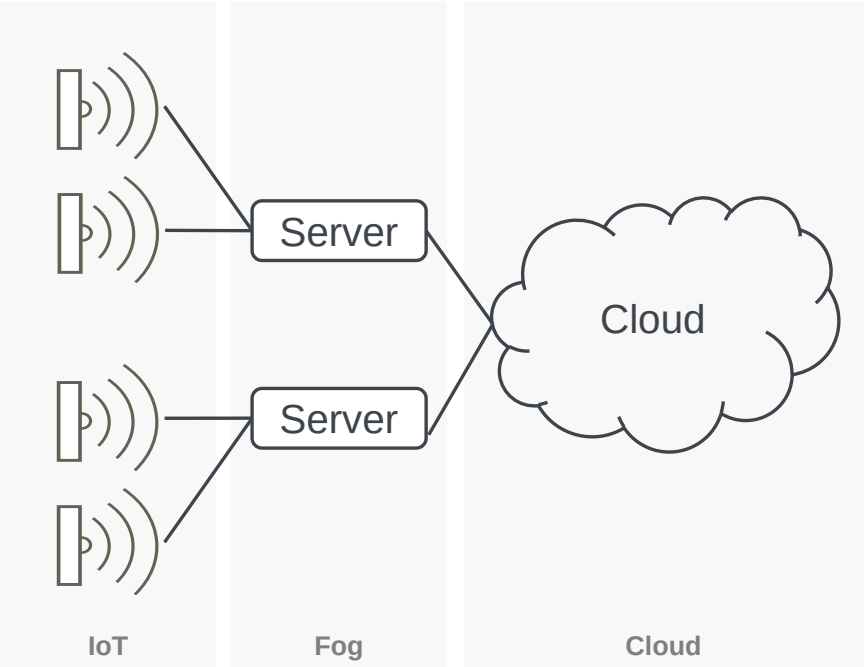


Abbildung 2.3: Architektur von Fog Computing (nach dem Vorbild von: [LGL+15])

ist eine XML-basierte Modellierungssprache. Der Standard hat die folgenden zwei Hauptkonzepte: Die Anwendungstopologie und Managementpläne. Die Anwendungstopologie ist ein Graph, dessen Knoten die Komponenten der Anwendung und Kanten deren Beziehungen zueinander beschreiben. Diese Knoten und Kanten können wiederum weitere Eigenschaften haben. Dazu gehören zum Beispiel Managementoperationen und Artefakte, welche gebraucht werden, um die betroffene Komponente auszuführen. Damit beschreibt die Topologie ebenfalls Verwaltungsfähigkeiten [BBKL14]. In 2.4 ist eine TOSCA-basierte Anwendung dargestellt. Dabei handelt es sich um ein *Topology Template*. Wie zuvor beschrieben, besteht die Topologie eines IT-Services aus Knoten und Kanten. Diese bilden einen gerichteten Graphen. Die Knoten sind dabei *Node Templates*, welche von einem bestimmten *Node Type* sind. Der Node Type gibt dann die Eigenschaften (*Node Type Properties*) und die Operationen (*Interfaces*) an, mithilfe derer die Komponente verändert werden kann. Die Kanten sind sogenannte *Relationship Templates* und stellen die Beziehungen zwischen den Node Templates dar. Wie auch bei den Node Templates, basieren die Relationship Templates auf *Relationship Types*. Für die Node Types existieren zwei Arten von *Node Type Implementations*. Diese enthalten ausführbaren Code für die Node Types. *Implementation Artifacts* stellen den Code für die Managementoperationen zur Verfügung. Dazu gehören zum Beispiel die Bereitstellung, Konfiguration oder Aktualisierung einer Komponente. Zusätzlich gibt es *Deployment Artifacts*, welche die nötige Implementierung enthalten, um das jeweilige Node Template umzusetzen. Ein Beispiel hierfür ist ein Abbild eines Betriebssystems oder die Installationsdatei für eine Anwendung, wie Tomcat. Die Artefakte werden dabei in *Artfact Templates* definiert und von den Deployment und Implementation Artifacts im Node Type referenziert [BBKL14; OAS21b]. Die erstellte TOSCA-Topologie wird in einer Cloud Service Archive (CSAR)-Datei gebündelt, welche dann auch die Implementierungen enthält [OAS21a].

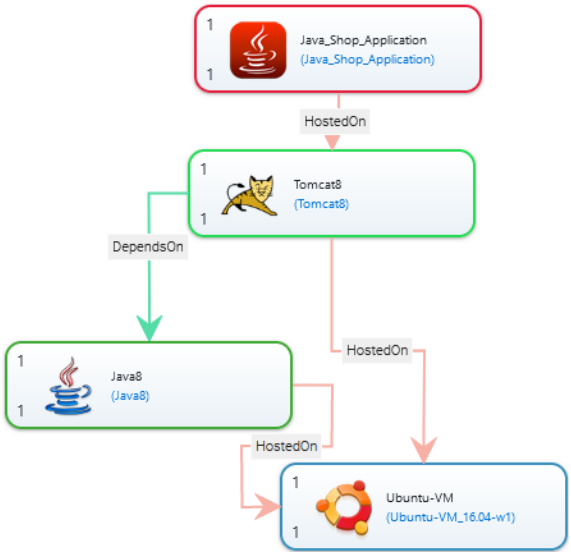


Abbildung 2.4: Beispiel eines TOSCA-Modells (modelliert in Winery)

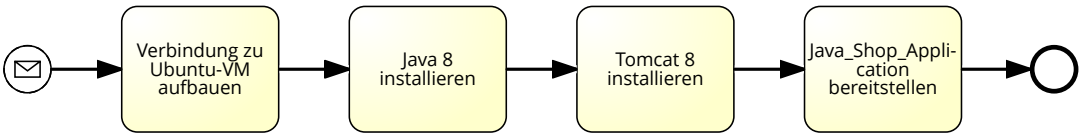


Abbildung 2.5: Managementplan des TOSCA-Modells in Abbildung 2.4 in BPMN

Managementpläne (*Management Plans*) können automatisch ausgeführt werden. Sie verwenden die Verwaltungsfähigkeiten aus den Topologien. Die daraus entstehenden Pläne können verwendet werden, um Anwendungen automatisch bereitzustellen, zu konfigurieren oder zu verwalten. Dies beschreibt die Beziehung zwischen den beiden Konzepten: Nachdem der Managementplan aufgerufen wird, ruft dieser Operationen auf den Knoten in der Topologie auf. Pläne in TOSCA werden in Workflow Sprachen, wie Business Process Model and Notation (BPMN) oder Business Process Execution Language (BPEL), geschrieben [BBKL14]. Das BPMN-Modell, welches in Bild 2.5 dargestellt ist, zeigt den Managementplan zur Topologie in Abbildung 2.4. Hier sind die auszuführenden Schritte in ihrer Reihenfolge zu sehen. Zunächst wird eine Verbindung zu einer Ubuntu-Virtuelle Maschine (VM) hergestellt. Daraufhin wird erst Java und dann Tomcat darauf installiert. Anschließend wird eine Java-Web-Anwendung auf dem Tomcat bereit gestellt.

Die Ausführung der Topologien kann über zwei Verarbeitungsansätze erfolgen: *imperativ* oder *deklarativ*. Für den imperativen Ansatz werden die gesamte Topologie und die Managementpläne benötigt, was bedeutet, dass genau angegeben wird, wie eine Anwendung strukturiert ist und verwaltet wird. Die genannten Verwaltungspläne sind Teil von TOSCA und sind zuvor beschrieben worden. Die Implementierung der nötigen Managementoperationen sind dabei in Implementation Artifacts enthalten. Der deklarative Verarbeitungsansatz wiederum gibt an, welche Strukturen und Managementoperationen nötig sind. Statt der Nutzung von Plänen, wird hier die Logik für die Bereitstellung und Verwaltung von der Laufzeitumgebung durchgeführt. Aus diesem Grund wird eine genaue Definition der Node und Relationship Types benötigt, damit Annahmen über die Managementoperationen gemacht werden können [BBH+13; OAS21a].

2.6.1 Winery

Winery gibt die Möglichkeit TOSCA-basierte Applikationen im Browser zu modellieren. Mithilfe von Winery können sowohl Typen verwaltet, als auch graphische Modelle erstellt werden. Abbildung 2.4 ist unter Verwendung von Winery entstanden. Die graphische Modellierung entsteht mithilfe des *Topology Managers*, aus welchem der Screenshot des Beispiels stammt. Hierfür wird zunächst ein sogenanntes *Service Template* erstellt. Anschließend kann der Topology Manager geöffnet werden. Zur Erstellung der Topologie können Knoten aus einer Liste von Node Types ausgewählt und über Drag-and-Drop in den Editor gezogen werden. Diese werden, wie in 2.4 zu erkennen, als Rechtecke dargestellt. Die im Editor dargestellten Knoten sind die Node Templates, denen zum Beispiel die Anforderungen und Werte der jeweiligen Eigenschaften zugewiesen werden können. Zusätzlich können Beziehungen, also Relationship Templates, zwischen den Knoten festgelegt werden. Dies ist durch einen Klick auf einen Knoten möglich, woraufhin die gewünschte Beziehung ausgewählt und mit einem anderen Knoten verbunden werden muss. In Abbildung 2.4 sind diese als Pfeile dargestellt. Um zum Beispiel Typen und Implementierungen zu erstellen wird der sogenannte *Element Manager* verwendet. Die dritte Komponente von Winery ist das *Repository* für die Speicherung aller Daten. Ist eine Topologie fertig gestellt, folgt die Erstellung der Managementpläne. Dafür benötigt Winery allerdings ein anderes Modellierungstool, da dies keine Funktion der Applikation ist. Anschließend kann eine CSAR-Datei heruntergeladen werden, welche dazu verwendet werden kann, die modellierte Anwendung bereit zu stellen [KBBL13]. Winery unterstützt jedoch nicht das Ausführen erstellter Topologien. Hierfür wird eine Laufzeitumgebung, wie OpenTOSCA benötigt [BBH+13; KBBL13].

2.6.2 OpenTOSCA

OpenTOSCA ist eine Implementierung des TOSCA-Standards [BEK+16]. Wie bereits erwähnt, kann die Laufzeitumgebung OpenTOSCA dazu verwendet werden, TOSCA Applikationen automatisch bereit zu stellen [BBH+13]. Durch die Erstellung von *Build Plans* wird sowohl der imperative, als auch der deklarative Verarbeitungsansatz unterstützt [BBK+14]. Das OpenTOSCA Ecosystem besteht unter anderem aus Winery und dem OpenTOSCA Container, welcher die Laufzeitumgebung darstellt [BEK+16]. Die Hauptaufgaben von OpenTOSCA sind die Ausführung der Managementoperationen, das Ausführen von Plänen und die Verwaltung des Zustandes. Für die Bereitstellung einer Anwendung wird eine Anfrage an die Kontrollkomponente versendet. Diese verwaltet und interpretiert die verschiedenen Komponenten. Werden die Managementoperationen durch Implementation Artifacts bereit gestellt, welche auch durch bereits laufende Services bereit gestellt werden können, so werden sie von Plugins der *Implementation Artifact Engine* ausgeführt, sodass diese für die Pläne verfügbar sind. Die Plugins stellen die Artefakte nicht nur bereit, die Endpunkte der bereitgestellten Artefakte werden zusätzlich in einer Datenbank gespeichert. Managementpläne werden von der sogenannten *Plan Engine* verarbeitet. Die von der Plan Engine bereitgestellten Plugins binden die Services aus den Plänen an die zuvor gespeicherten Endpunkte der Operationen und deployen die Managementpläne. Durch die Ausführung der Managementpläne wird eine Instanz der Anwendung erstellt [BBH+13].

3 Verwandte Arbeiten

Im Laufe der Arbeit soll ein Modell der Data Loop entwickelt werden, welches anschließend deren automatische Bereitstellung ermöglicht. Da die Data Loop ein neues Konzept ist, gibt es hierzu keine verwandten Arbeiten. Aus diesem Grund werden in diesem Kapitel zunächst die verschiedenen Komponenten der Data Loop im Kontext von Connected Cars betrachtet. Dazu gehören Data Lakes und der Digitale Zwilling. Zusätzlich wird die Verwendung von Messagingsystemen diskutiert, da die verschiedenen Komponenten der Data Loop miteinander kommunizieren müssen. Anschließend wird eine Auswahl existierender Standards zur Modellierung und Bereitstellung von Applikationen, beziehungsweise Systemen, vorgestellt. Im Automobilbereich sind hierzu kaum ähnliche Arbeiten zu finden, darum werden im Folgenden auch Artikel betrachtet, welche sich mit der Bereitstellung komplexer Softwaresysteme in anderen Bereichen beschäftigen. Dazu gehören Cloud-, IoT- und Pipeline-Anwendungen, da die gesammelten Daten verarbeitet werden müssen und darum auch Datenverarbeitungspiplines von Nöten sind.

Data Lakes

Data Lakes finden in der Automobilindustrie ihren Einsatz. Fahrzeuge produzieren viele Daten, welche über das Internet in einem Data Lake abgespeichert werden können. Diese von verschiedenen Fahrzeugen stammenden Daten bilden einen großen Data Lake, welcher für Berechnungen verwendet werden kann, die weiteren Wert bringen können. Dazu gehören beispielsweise das Wetter oder die Verkehrssituation. Anschließend können diese Analyseergebnisse von dem Auto, oder von anderen Parteien verwendet werden [Bon18].

Luckow et al. [LKM+15] stellen eine auf Hadoop basierende Architektur für einen automobilen Data Lake vor. Die Autoren beschreiben, dass ihr Data Lake dabei die für den Automobilbereich typische Verwendung von SQL unterstützt.

In der Praxis verwendet die BMW Group beispielsweise einen Data Lake, welcher auf Amazon Web Services (AWS) basiert. Ziel war es unter anderem, Echtzeit-Zugriff auf Telemetriedaten zu ermöglichen. Zudem sollen Analysen und Machine Learning im Data Lake möglich sein. Hierbei wird eine Terabyte-große Menge an Daten zu ihren Fahrzeugen verarbeitet. So überwacht die Firma zum Beispiel Fehlermeldungen, um Probleme verschiedener Baureihen zu finden [Ama21].

Aus einer Präsentation von Metje et al. [Met20] geht hervor, dass auch Bosch einen Data Lake verwendet, welchen sie REDLake nennen. Im Zusammenhang mit ihrem Enterprise Data Lake werden relationale Datenbanken und Hadoop als Werkzeuge erwähnt.

Um auch im Kontext dieser Arbeit Daten verschiedenster Art abspeichern zu können, wird ein Data Lake benötigt. Hierfür werden verschiedene Technologien, wie Hadoop, relationale Datenbanken oder AWS verwendet. Auch in dieser Arbeit soll der Data Lake darum aus verschiedenen Technologien zusammengesetzt werden.

Digitaler Zwilling

Im Bereich Connected Cars gibt es verschiedene Ansätze für die Verwendung von DZs. Durch die zunehmende Vernetzung der Fahrzeuge steigt die Menge an eingesetzter Software immer mehr. Da diese konstant auf dem neuesten Stand bleiben und sichergestellt werden muss, dass die Software sicher ist, diskutiert Krüger [Krü20] den Einsatz von DZs für die Verwaltung von Updates. Hierdurch soll die Informationssicherheit zu jedem Zeitpunkt gesichert werden. Hierbei soll mithilfe des DZs die Integrität der Software sichergestellt werden. Zusätzlich sollen Software-Updates verifiziert und auch zu einem passenden Zeitpunkt durchgeführt werden, da mithilfe eines Nutzungsprofils die Wahrscheinlichkeit festgestellt werden kann, ob das Auto in nächster Zeit verwendet wird.

Rassõlkin et al. [RVKK19] diskutieren in ihrem Artikel die Nutzung von DZs für autonomes Fahren von elektrischen Autos. Hierfür beschreiben die Autoren eine Kombination aus Machine Learning und dem DZ für die Betreibung des Antriebs.

Auch Veledar et al. [VD19] erörtern die Verwendung von DZs im Kontext des autonomen Fahrens. Hier soll der DZ dazu verwendet werden, die Security und Safety zu bewahren. Auch in diesem Fall wird der DZ mit verschiedenen Analysemöglichkeiten verbunden. Beispielsweise werden von den Autoren hierfür Künstliche Intelligenz, Echtzeit-Vorhersage-Analysen und Vorhersagen mithilfe der IoT-Sensordaten aufgezählt. Im Laufe der Arbeit wird eine Methode vorgestellt, die den DZ verwendet, um verschiedene Schwachstellen zu behandeln.

Kumar et al. [KMC+18] kombinieren verschiedene Ansätze, um Verkehrsmanagement in einer immer mehr vernetzten Stadt zu realisieren. Hierfür nutzen sie unter anderem einen Data Lake, welcher Verkehrs- und Fahrzeugdaten speichert. Zusätzlich verwenden sie Machine Learning und ein virtuelles Modell für Fahrzeuge. Dieses virtuelle Modell ist ein DZ. Die Autoren beschreiben, dass die verwendeten Technologien intelligentere Transportmöglichkeiten und das Verhindern von Staus ermöglichen.

Wie Kumar et al. es in ihrer Arbeit beschrieben haben, sollen auch der Data Lake und der DZ in der Data Loop eine gemeinsame Verwendung finden. Auf diese Weise sollen in der Data Loop dann nicht nur Staus verhindert, sondern auch der Zustand der Fahrzeuge überwacht werden können, um zum Beispiel Updates durchzuführen.

Messagingsysteme

Um Daten aus den Fahrzeugen zu übermitteln, damit diese gespeichert und verarbeitet werden können, werden Messagingsysteme benötigt. Eine Möglichkeit für den Nachrichtenaustausch im Bereich von Connected Cars ist MQTT. Skerrett [Ian20] beschreibt, dass MQTT in der Praxis dazu verwendet wird, Connected Cars mit der Cloud zu verbinden. Dabei kann MQTT eine anhaltende Verbindung zum MQTT-Broker aufbauen. Verliert ein Auto die Verbindung, kann diese einfach wieder hergestellt werden.

Eine weitere Möglichkeit für das Versenden von Nachrichten ist Apache Kafka. Hierfür beschreibt Mamella [Sri20] die Verarbeitung von Echtzeitdaten bei Porsche. Dabei wird die Verarbeitung von Streaming-Daten auch im Bereich von Connected Cars eingesetzt. Als Beispiel wird and dieser Stelle der Porsche Taycan aufgeführt, welcher viele Sensoren hat. Die gesammelten Daten, wie Verhaltens-

und Diagnosedaten, können dann in Echtzeit analysiert werden und den Fahrer unterstützen. So kann zum Beispiel analysiert werden, ob der Fahrer müde ist. Die von Porsche entwickelte Anwendung für die Verarbeitung von Streaming-Daten nennt sich Streamzilla und verwendet Apache Kafka.

In seinem Artikel diskutiert Waehler [Wae20a], warum Kafka für Anwendungsbeispiele, wie zum Beispiel Connected Cars, verwendet werden kann. Dazu gehört die Fähigkeit zur Skalierung, welche gut für große Mengen an Daten ist, die kontinuierlich von vielen vernetzten Fahrzeugen generiert wird. Hierfür bietet Confluent als ursprünglicher Entwickler von Kafka eine Erweiterung an, die zusätzliche Funktionen besitzt [Con21].

MQTT und Kafka können auch in Kombination verwendet werden. Erber [Mar21] beschreibt in ihrem Artikel, dass MQTT verwendet werden sollte, weil Kafka-Producer nicht für durch Hardware-Ressourcen beschränkte Geräte, wie IoT-Geräte gedacht sind. Außerdem ist Kafka nicht für instabile Netzwerke ausgelegt und geht davon aus, dass selten neue Verbindungen hergestellt werden müssen. Es gibt trotzdem mehrere Möglichkeiten, um die Nachrichten aus MQTT in Kafka zu erhalten: Die erste Möglichkeit beschreibt die Verwendung von *Kafka Connect for MQTT* für den Nachrichten-Austausch zwischen dem MQTT-Broker und dem Kafka-Cluster. Kafka Connect ist ein Extension Framework, mit welchem andere Systeme an ein Kafka-Cluster angeschlossen werden können. Laut der Autorin wirkt sich dies jedoch negativ auf die Performanz und Skalierung aus. Als zweite Möglichkeit ist die Verwendung eines *MQTT Proxys* aufzuführen. Hier ist kein MQTT-Broker nötig, da dieser die Nachrichten direkt in das Kafka-Cluster streamt. Diese Lösung ist einfach zu skalieren, allerdings werden vom Proxy einige MQTT-Features nicht implementiert. Dies hat Auswirkungen, wie den Verlust von loser Koppelung und möglichem Nachrichtenverlust. Die dritte Möglichkeit ist die Entwicklung einer eigenen Anwendung für den Nachrichtenaustausch zwischen MQTT und Kafka, meist basiert auf vorhandenen Open-Source-Anwendungen. Als letztes ist die Verwendung einer MQTT Broker Extension möglich, um die Nachrichten vom MQTT Broker direkt in das Kafka-Cluster zu schreiben. Hierfür wird jedoch Zugriff auf den MQTT-Broker benötigt. Dieser muss erweiterbar sein, dafür sind alle MQTT-Features erhältlich, wie zum Beispiel Zustellungsgarantien, lose Koppelung, hohe Verfügbarkeit und Performanz. Zusätzlich bietet auch Confluent eine Möglichkeit an, MQTT und Kafka zu kombinieren. Im Bereich Connected Cars können die Sensoren der Fahrzeuge auf diese Weise Daten über MQTT versenden, welche dann in ein Kafka Cluster gespeist werden [Wae20b].

Da Kafka in der Praxis bereits von einigen Firmen, wie beispielsweise Porsche, in der Autoindustrie verwendet wird, soll auch in dieser Arbeit lediglich Kafka genutzt werden.

Modellierung und automatische Bereitstellung von komplexen Systemen

Im nächsten Abschnitt wird die Modellierung von Datenfluss und Pipelines, sowie verschiedener Anwendungen betrachtet. In ihrem Artikel beschreiben Wu et al. [WZX+16] ein Framework namens Pipeline61 für die Erstellung von Pipelines in heterogenen Laufzeitumgebungen. Dabei ist das Ziel des Frameworks den Aufwand für die Wartung und Verwaltung von Pipelines zu reduzieren. Es modelliert die einzelnen Komponenten der Pipeline als Pipes. Diese haben unter anderem einen

Namen, eine Versionsnummer und einen Ausführungskontext, welcher die Ausführungsumgebung und für die Ausführung benötigte Informationen enthält. Auf diese Weise können einfach neue Verarbeitungskomponenten ergänzt werden.

In ihrem Artikel legen Eichelberger et al. [EQS17] einen Ansatz für die Generierung von Big Data Pipelines dar. Dieser ist Modell-basiert und der benötigte Code wird automatisch generiert. Mithilfe einer grafischen Oberfläche können Pipeline-Anwendungen entworfen werden. Das Programm erstellt dann die zur Bereitstellung benötigten Artefakte.

Masek et al. [MTA+16] erläutern in ihrer Arbeit die Evaluation von Sandboxed Software Deployment. Für die Bereitstellung dieser Software auf einem selbstfahrenden Truck wird Docker verwendet. Docker gibt die Möglichkeit ein Skript zu schreiben, welches bei seiner Ausführung die darin beschriebenen Anwendungen bereitstellt [Doc21]. In seiner Forschung zu selbstfahrenden Fahrzeugen beschäftigt sich Berger [Ber16] unter anderem damit, den Aufwand für die Instandhaltung und das Bereitstellen von Software zu reduzieren. Hierfür verwendet er ebenfalls Docker.

Rivera et al. [RVT+18] beschreiben einen weiteren Ansatz für die automatische Bereitstellung von Software. Ihr Ansatz namens URANO verwendet UML nicht nur für die Architektur, sondern auch für das Deployment. Die UML-Diagramme werden so erweitert, dass diese ebenfalls die nötigen Informationen für die Installation, Konfiguration und die Aktualisierung von der beschriebenen Software Komponenten beinhalten.

Auch Ribeiro et al. [RRSM16] verwenden für das automatische Bereitstellen von Cloud-Anwendungen UML-Diagramme. Diese Diagramme enthalten alle Informationen als Parameter, welche für die Bereitstellung nötig sind. Dazu gehören unter anderem, die VMs, Services, Datenbanken und Keys für den Zugriff. Damit ist kein Programmieren notwendig, um eine VM zu erstellen. Für das Deployment werden zwei Modelle benötigt, eines das unabhängig vom Cloud-Provider ist und eines das auf die Eigenheiten des Providers eingeht.

Bergmayr et al. [BBK+16] beschäftigen sich in ihrem Artikel mit der Umwandlung von UML-Architekturen in TOSCA-Modelle. Mithilfe dieses Ansatzes sollen Cloud-Anwendungen automatisch bereit gestellt werden können.

Ein weiterer, ebenfalls auf einem Modell basierender Ansatz, wird von Artač et al. [ABD+16] erläutert und beschäftigt sich mit der Bereitstellung von Cloud-Anwendungen. In diesem Zusammenhang sprechen die Autoren auch von Anwendungen, welche datenintensiv sind. Wird eine Anwendung in ihrer Sprache für die Bereitstellung, MODAClouds4DICE geschrieben (über Drag-and-Drop in einer grafischen Oberfläche) und ausgeführt, so transformiert ihr Framework DICER das Modell in ein TOSCA-Modell, welches dann verwendet wird, um die Anwendung bereit zu stellen.

Wurster et al. [WBK+18] verwenden TOSCA für die Verwaltung und automatische Bereitstellung von serverless Multi-Cloud-Anwendungen. Ein weiterer möglicher Einsatzort für TOSCA ist die Modellierung und Bereitstellung von Quantum Computing-Anwendungen [WBH+20a]. Connected Cars können auch im Kontext von Industrie 4.0 betrachtet werden. In der Industrie 4.0 werden Maschinen nicht nur als Datenquellen gesehen, sie können auch als Aktoren gesehen werden, welche die Produktentwicklung beeinflussen können [FBK+16]. Auch Connected Cars können als Datenquellen betrachtet werden, welche außerdem die Fähigkeit haben etwas zu verändern, wie zum Beispiel das Bremsen, wenn die Sensoren ein Hindernis erkennen. Auch im Bereich Industrie 4.0 kann TOSCA verwendet werden. Eine der Herausforderungen ist es, die in der Industrie gesammelten Daten so nah wie möglich an ihren Quellen zu verarbeiten. Hierfür beschreiben

Falkenthal et al. [FBK+16] eine auf OpenTOSCA basierende Werkzeugkette, welche Apache Flink bereitstellt, was für Analysen verwendet werden kann. Des Weiteren beschreiben Franco da Silva et al. [FHB+17] in ihrem Artikel die Verwendung des Standards TOSCA für die Bereitstellung von Complex Event Processing (CEP)-Anwendungen. CEP wird dabei oft im IoT-Bereich verwendet, um kontinuierlich generierte Daten zu verarbeiten. Franco da Silva et al. [SBH+17] erläutern in einem weiteren Artikel einen Ansatz für das automatische Bereitstellen von IoT-Anwendungen. Hierbei ist die Herausforderung, dass die Hardwarekomponenten unterschiedlich sind. Um die automatische Bereitstellung zu verwirklichen, verwenden sie ebenfalls den Standard TOSCA. Unter Verwendung dessen können Modelle für die Geräte, Middleware und Anwendungen erstellt werden, anhand derer die Teile der Anwendung automatisch installiert werden können. Ein weiterer Artikel erörtert ebenfalls die automatische Bereitstellung von IoT-Anwendungen unter Verwendung des TOSCA Standards und der Laufzeitumgebung OpenTOSCA. Für die Anwendung werden MQTT und ein Mosquitto Message Broker verwendet [SBK+16].

Insgesamt betrachtet wird also zunächst ein Modell geschaffen, welches daraufhin dazu verwendet wird die Anwendungen zu installieren. Dabei kann das Modell, wie im Fall von Docker, ein Skript sein oder wie im Fall von TOSCA (z.B. unter Verwendung von Winery) grafisch. Wie zuvor beschrieben, können viele verschiedene Anwendungen mithilfe von TOSCA modelliert und bereit gestellt werden. Vor allem IoT-Anwendungen, wie es die Data Loop ist, können mit TOSCA modelliert werden. Aus diesem Grund ist TOSCA die ideale Wahl für die Modellierung einer solchen Anwendung und soll somit im Praxisteil dazu verwendet werden die Data Loop zu modellieren und anschließend auch bereit zu stellen, ähnlich wie Franco da Silva et al. [SBH+17] TOSCA in ihrer Arbeit für die Modellierung und das Deployment von IoT-Szenarien verwendet haben.

Für die Implementierung der Komponenten, welche die benötigten Daten liefern, beschreiben Zimmermann et al. [ZBL17] eine Möglichkeit basierend auf TOSCA. Die Herausforderung bei der automatischen Bereitstellung der Komponenten ist die Verknüpfung der verschiedenen Teile über ihre Endpunkte, zum Beispiel URLs. Dieses Programmiermodell soll die Implementierung von miteinander interagierenden Komponenten erleichtern und kann für Cloud- und IoT-Anwendungen verwendet werden. Hierfür wird TOSCA von den Autoren erweitert.

Zusätzlich muss betrachtet werden, ob die Bereitstellung über TOSCA, wie in Abschnitt 2.6 erläutert, imperativ oder deklarativ stattfinden soll. Breitenbücher et al. [BKLW17] betrachten in ihrem Artikel hierfür die Bereitstellung von IoT-Anwendungen. Dabei kommen die Autoren zu dem Schluss, dass die beiden Ansätze gegenseitig ihre Nachteile ausgleichen, weshalb sie einen kombinierten Ansatz vorschlagen. Komplexe Deployments können unter Verwendung von deklarativen Deployment-Modellen nicht modelliert werden. Hierfür kann alternativ zu einem imperativen Ansatz BPMN4TOSCA verwendet werden. Dies soll die Modellierung von Managementplänen vereinfachen, indem BPMN um TOSCA-spezifische Elemente erweitert wird [KBBL12].

Für das Deployment auf Technologien in der Praxis beschreiben Wurster et al. [WBH+20b] TOSCA Light. Hierbei wird keine TOSCA-native Deployment-Engine benötigt.

4 Datenerfassung in Fahrzeugen

Im folgenden Kapitel werden zunächst die von Fahrzeugen gesammelten Daten beschrieben. Anschließend wird diskutiert, für was die Daten nach einer geeigneten Analyse verwendet werden können. Anhand dieses Kapitels findet dann die spätere Implementierung, welche in Kapitel 7 geschildert wird, statt.

4.1 Welche Daten werden gesammelt?

Zunächst stellt sich die Frage, welche Daten von Autos bereits gesammelt werden. Hierzu gibt es eine Untersuchung vom ADAC aus dem Jahr 2020. Hierfür sind zwei Autos untersucht worden, ein Mercedes B-Klasse und ein Renault Zoe. Die B-Klasse überträgt alle zwei Minuten den GPS-Standort an Mercedes, zusätzlich werden ebenfalls andere Daten, wie der Verbrauch und der Kilometerstand übermittelt. Neben den vom Auto ausgeführten Gurtstraffungen oder gefahrenen Kilometern auf verschiedenen Straßen, wie der Autobahn, der Landstraße oder in der Stadt, werden unter anderem auch Informationen über hohe Motordrehzahlen oder Temperaturen versendet. Abgesehen davon wird gemessen, wann die Beleuchtung verwendet und wann die Batterie geladen, beziehungsweise entladen wird. Bei dem untersuchten Renault werden zum Beispiel bei jeder Nutzung oder alle 30 Minuten unter anderem die Uhrzeit, die Position und die Ladung der Batterie versendet. Zusätzlich besteht die Möglichkeit zur Aktivierung der Ferndiagnose. Über diese Funktion kann Renault Daten des Fahrzeugs über den Mobilfunk empfangen [ADA21].

4.2 Wofür werden diese Daten verwendet?

Im Laufe der Arbeit soll eine Data Loop für das Connected Car-Szenario entwickelt werden. Hierbei sollen gemessene Daten gespeichert und verarbeitet werden. Da sich die Daten in einem Auto kontinuierlich verändern und, wie oben erwähnt, in kurzen Abständen gemessen werden, handelt es sich um Echtzeitdaten, welche auch in Echtzeit weiter verarbeitet werden müssen, um den erzeugten DZ immer wieder auf den neuesten Stand bringen zu können. Zusätzlich kann der DZ verändert werden, um diese Änderungen auch in der physischen Welt zu bewirken. Um das Anwendungsszenario für die Data Loop möglichst einfach zu gestalten sollen Daten verwendet werden, die bereits von Sensoren in Fahrzeugen gemessen werden. Eine erste Idee wäre es, die Geschwindigkeit eines Autos zu übermitteln. Diese kann analysiert werden und es kann festgestellt werden, ob der Fahrer zu schnell fährt. Als Ergebnis der Analyse könnte eine Nachricht an das Auto gesendet werden, welche bewirkt, dass ein zu schnell fahrendes Auto seine Geschwindigkeit selbst reduziert. Die GPS-Position kann verwendet werden, um Rückschlüsse auf die aktuellen Straßenbedingungen zu ziehen. So kann beispielsweise die aktuelle Wettersituation an den Koordinaten abgerufen werden. Damit kann das Fahrzeug dann zum Beispiel auf nasse Straßen reagieren. Im Connected-Car Szenario werden oft

auch Daten von außerhalb des Autos gesammelt, zum Beispiel über Kameras [JG17]. Dies soll ebenfalls im Anwendungsszenario aufgegriffen werden. Fotos von der Umgebung sollen die anderen Teilnehmer des Straßenverkehrs zeigen. Hierüber könnte deren relative Position zum eigenen Auto berechnet werden. Durch die Verarbeitung solcher Daten können Abstände zu anderen Fahrzeugen eingehalten und im Notfall auch Bremsmanöver ausgelöst werden.

Der DZ spiegelt in diesem Szenario somit nicht nur das eigene Fahrzeug wider, sondern ebenfalls dessen Reaktion auf die Umwelt.

5 Anforderungen an die Data Loop

Um die Nutzung einer Data Loop möglichst effizient zu gestalten, werden in diesem Kapitel einige Anforderungen zusammengetragen, welche die Data Loop im Kontext von Connected Cars erfüllen sollte.

Die Data Loop ist ein verteiltes System. Ein verteiltes System hat Komponenten, welche sich auf verschiedenen vernetzten Maschinen befinden. Diese kommunizieren miteinander, indem sie Nachrichten austauschen. So können sie ihre Aktionen koordinieren [CDKB12]. Dabei erscheint das System für seine Nutzer als ein Gesamtes [TV07]. Somit ergeben sich bei einer Data Loop die selben Herausforderungen, wie bei anderen verteilten Systemen. Dazu gehören *Skalierbarkeit*, *Sicherheit*, *Heterogenität*, *Fehlerbehandlung*, *Nebenläufigkeit*, *Transparenz* und *Erweiterbarkeit*. Heterogenität bezieht sich unter anderem auf die verwendete Hardware, Programmiersprachen oder Netzwerke. Trotz all dieser Unterschiede soll es möglich sein, dass die Komponenten miteinander interagieren. Da die Informationen, welche das System verwaltet, von großem Wert für den Nutzer sind, müssen diese gesichert werden. Dabei ist einerseits darauf zu achten, dass die Inhalte der Nachrichten geheim bleiben und andererseits, wer die Nachrichten versendet hat. Das System muss skalierbar sein, was bedeutet, dass es weiter effektiv nutzbar ist, selbst wenn sich die Anzahl der Ressourcen oder Nutzer erhöht. Fehlerbehandlung ist in einem verteilten System komplexer, da die Fehler meist nur in einem Teil des Systems auftreten. Des Weiteren muss Nebenläufigkeit gewährleistet sein, da die Ressourcen von mehreren Nutzern verwendet werden können. Zusätzlich soll das System erweiterbar sein. Dies ist möglich, wenn die Dokumentation der wichtigsten Schnittstellen verfügbar ist [CDKB12]. Ein weiterer wichtiger Punkt ist die zuvor erwähnte Transparenz. Dabei soll das verteilte System, welches aus mehreren Komponenten besteht, als ein einzelnes betrachtet werden [CDKB12; TV07].

Die genannten Anforderungen, wie Sicherheit, sind zwar allgemein von größter Wichtigkeit, jedoch liegen diese nicht im Fokus bei der Entwicklung eines Konzeptes für die Data Loop. Im Rahmen dieser Arbeit liegen die folgenden Anforderungen im Fokus:

A.1 Skalierbarkeit: Für die Data Loop ist die Skalierbarkeit von besonderer Wichtigkeit. So beschreibt zum Beispiel Schneider [Sch16] in seinem Artikel die Bedeutsamkeit von Skalierbarkeit im Connected Car Bereich. Er gibt an, dass Komponenten unterschiedlich oft genutzt werden und unterschiedlich viele Ressourcen benötigen. Für Ressourcen-intensive Komponenten muss laut ihm eine effiziente Replizierung möglich sein.

A.2 Verarbeitung und Speicherung heterogener Echtzeitdaten: Da die Daten, wie in Kapitel 4 beschrieben, genauso wie in der Realität sehr unterschiedlich sind, muss die Data Loop es ermöglichen, *heterogene Daten* zu speichern und zu verarbeiten. Dazu gehören in Connected Car-Umgebungen Bilder der Umgebung, aber auch Positionsdaten. Zusätzlich handelt es sich bei den Daten um *Echtzeitdaten*, da diese in der Praxis kontinuierlich generiert und weitergesendet werden.

A.3 Hohe Erweiterbarkeit und Modularisierung: Da sich auch die verwendeten Technologien immer weiter entwickeln, verändern sich möglicherweise die gesammelten Daten und auch die Anforderungen an deren weitere Verarbeitung. Aus diesem Grund ist es notwendig, dass die Data Loop einfach zu erweitern ist. Hierbei ist auch die Forderung von Modularisierbarkeit von Nutzen. Wenn die einzelnen Komponenten in Module unterteilt sind, lassen diese sich beliebig kombinieren und durch neue Module erweitern, wenn für eine geeignete Schnittstelle zur Kommunikation zwischen ihnen gesorgt wird.

A.4 Automatisierbarkeit: Des Weiteren soll die Möglichkeit zur Automatisierung gegeben sein, da die Anwendungen ein komplexes System bilden. Der gesamte Prozess soll ohne menschliches Eingreifen funktionieren. Ebenso soll die Bereitstellung der gesamten Data Loop und aller enthaltenen Komponenten automatisch erfolgen, sodass ein solch komplexes System keine manuelle Installation und Konfiguration benötigt.

A.5 Verwendung von Standards und De-Facto-Standards: Zuletzt sollen für den Entwurf der Data Loop Standards und De-Facto-Standards verwendet werden. Auf diese Weise soll gesichert werden, dass der Prozess, beziehungsweise die Teile des Prozesses, ihren Zweck erfüllen. Durch die Verwendung von Standards soll nicht nur die Wiederverwendbarkeit, sondern auch die Zusammenarbeit aller Komponenten der Data Loop gesichert werden.

6 Konzept und Modellierung einer Data Loop für Connected Car-Umgebungen

Dieses Kapitel beschreibt das in dieser Arbeit entwickelte Konzept der Data Loop. Hierfür wird zunächst allgemein definiert, was die Data Loop ist. Daraufhin werden deren Bestandteile und Architektur besprochen.

Im Bereich Connected Cars kann auf den Begriff IoV gestoßen werden. Dabei bildet eine Flotte aus Fahrzeugen ein Netzwerk aus autonomen Fahrzeugen, welche Sensordaten sammeln [JG17]. Diese tauschen sich untereinander aus, um Unfälle zu verhindern und die Fahrzeuginsassen sicher an ihr Ziel zu bringen [JG17; PKÅ+20]. Diese Entwicklung ist mit der Entstehung des IoT vergleichbar, welches bereits in Abschnitt 2.4 beschrieben ist. Im IoV gibt es ebenfalls Things. Dazu gehören in einer Connected Car-Umgebung interne und externe Sensoren der Fahrzeuge, ebenso wie die Nachrichten aus den Social Media des Fahrers. Zu den gemessenen Daten gehören unter anderem die GPS-Position, die Beschleunigung und der Zustand des Fahrers [JG17].

Die größte Herausforderung in diesem IoT-Szenario ist dabei der Datenaustausch. Fahrzeuge haben unterschiedliche Sensoren und Schnittstellen, wobei die Schnittstellen der verschiedenen Fahrzeuge zusätzlich unterschiedlich, proprietär und meist abhängig vom Fahrzeughersteller sind. Darum ist eine Kollaboration der verschiedenen Hersteller wichtig, um autonom fahrende Fahrzeuge einführen zu können.

Für die Verwirklichung eines solch komplexen IoT-Konzeptes muss modelliert werden, wie die Daten gesammelt, gespeichert und weiterverarbeitet werden. Hierfür ist im Laufe dieser Arbeit das Konzept der Data Loop erarbeitet worden.

Die *Data Loop* ist ein Prozess, in welchem zunächst Daten in der physischen Welt gesammelt und in einem Data Lake gespeichert werden. Anschließend werden diese Daten weiterverarbeitet, wozu die Generierung, beziehungsweise Aktualisierung von DZs gehört. Werden Änderungen am DZ durchgeführt, so sollen diese auch wieder in den physischen Zwilling zurückgeführt werden, womit sich der Kreislauf, also die Loop, schließt. Dieser Prozess ist als Skizze in Abbildung 6.1 abgebildet.

6.1 Bestandteile

In Abbildung 6.2 ist ein detaillierterer Überblick über die Data Loop zu sehen, in welcher mögliche Technologien vermerkt sind. Das in Abschnitt 2.1 beschriebene Modell des DZs soll als Teil der Data Loop modelliert werden. Hierbei werden zunächst Daten gesammelt, welche von den physischen Zwillingen stammen. Diese können entweder vorverarbeitet oder direkt versendet werden. Dabei kann die Vorverarbeitung auf dem Auto selbst, oder nach dem Versenden der Daten und vor dem Eintreffen im Data Lake stattfinden. Hierfür ist ein Messagingsystem sinnvoll, da hierdurch eine

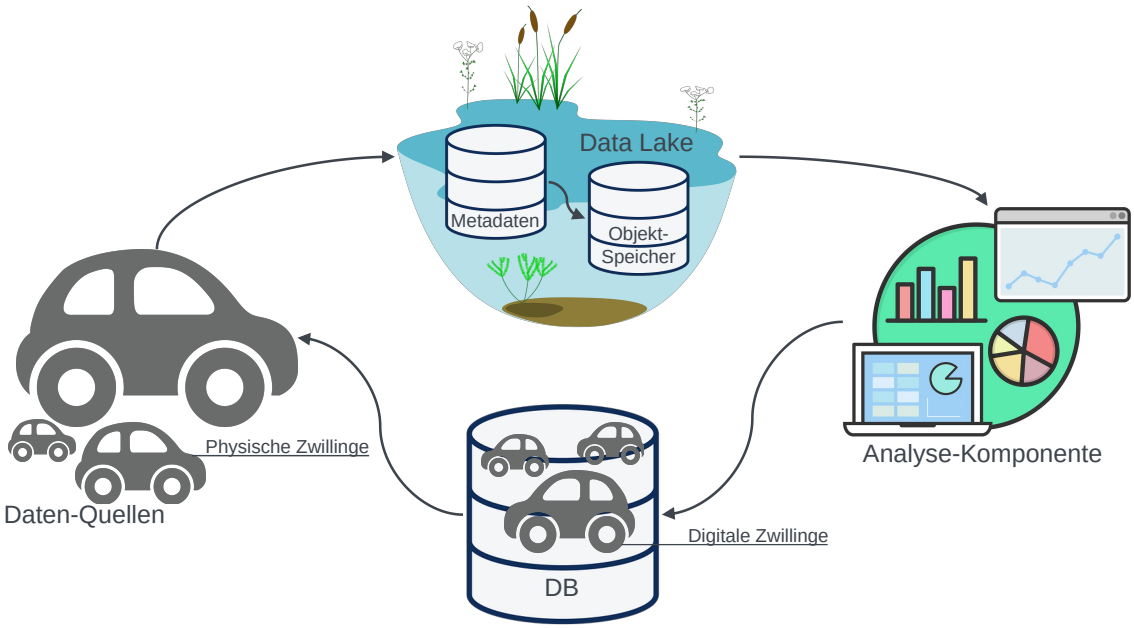


Abbildung 6.1: Allgemeiner Überblick über die Data Loop

lose Koppelung zwischen den Teilen des Systems realisiert werden kann. Außerdem ermöglicht die Verwendung eines Messagingsystems asynchrone Kommunikation. Da die versendeten Daten heterogen sind und in großen Mengen gesammelt werden, sollen diese in ihrer Rohform in einem, wie in Abschnitt 2.2 definierten, Data Lake abgespeichert werden. Dieser verwaltet Metadaten und besitzt verschiedene Objektspeicher, welche dafür sorgen, dass alle gesammelten Daten abgespeichert werden können. Die Wahl verschiedener Datenbanktechnologien, wie die im Diagramm aufgelisteten, relationale Datenbanken, dokumentbasierte Datenbanken, Zeitreihendatenbanken und Schlüssel-Werte-Datenbanken ermöglichen die Speicherung der unterschiedlichen Daten. Anschließend werden die im Data Lake abgelegten Rohdaten analysiert und es werden DZs generiert, beziehungsweise entsprechend der neu eingetroffenen Daten aktualisiert. Hierfür gibt es sehr viele Analyse-Möglichkeiten. Dazu gehört Machine Learning, welches zum Beispiel für die Bilderkennung verwendet werden kann. Zusätzlich spiegeln sich Veränderungen am DZ auch in der realen Welt, also dem physischen Zwilling wider.

6.2 Auswahl der Architektur

Für die Modellierung eines solch komplexen IoT-Szenarios muss eine geeignete Architektur gewählt werden. Hierfür sind in Abschnitt 2.5 verschiedene Möglichkeiten aufgezeigt worden, wie IoT und die Cloud miteinander integriert werden können. Diese aufgezählten Möglichkeiten sind Edge und Fog Computing. Beim Vergleich zwischen Fog und Edge Computing, wirken diese auf den ersten Blick sehr ähnlich. Der Unterschied liegt bei den durch die jeweiligen Geräte zur Verfügung gestellten Ressourcen. So befinden sich auf der Fog-Ebene Server, welche Cloud-ähnliche Ressourcen bereitstellen [LGL+15]. Die Edge Devices hingegen haben nur eine begrenzte Menge an Ressourcen [DB16]. Da der DZ viele Informationen und auch große Kapazitäten für die Verarbeitung benötigt, ist für die weitere Arbeit die Fog-Architektur verwendet worden [Gri17]. Die

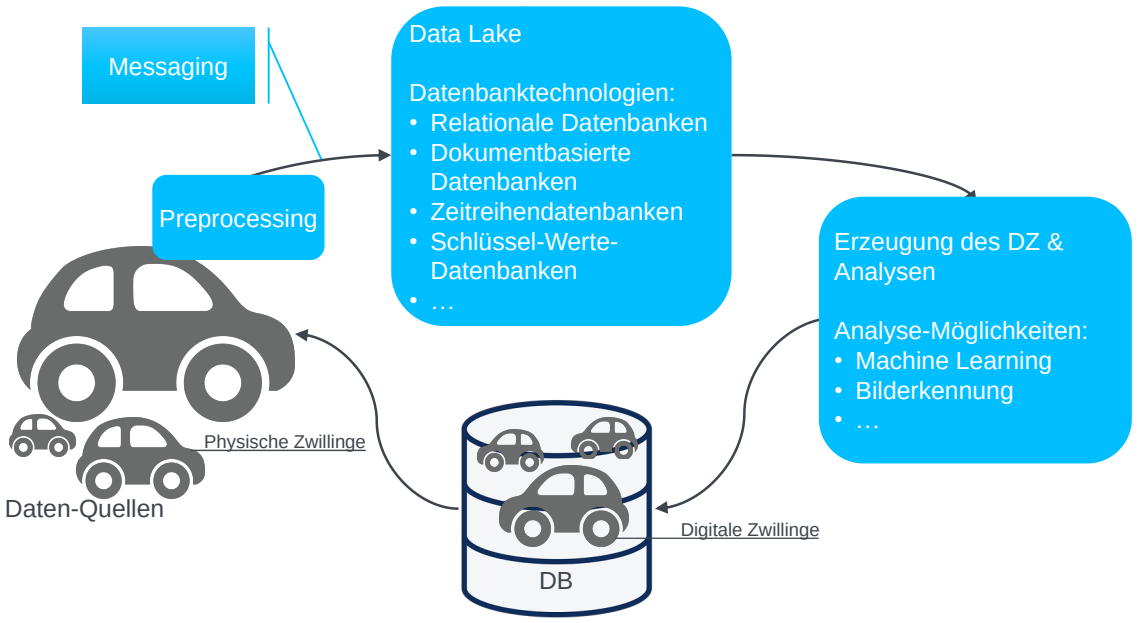


Abbildung 6.2: Detaillierter Überblick über die Data Loop

Fog-Server können dann zum Beispiel für die Vorverarbeitung der Daten verwendet werden, wie eine Filterung vor dem Abspeichern der Daten im Data Lake. Des Weiteren ist es möglich, das Messagingsystem für die Kommunikation zwischen den verschiedenen Teilen der Data Loop, auf Fog-Servern bereitzustellen.

Xiao et al. [Xia17] beschreiben eine Möglichkeit für die Umsetzung einer Fog-Architektur im Kontext von Connected Cars. Dabei schlagen die Autoren vor, auf bestimmten Fahrzeugen, wie selbstfahrenden Bussen und Taxis, Fog-Knoten zu installieren, welche *Vehicular Fog Nodes* genannt werden. Diese sollen zum Beispiel die Sensordaten lokal verarbeiten. Sind sie überlastet, so gibt es zusätzlich *Cellular Fog Nodes*, welche am Rand des Mobilfunknetzwerks platziert werden. Jene können im Falle einer Überlastung die Verarbeitung übernehmen.

Ein weiterer Ansatz wird von Huang et al. [HLC17] beschrieben. Hier werden von den Fahrzeugen Daten gesammelt und bereits vorverarbeitet. Für Analysen, die über das Fahrzeug hinaus gehen, werden die Daten an Fog-Server weitergegeben. Diese werden in diesem Ansatz am Straßenrand positioniert. Die Fog-Server sollen die Daten der verschiedenen Fahrzeuge zunächst sammeln und dann verarbeiten, bevor sie diese an die Cloud weiter senden. Zusätzlich sollen die Fog-Server als Middleware für die Kommunikation zwischen den Fahrzeugen und zwischen den Cloud-Servern dienen. Außerdem sollen die Fog-Server bereits verschiedene Services bieten, wie zum Beispiel die Navigation. Rechenintensive Analysen, wie die Kontrolle des Verkehrs, werden von der Cloud durchgeführt.

6.3 Modellierung der Data Loop

Des Weiteren muss die Data Loop modelliert werden. Hierfür muss ein geeignetes Modellierungstool oder ein Modellierungsstandard gewählt werden. In Kapitel 3 sind bereits verschiedene Möglichkeiten für das Lösen dieser Aufgabe betrachtet worden. Dabei ist das Vorgehen allgemein wie folgt: Zunächst wird ein Modell der Anwendung erstellt, welches anschließend dafür verwendet wird die Anwendungen zu installieren und auszuführen. Dabei kann das Modell ein Skript oder eine grafische Repräsentation der Anwendung sein.

Docker ist ein Beispiel für die Erstellung eines Skriptes als Modell [Doc21]. Masek et al. und Berger verwenden dies für ihre Arbeiten mit selbstfahrenden Fahrzeugen [Ber16; MTA+16]. Für die Modellierung von Datenfluss und Pipelines gibt es zum Beispiel Pipeline61 und einen weiteren Ansatz von Eichelberger et al., beides sind grafische Ansätze [EQS17; WZX+16]. Eine weitere Möglichkeit ist die Erweiterung von UML, damit die erstellten Modelle automatisch Bereitgestellt werden können [BBK+16; RRSM16; RVT+18]. Ein Standard für die Modellierung und das automatische Bereitstellen von Cloud-Anwendungen ist TOSCA [BBKL14]. Wurster et al. [WBK+18] verwenden den Standard für die Verwaltung und automatische Bereitstellung von serverless Multi-Cloud-Anwendungen. Wild et al. [WBH+20a] verwenden TOSCA für die Modellierung von Quantum Computing Anwendungen. Des Weiteren findet TOSCA auch im Bereich Industrie 4.0 Verwendung [FBK+16]. Auch für die Bereitstellung von Complex Event Processing (CEP)-Anwendungen, welche im Bereich IoT eine Rolle spielen, wird der Standard verwendet [FHB+17]. Passend dazu werden auch IoT-Anwendungen mithilfe von TOSCA modelliert und automatisch bereitgestellt [SBH+17; SBK+16].

Somit können unter Verwendung von TOSCA viele verschiedene Anwendungen modelliert werden. Connected Cars können als Datenquellen betrachtet werden, welche ebenfalls Auswirkungen auf die Umwelt nehmen können, zum Beispiel durch Bremsen. Darum kann dies auch als ein Industrie 4.0-Szenario betrachtet werden, was wie zuvor beschrieben mithilfe von TOSCA modelliert werden kann. Des Weiteren gehören auch IoT-Anwendungen zu den modellierbaren Systemen, worum es sich im Falle der Data Loop handelt. Aus diesem Grund macht es Sinn, die Data Loop mithilfe des TOSCA-Standards zu modellieren und anschließend bereit zu stellen.

Zusätzlich bietet TOSCA noch weitere Vorteile, welche diese Auswahl unterstützen. Zum einen ist TOSCA, wie bereits erwähnt, ein Standard. Die Verwendung von TOSCA würde also helfen Anforderung A.5 zu erfüllen. Ein weiterer Vorteil ist, dass es durch Anwendungen, wie zum Beispiel Winery, möglich ist, Modelle über eine grafische Oberfläche zu erstellen. Darüber hinaus gibt es Open-Source Implementierungen des Standards, wie OpenTOSCA, welche bereits in Abschnitt 2.6.2 vorgestellt worden ist [BEK+16].

7 Prototypische Implementierung des Data Loop-Konzepts für Connected Car-Umgebungen

Dieser Teil beschäftigt sich mit der Implementierung eines Prototyps anhand der Beschreibung der Daten und der Modellierung der Data Loop in TOSCA. Im Laufe dieses Kapitels wird erst die prototypische Implementierung des in Kapitel 6 beschriebenen Konzepts erläutert. Hierbei folgt zunächst die Erläuterung der Auswahl der verwendeten Technologien. Anschließend werden der minimale Prototyp und seine Architektur beschrieben. Zu guter Letzt werden die für das automatische Deployment, mit TOSCA, benötigten Service Templates und Node Types geschildert.

7.1 Auswahl der verwendeten Technologien

Ein wichtiger Teil der Data Loop ist der Data Lake. Für die Implementierung eines Data Lakes werden verschiedene Datenbanken benötigt, darum wird im Folgenden zunächst die Wahl der Datenbanken für das Szenario begründet. Anschließend folgt eine Erklärung des verwendeten Messagingsystems Kafka[1].

7.1.1 Datenbanken

Der Data Lake soll für das Szenario möglichst minimal gehalten werden. Aus diesem Grund wird zunächst eine Datenbank für den Metadatenspeicher ausgewählt. Für die Speicherung der Daten werden weitere Datenbanken benötigt, welche anhand der zu speichernden Daten ausgewählt werden. Für den Metadatenspeicher wird eine MySQL-Datenbank[2] verwendet. Es handelt sich um eine relationale Datenbank und besteht somit aus Tabellen, auch Relationen genannt. Die Datensätze werden in den Zeilen der Tabellen gespeichert [Sch14]. Für den Prototyp und die lokale Installation fällt die Wahl auf den MySQL Communitiy Server. Dieser kann jedoch durch die entsprechende Cluster-Version ersetzt werden, um eine höhere Verfügbarkeit zu erzielen. Die Architektur dieser ist verteilt und skaliert horizontal [Ora21]. Da die Metadaten im ausgewählten Anwendungsfall, beispielsweise die Fahrgestellnummer, einen Zeitstempel, die Marke und das Automodell enthalten, lassen sich diese auf strukturierte Weise darstellen. Dies legt nahe, eine relationale Datenbank, wie MySQL, zu verwenden, da relationale Datenbanken hauptsächlich strukturierte Daten verarbeiten [MK16]. Für die Speicherung der Datensätze wird, wie zuvor bereits erläutert, eine Auswahl an verschiedenen

[1] Zu finden unter: https://kafka.apache.org/
[2] Zu finden unter: https://www.mysql.com/de/

Datenbanken verwendet. Eine davon ist die MongoDB[3]. Diese ist eine verteilte dokumentbasierte Datenbank. In Dokumentdatenbanken werden Daten als ein Schlüssel-Wert-Paar abgespeichert. Dabei besteht der Wert aus einem Dokument, welches wiederum Schlüssel-Wert-Paare oder auch eingebettete Dokumente enthalten kann. MongoDB speichert Daten in einem JSON-ähnlichen Format, namens Binary JSON (BSON), ab. Die Hersteller beschreiben, dass dies die Datenbank leistungsfähiger macht als Datenbanken, welche auf Tabellen basiert sind. Die dazugehörige Abfragesprache verwendet JSON-Queries. Diese unterstützt unter anderem die Filterung, Sortierung und Aggregation von Daten. In diesem Fall wird die Community-Version der MongoDB genutzt. Im Gegensatz zur Enterprise-Version ist diese nicht Teil eines Abonnements. Die Community-Version bietet, genauso wie die Enterprise-Version, hohe Verfügbarkeit, da sie unter anderem eine eingebaute Replikation verwendet und horizontal skaliert [Mon21]. Aufgrund der Skalierbarkeit und der hohen Verfügbarkeit dieser Datenbank, ist sie eine gute Wahl für das Anwendungsszenario, da dieses ebenfalls eine hohe Verfügbarkeit fordert. Darüber hinaus werden später in der Implementierung bereits JSON-Dokumente als Nachrichten versendet, was zur internen Darstellung der Daten in der MongoDB passt. Da viele der gesammelten Daten im Anwendungsszenario auf einer Zeit basieren, wie zum Beispiel die aktuell gemessene Geschwindigkeit des Autos, macht es Sinn, ebenfalls eine Time-Series-Datenbank zu verwenden. Bader et al. [BKF17] definieren Time-Series-Datenbanken als ein Datenbank Management System, welches Reihen von Daten speichern kann, die aus einem Timestamp, einem Wert und möglicherweise weiteren Tags besteht. Zusätzlich können die Reihen gruppiert und abgefragt werden. Bei der Abfrage können Daten dann auch nach ihrem Timestamp oder einer Zeitspanne abgefragt werden. InfluxDB[4] ist ein Beispiel für eine solche Datenbank und ist somit Auswahlmöglichkeit in Betracht gezogen worden. Die kostenlose Open-Source-Version von InfluxDB ist jedoch nicht horizontal skalierbar und hat damit keine hohe Verfügbarkeit. Im Gegensatz dazu, kann Apache Druid[5] als Cluster bereitgestellt werden, welches skalierbar und fehlertolerant ist. Apache Druid ist eine analytische Datenbank und verwendet Ideen aus verschiedenen Bereichen. Dazu gehören Data Warehousing, Time-Series-Datenbanken und auch Suchsysteme. Zum Speichern der Daten wird ein Column-oriented Storage verwendet. Wie MongoDB hat Druid eine auf JSON basierte native Abfragesprache, wobei durch Druid SQL auch SQL-Abfragen unterstützt werden. Die native Abfragesprache unterstützt Aggregation und auch sogenannte Timeseries-Abfragen, anhand derer die Datensätze mithilfe eines Zeitintervalls abgefragt werden können. Dies macht Druid zu einer guten Alternative zur InfluxDB, da auch hier ein Timestamp für jeden Datensatz angegeben werden muss [Apa21a]. Wie auch bei dem MySQL-Server ist für den minimalen Prototypen aus Einfachheit kein Druid-Cluster aufgesetzt worden.

7.1.2 Messaging

Wie im Anwendungsfall beschrieben, müssen Nachrichten in Echtzeit zwischen den Komponenten versendet werden. Dazu gehören zum Beispiel die aktuellen Standort-Koordinaten eines Autos, welche sich während einer Fahrt ständig ändern. Hierfür kann eine Event-Streaming-Plattform wie Apache Kafka verwendet werden, welches als Messagingsystem eingesetzt werden kann. Kafka ist ein verteiltes und skalierbares System. Wie die Bezeichnung Messagingsystem beschreibt, gibt es die Möglichkeit Nachrichten zu senden. Abbildung 7.1 gibt einen Überblick über Kafka und

[3] Zu finden unter: https://www.mongodb.com/de-de
[4] Zu finden unter: https://www.influxdata.com/products/influxdb/
[5] Zu finden unter: https://druid.apache.org/

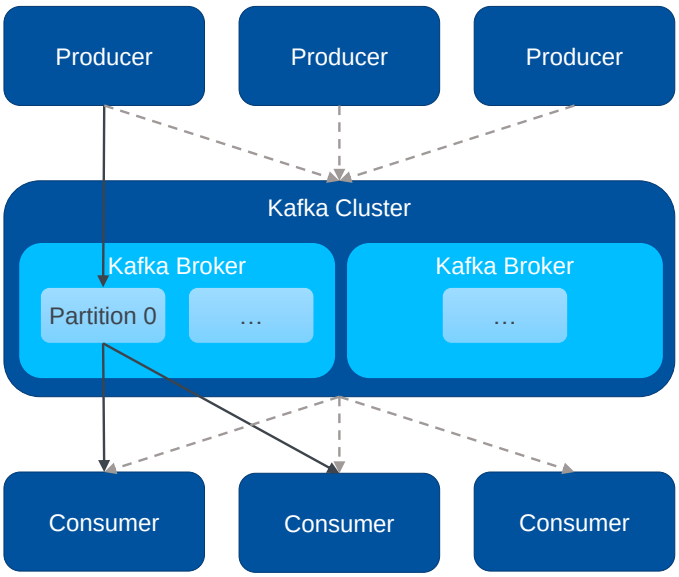


Abbildung 7.1: Grober Überblick über Kafka und dessen Komponenten (nach Beschreibungen unter: [Apa21b])

seine Komponenten. Die im Schaubild zu erkennenden *Producer* sind Prozesse, welche Nachrichten an sogenannte *Topics* senden. Die gesendeten Nachrichten sind in der Abbildung durch die gestrichelten Pfeile symbolisiert. Ein Topic ist auf verschiedene *Partitionen* aufgeteilt, die ein Protokoll bilden, an welches die Nachrichten angehängt werden. Diese werden dann für einen vom Nutzer spezifizierten Zeitraum aufgehoben. Partitionen befinden sich auf Servern, welche *Broker* genannt werden. Ein oder mehrere solche Server bilden, wie in Abbildung 7.1 abgebildet, ein Kafka-Cluster. *Consumer* wiederum sind Prozesse, welche eines oder auch mehrere Topics abonnieren. So können diese Nachrichten, welche an eines ihrer abonnierten Topics gesendet worden sind, empfangen und verarbeiten. Die schwarzen Pfeile zeigen beispielhaft, wie ein Producer eine Nachricht an Partition 0 eines Topics sendet. Diese Nachricht wird dann von zwei Consumern empfangen und verarbeitet [Apa21b].

7.2 Erster minimaler Prototyp

Um einen besseren Überblick über alle Konzepte zu gewinnen, welche innerhalb der Data Loop verwendet werden sollen, ist zunächst ein erster Entwurf für die Beispielanwendung erstellt worden. Im Szenario des minimalen Beispiels werden zu Beginn Geschwindigkeitsdaten gesammelt. Diese gehören zu einem bestimmten Auto, welches eine Fahrgestellnummer, eine Marke und ein Modell hat. Zusätzlich wird ebenfalls die Einheit der Geschwindigkeitsmessung aufgenommen, sowie der Zeitpunkt der Messung. Diese Daten werden in einem Data Lake gespeichert und anschließend verarbeitet. Dabei wird für jedes Auto die durchschnittliche Geschwindigkeit errechnet. Das Resultat wird zusammen mit den allgemeinen Daten über das jeweilige Auto als DZ in einer weiteren Datenbank abgelegt. Anschließend findet eine weitere Analyse anhand der Daten des DZ statt, welche eine Nachricht ausgibt, ob das Fahrzeug im Durchschnitt zu schnell, also schneller als 50 km/h, fährt. Das zuvor beschriebene Szenario ist zunächst lokal umgesetzt worden. Für die Umsetzung

Listing 7.1 Beispiel: Generierte Fahrzeugdaten als JSON-Objekt

```
{
  "chassisNr" : "WDB2202561A123456",
  "make" : "Mercedes",
  "model" : "S 600",
  "measurement" : "km/h",
  "averageVelocity : 50
}
```

sind verschiedene Java-Anwendungen entstanden. Abbildung 7.2 stellt den ersten Entwurf der Data Loop dar. Hierfür befindet sich die Implementierung, zur Vereinfachung, in einem einzigen Java-Projekt. Zunächst gibt es einen Fahrzeugdaten-Simulator. Dieser erzeugt alle zwei Minuten Geschwindigkeitsdaten. Die Daten werden in Form eines JSON-Objektes generiert, wofür ein Beispiel in Listing 7.1 zu sehen ist. Dieses Objekt wird vom im Schaubild 7.2 dargestellten Data Lake Handler in den verschiedenen Teilen des Data Lakes gespeichert. Der Data Lake besteht in diesem Fall aus zwei Datenbanken: Zum einen einer MySQL, für den Metadatenspeicher und eine MongoDB für die Objekte. Näheres zur Begründung der Auswahl der Datenbanken findet sich in Abschnitt 7.1.1. Zu den Metadaten gehören die Fahrgestellnummer, die Marke, das Modell, die Einheit der Geschwindigkeit und der Zeitstempel. In der MySQL sind dafür zwei Tabellen angelegt worden. In der ersten Tabelle Car, befinden sich die allgemeinen Informationen über das Auto, also die Fahrgestellnummer, die Marke und das Modell. Die Fahrgestellnummer wird hierbei als Primärschlüssel verwendet. In der zweiten Tabelle befinden sich die Fahrgestellnummer als Fremdschlüssel, der Zeitstempel, die Einheit und als Primärschlüssel eine Objekt-Id. Unter dieser Objekt-Id ist dann in der MongoDB ein JSON-Objekt gespeichert, welches die Geschwindigkeit des Fahrzeugs enthält. Die Analyse-Komponente fordert über den Metadatenspeicher die benötigten Objekt-Ids an, um auf die Geschwindigkeiten aus der MongoDB zugreifen zu können. Anschließend werden diese miteinander verrechnet und es wird die durchschnittliche Geschwindigkeit des Autos kalkuliert, welche zusammen mit der Fahrgestellnummer, ebenfalls als JSON-Objekt, in einer weiteren MongoDB abgespeichert wird. Die Datensätze werden in der Datenbank `digitaltwin` unter der Collection `twins` eingetragen. Zuletzt werden die Daten der DZs von einer weiteren Analyse-Komponente verwendet, im Diagramm 7.2 als Datensenke verbildlicht, welche auf der Konsole ausgibt, ob die in der DZ-Datenbank enthaltenen Fahrzeuge schneller als 50 km/h, also im Durchschnitt „zu schnell“ fahren.

7.3 Erweiterter minimaler Prototyp

Im nächsten Schritt ist der Prototyp unter anderem um eine Messaging-Komponente erweitert worden. Dies ist in Abbildung 7.3 abgebildet. Nun werden die vom Fahrzeug-Simulator erzeugten Daten nicht mehr direkt an den Data Lake Handler übergeben, sondern zunächst von einem Kafka-Producer an ein Kafka-Topic namens `carDataTopic` übersandt. Diese Nachrichten können danach mithilfe eines Kafka-Consumers vom Data Lake Handler aus dem Topic ausgelesen werden. Daraufhin werden die Daten, wie zuvor beschrieben, abgespeichert und weiterverarbeitet. Zusätzlich zu den bisher für den Data Lake verwendeten Datenbanken, kommt nun auch eine Time Series Da-

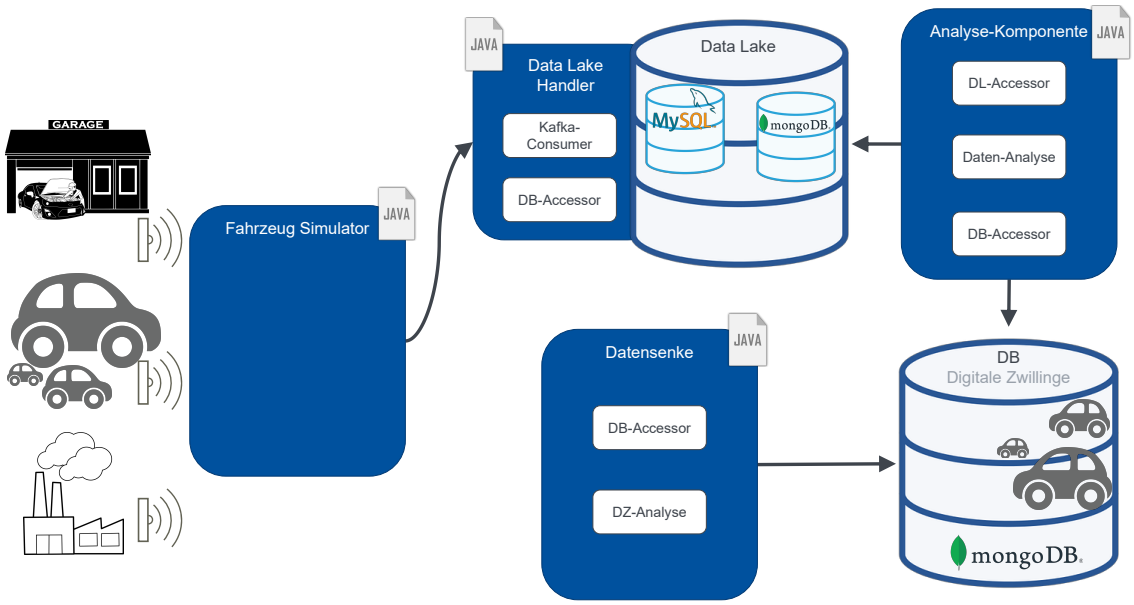


Abbildung 7.2: Architektur des ersten Entwurfs für die Data Loop

tenbank zum Einsatz, da die Geschwindigkeitsdaten immer in Verbindung zu einem Datum stehen. Hierfür sind, wie auf dem Schaubild (Abbildung 7.3) erkennbar, zwei verschiedene Datenbanken getestet worden: InfluxDB und Apache Druid. Wie in Abschnitt 7.1.1 beschrieben, fällt die Wahl zwischen den beiden Time Series Datenbanken auf Apache Druid, da InfluxDB nur in der bezahlten Version skalierbar ist. Zuletzt soll die Data Loop geschlossen werden, indem das Resultat aus der Datensenke an das Auto zurück gegeben wird. Aus diesem Grund sendet ein Kafka-Producer eine Nachricht an das dafür bestimmte `digitalTwinTopic`. Je nach durchschnittlicher Geschwindigkeit wird entweder bei eingehaltener Geschwindigkeitsbegrenzung „On average, you are driving well! - [averageVelocity <= 50]“ oder wenn das Auto im Durchschnitt zu schnell fährt: „On average, you are driving too fast! [averageVelocity > 50]“ als Nachricht an das Kafka-Topic übergeben. Damit soll simuliert werden, dass das Auto anhand einer solchen Nachricht seine Geschwindigkeit anpassen könnte.

7.4 Resultierender Prototyp

Die ersten Entwürfe implementieren eine unidirektionale Kommunikation, bei welcher der benötigte Nachrichtenaustausch in Richtung der Autos, beziehungsweise der letzte Schritt, dass Änderungen im digitalen auch Auswirkungen auf den physischen Zwilling haben sollen, außen vor gelassen wurde. Um eine bidirektionale Kommunikation zu ermöglichen, wird nun SuperTuxKart[6] verwendet. SuperTuxKart ist ein Open-Source Rennspiel, welches für verschiedene Plattformen, wie Windows, Mac und auch Linux, erhältlich ist. Es kann aus zwei verschiedenen Spielmodi gewählt werden. Neben dem Story Modus, können auch Rennen gegen andere Spieler oder den Computer gefahren

[6]Zu finden unter: `https://supertuxkart.net/Main_Page`

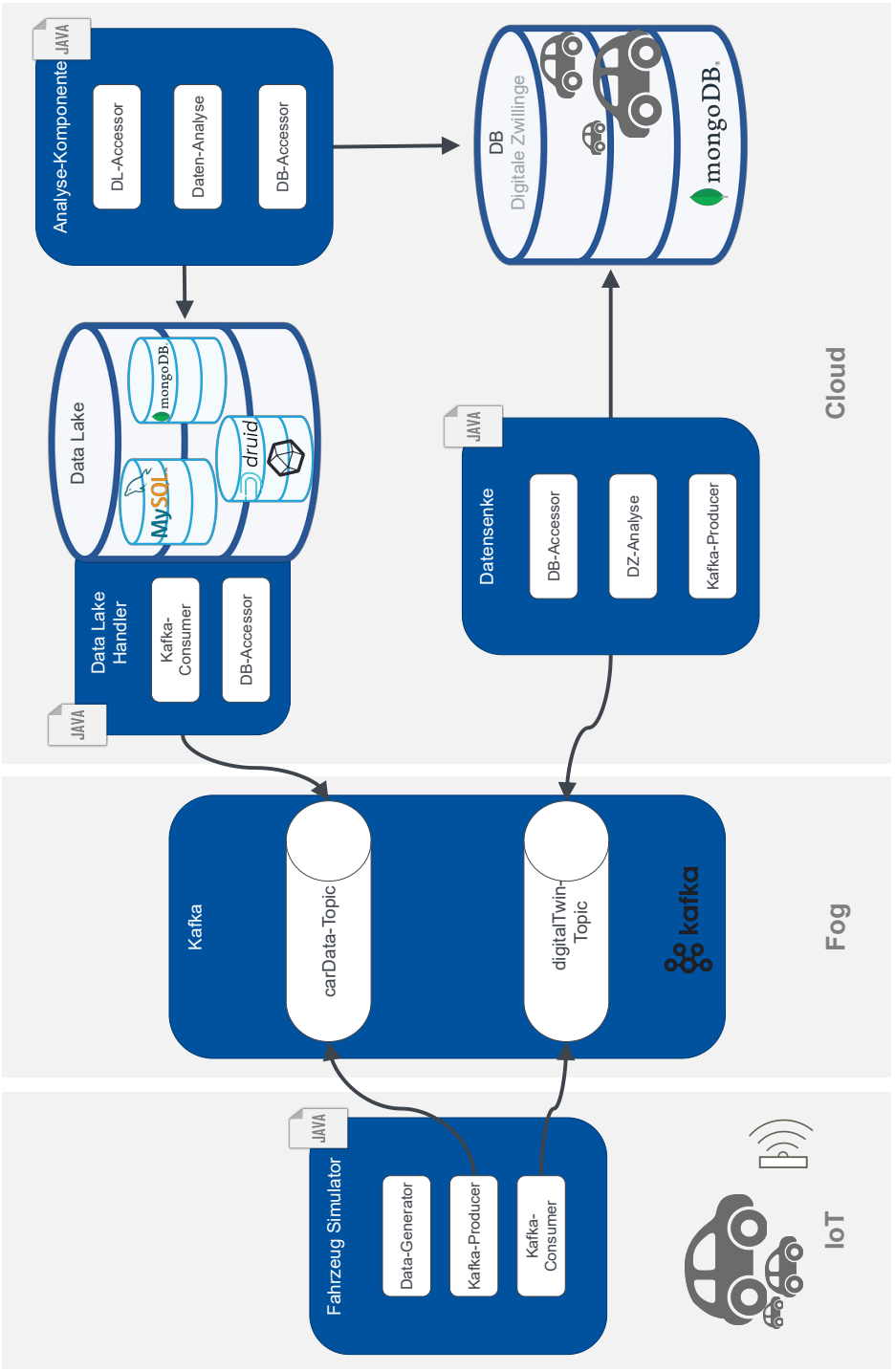


Abbildung 7.3: Architektur des erweiterten Entwurfs für die Data Loop

Abbildung 7.4: Screenshot von einem Rennstart in SuperTuxKart

werden. Im Spiel selbst ist es möglich, sich seinen Charakter aus einer Liste von Karts auszuwählen. Dazu gehört zum Beispiel Tux, das Linux-Maskottchen. Außerdem können verschiedene Rennstrecken ausgewählt werden [Sup21]. Beim Starten eines Spiels fällt auf, dass es ebenfalls die Möglichkeit gibt zusätzliche, von anderen Nutzern erstellte, Addons herunterzuladen. Dazu gehören auch Karts und Rennstrecken (Tracks). In Abbildung 7.4 ist ein Screenshot aus dem Spiel zu sehen. Es handelt sich dabei um den Rennstart auf der Strecke Hacienda im Einzelspieler-Modus. Das letzte zu sehende Kart, mit dem Linux-Maskottchen Tux, wird dabei vom Spieler gesteuert. SuperTuxKart soll nun verwendet werden, um den Physischen Zwilling darzustellen, welcher auf die Änderungen des DZ reagiert. Da hierfür keine Modifizierungen am Code von SuperTuxKart durchgeführt werden sollen, ist betrachtet worden, welche Konfigurationsdateien vor dem Start der Applikation bearbeitet werden können und was durch eine Modifikation dieser beeinflusst werden kann. Die Konfiguration wird bei Windows unter `<APPDATA>/supertuxkart/config-0.10` gespeichtert [Sup21]. Hier sind unter anderem einige XML-Dateien zu finden, durch deren Modifikation Einfluss auf das Spiel genommen werden kann. Dazu gehört `players.xml`, aus welcher ein Ausschnitt in Listing 7.2 zu sehen ist. Darin ist erkennbar, dass diese Datei Daten zu den Spielern enthält. Des Weiteren finden sich hier Daten zum Story-Modus Fortschritt und zu den Errungenschaften, welche aber für die weitere Arbeit nicht relevant sind. Eine Änderung, die mithilfe dieser Datei vorgenommen werden kann, ist die Bestimmung der Standard-Farbe der Karts, indem der Wert von `doc("players.xml")/players/player/@default-kart-color` angepasst wird. Der Wert beinhaltet eine Zahl von 0 bis 1, wobei 0 bedeutet, dass die Originalfarbe des Karts verwendet wird. Die anderen Zahlen stellen Farben dar, 0.5 ist zum Beispiel türkis und 1 repräsentiert rot. Diese Einstellung wirkt sich jedoch nicht auf alle Karts aus, einige Kart-Addons werden von ihr nicht beeinflusst. Eine weitere Konfigurationsdatei, in diesem Ordner, ist `config.xml`. Hier können sehr viele Änderungen vorgenommen werden. Einen Ausschnitt dieser Datei kann in Listing 7.3 betrachtet werden. Das erste abgebildete Beispiel `doc("config.xml")/stkconfig/RaceSetup` zeigt die Möglichkeit über die Attribute `numkarts` und `numlaps` die Anzahl der verwendeten Karts und

Listing 7.2 SuperTuxKart-Konfiguration: players.xml

```
<?xml version="1.0"?>
<players version="1" >
  <current player="Katharina"/>
  <player name="Katharina" guest="false" use-frequency="30"
    icon-filename="1.png"
    unique-id="1" saved-session="false"
    saved-user="0" saved-token=""
    last-online-name="" last-was-online="false"
    remember-password="false"
    default-kart-color="0">
    ...
  </player>
</players>
```

der zu fahrenden Runden anzupassen. Des Weiteren kann, wie im Ausschnitt in Listing 7.3 zu sehen, das derzeit verwendete Kart unter `doc("config.xml/stkconfig/kart/@value)` angepasst werden. Es besteht zum Beispiel auch die Möglichkeit den Fahrern zu Weihnachten oder auch immer Nikolausmützen aufzusetzen oder, passend zu Ostern, Hasenohren. Diese Modifikationen können unter `doc("config.xml/stkconfig/GFX/christmas-mode||easter-ear-mode)` vorgenommen werden. Wird `doc("config.xml/stkconfig/unlock_everything/@value)` auf den Wert 2 gesetzt, werden alle Karts und Rennstrecken frei geschaltet, ohne den Story-Modus spielen zu müssen. Bei Betrachtung des Codes[7], ist in der `main.cpp`, ein Überblick darüber zu finden, mit welchen Optionen SuperTuxKart über die Konsole aufgerufen werden kann. Neben vielen der in `config.xml` anpassbaren Eigenschaften, lassen sich noch einige weitere über einen Konsolenaufruf unter der Verwendung einiger Optionen anpassen. In Tabelle 7.1 ist eine Auswahl verfügbarer Optionen dargestellt. Ein Mögliches Beispiel für einen Konsolenaufruf ist: `supertuxkart.exe -N --track=hacienda --laps=1 --profile-laps=1 --aiNP=emule,tux`. Hierbei wird durch die Option `-N` das Rennen sofort gestartet, ohne zuvor in das Menü zu gehen. Zusätzlich wird die Strecke `Hacienda` gewählt und es wird eine Runde gefahren, welche durch die Option `--profile-laps=1` durch den Computer gesteuert wird. Das Rennen findet dabei zwischen den zwei Karts `Tux` und `Emule` statt.

Somit kann über die Konfigurationsdateien und die Konsole wenig Einfluss auf das eigentliche Auto genommen werden. Die Anzahl der mitfahrenden Autos kann jedoch bestimmt werden und darüber hinaus sogar, welche dies im genauen sind. Damit kann dann festlegt werden, welche Farben die umliegenden Autos haben. Außerdem kann über die Wahl der Strecke auch das Wetter und die Tageszeit beeinflusst werden, welche dann nach dem Start eines Rennens zu sehen sind. So ist es zum Beispiel auf der Strecke `Hacienda` Tag und die Sonne scheint, auf der Strecke `Snowmountain` ist es Nacht und es schneit. Diese zwei Möglichkeiten sollen im weiteren Verlauf der Arbeit Änderungen am Physischen Zwilling darstellen, welche durch die Modifikation des DZ hervorgerufen werden. Als Folge dessen stellt SuperTuxKart nun einen Teil des Physischen Zwillings dar.

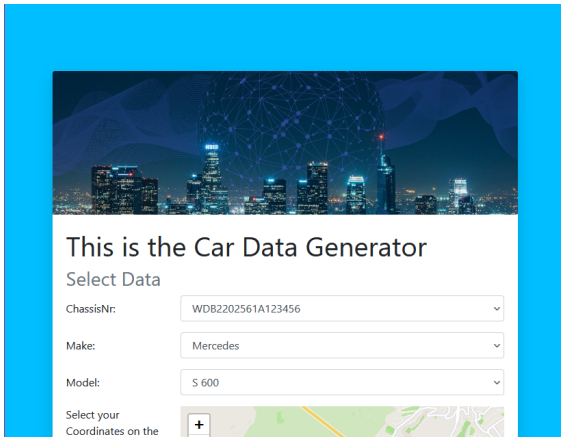
[7] Zu finden unter: https://github.com/supertuxkart/stk-code/

Listing 7.3 SuperTuxKart-Konfiguration: config.xml

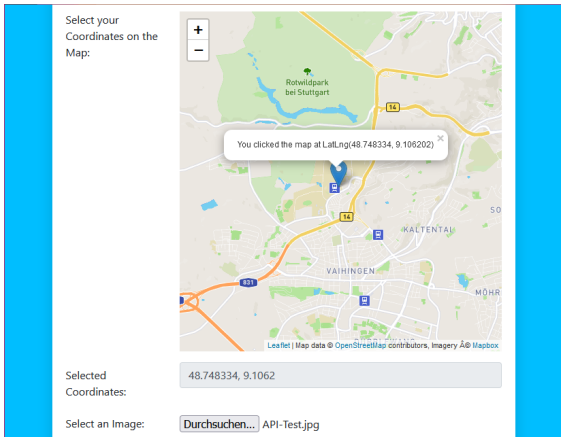
```
<?xml version="1.0"?>
<stkconfig version="8" >
  ...
  <RaceSetup
    numkarts="6"
    numlaps="1"
    ...
  >
  </RaceSetup>
  ...
  <kart value="tux" />
  ...
  <GFX
    ...
    christmas-mode="1"
    easter-ear-mode="0"
    ...
  >
  </GFX>
  ...
  <unlock_everything value="2" />
  ...
</stkconfig>
```

Option	Beschreibung
-N	Startet sofort ein Rennen (überspringt das Menü)
-t, --track=<TrackName>	Verwendet die angegebene Strecke beim Start
-k, --numkarts=<Anzahl>	Verwendet die angegebene Anzahl an Karts im Rennen
--kart=<KartName>	Verwendet angegebenes Kart für Spieler
--aiNP=<ListeKarts>	Verwendet die angegebene komma-separierte Liste an Karts für AI-Fahrer; Dabei ist kein Kart für einen Spieler aufgezählt
--laps=<Anzahl>	Gibt an wie viele Runden zu fahren sind
--profile-laps=<Anzahl>	Gibt an, wie viele Runden automatisch, also ohne Steuerung eines Karts, gefahren werden sollen

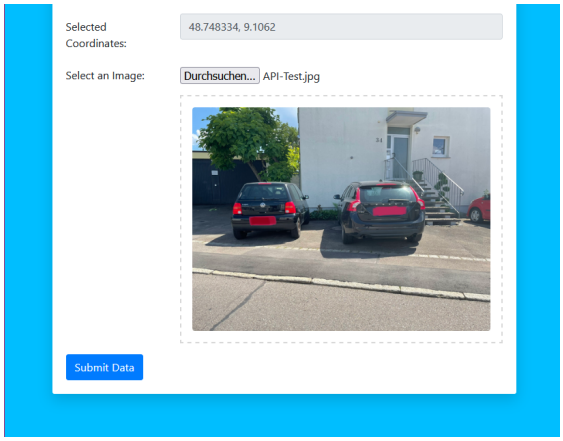
Tabelle 7.1: Auswahl einiger Optionen beim Konsolenaufruf von SuperTuxKart (welche unter anderem bei der Verwendung der Option --help ausgegeben werden)



(a) Eingabe der allgemeinen Daten



(b) Auswahl der Koordinaten



(c) Hochladen eines Bildes

Abbildung 7.5: Die Nutzeroberfläche zur Eingabe der Daten des physischen Zwillings

Um die Gegebenheiten in SuperTuxKart anzupassen, werden für die Analyse und die daraus resultierende Generierung des DZ andere Daten benötigt, als bei den bisherigen Entwürfen für den Prototyp. Um die umgebenden Autos zu finden, soll ein Foto verwendet werden. Des Weiteren werden für die Bestimmung der Tageszeit und des Wetters ein Zeitstempel ebenso wie die derzeitigen Koordinaten des Autos benötigt. Die allgemeinen Daten über das Auto, wie die Fahrgestellnummer, das Modell und die Marke sollen auch weiterhin gesammelt werden können. Die hierfür umgesetzte Nutzeroberfläche (UI) ist in Abbildung 7.5 zu betrachten. In Abbildung 7.5a ist die Eingabe der allgemeinen Daten zum Auto zu erkennen. Die Eingabe erfolgt über die Auswahl der Daten aus drei Drop-Down-Menüs. Darauf folgt, wie in Abbildung 7.5b zu sehen, das Festlegen der Koordinaten, durch einen Klick auf die abgebildete Karte. Als Standard sind hierbei die Koordinaten des Informatik-Gebäudes, der Universität Stuttgart, gewählt worden. Zuletzt ist in Abbildung 7.5c dargestellt, wie das Formular aussieht, nachdem über den Durchsuchen-Button ein Bild hochgeladen worden ist. Durch einen Klick auf den unteren blauen Button werden die Daten abgeschickt.

Dies ist die erste Komponente des finalen Prototyps, dessen Architektur in Abbildung 7.7 abgebildet ist. Wie zuvor sind die einzelnen Teile des Prototyps auf die drei Ebenen des Fog Computings aufgeteilt worden. Diese sind bereits in Abschnitt 2.5.2 vorgestellt worden. Die IoT-Ebene stellt

die Fahrzeuge dar. Dazu gehören die UI und der sogenannte SuperTuxKart Executor. Dieser führt SuperTuxKart entsprechend der Kommandos, welche sich im DZ befinden, aus. Darauf folgt die Fog-Ebene. Wie in dem vorigen Entwurf in Abschnitt 7.3, ist auch hier Kafka für die Nachrichtenübermittlung eingeordnet. Zusätzlich soll an dieser Stelle nun ebenfalls eine Vorverarbeitung der Daten stattfinden, welche in der Abbildung als Preprocessing bezeichnet wird. Die weitere Verarbeitung der Daten findet, wie zuvor, in der Cloud statt. Im Folgenden werden nun die einzelnen Komponenten näher beschrieben. Hierfür wird zusätzlich der in Abbildung 7.8 dargestellte Data Flow zwischen den Komponenten als Anhaltspunkt für die Reihenfolge der Beschreibungen herangezogen. Zunächst erfolgt, wie oben bereits beschrieben, die Eingabe der Daten über die UI. Sind die Daten über den Button abgesendet worden, wird ein Zeitstempel generiert und ein JSON-Objekt erzeugt, welches alle Daten außer das Bild enthält. Ein Beispiel dafür ist in Listing 7.4 zur Erklärung dargestellt. Die enthaltene `objectId` bildet die Verbindung des JSON-Objekts zum dazugehörigen Bild, welches für die weitere Verarbeitung in ein Byte-Array umgewandelt wird. Beide werden daraufhin über zwei Kafka-Producer an unterschiedliche Topics gesendet. Das JSON-Objekt befindet sich anschließend im `carDataTopic` und das Byte-Array, mit der `objectId` als Schlüssel, im `imageDataTopic`. Der nächste Verarbeitungsschritt, welcher in Abbildung 7.8 durch die Schritte 2 und 3 dargestellt ist, findet für das JSON-Objekt, im Preprocessing statt. Die Preprocessing-Komponente verwendet Kafka Streams[8]. Dies ist eine Bibliothek, mit welcher Daten verarbeitet werden können, welche in Kafka gespeichert sind. Dabei basiert es auf Stream Processing. Die Verarbeitung bildet eine Topologie, wobei es verschiedene Arten von `Processors`, beziehungsweise Verarbeitern, gibt. Diese bilden die Knoten in der Topologie und sind durch `Streams` miteinander verbunden. Streams sind Daten, welche durch Schlüssel-Wert-Paare repräsentiert werden. Ein `Source Processor` konsumiert Nachrichten von einem oder mehreren Kafka-Topics und leitet diese an darauf folgende Verarbeitungs-Knoten weiter. Für die Verarbeitung werden bereits einige Methoden, wie zum Beispiel `map` oder `filter`, angeboten. Zuletzt können `Streams` durch `Sink Processors` wieder an ein Kafka-Topic gesendet werden [Apa21c]. Im Preprocessing wird die `map`-Funktion verwendet, um JSON-Objekte, wie in 7.4, aus dem `carDataTopic` in zwei verschiedene JSON-Objekte zu spalten. Diese sind in Listing 7.1 und Listing 7.2 zu betrachten. Das Beispiel in Listing 7.1 enthält die Metadaten für den Data Lake, welche anschließend in das `metadataTopic` geschrieben werden. Das andere JSON-Objekt, welches in Listing 7.2 zu sehen ist, enthält die Daten, welche in Druid gespeichert werden sollen. Diese umfassen die aktuellen Koordinaten des Autos, welche in Verbindung mit dem Zeitstempel ihrer Aufnahme und der Fahrgestellnummer abgespeichert werden. Dabei werden die Koordinaten in diesem Fall als `Measurement` bezeichnet und die Fahrgestellnummer wird als `Tag` verwendet. Schritt 4 in Abbildung 7.8 symbolisiert in diesem Fall zwei Aktionen. Zum einen werden die Daten aus dem `metadataTopic` und `imageDataTopic` vom Kafka-Consumer des Data Lake Handlers konsumiert. Zum Anderen werden Daten aus dem `druidTopic` direkt von Druid geladen und abgespeichert. Dies geschieht mithilfe eines sogenannten Supervisors. Dieser kann entweder über die grafische Oberfläche von Druid erzeugt, selbst geschrieben und über die Oberfläche eingereicht oder per POST-Request an Druid gesendet werden. Die Supervisor-Spezifikation ist ein JSON-Objekt [Apa21a].

Die weiteren Daten werden mithilfe des Data Lake Handlers abgespeichert. Dabei werden die Metadaten aus dem `metadataTopic` in einer MySQL-Datenbank namens `MaMetadata` abgespeichert. Diese enthält zwei Tabellen, welche fast identisch zu den in Abschnitt 7.2 bereits beschriebenen sind. In der Data-Tabelle fehlt lediglich die Einheit für die Geschwindigkeit, da diese nun nicht mehr

[8] Zu finden unter: https://kafka.apache.org/documentation/streams/

Listing 7.4 Car Data JSON-Objekt, welches die eingegebenen Daten zu einem Fahrzeug enthält

```
{
  "coordinates":"48.608534, 9.1312",
  "chassisNr":"WDB2202561A074656",
  "model":"Intrepid",
  "make":"Starfleet",
  "objectId":"WDB2202561A074656782719",
  "timestamp":"2021-06-25T10:17:50"
}
```

benötigt wird und die Car-Tabelle bleibt unverändert. Die über das `imageDataTopic` gesendeten Byte-Arrays werden zusammen mit ihrer Objekt-Id als Schlüssel in einer MongoDB namens `imagesdb` unter der Collection `data`, in Form eines JSON-ähnlichen Dokuments, abgespeichert.

```
{
  "chassisNr":"WDB2202561A074656",
  "model":"Intrepid",
  "make":"Starfleet",
  "objectId":"WDB2202561A074656782719",
  "timestamp":"2021-06-25T10:17:50"
}
```

Listing 7.1: Metadaten

```
{
  "coordinates":"48.608534, 9.1312",
  "chassisNr":"WDB2202561A074656",
  "timestamp":"2021-06-25T10:17:50"
}
```

Listing 7.2: Daten welche in Druid abgespeichert werden sollen

Als nächstes folgt die Analyse, welche in Abbildung 7.8 als Schritt 5 dargestellt ist. Diese holt sich alle fünf Minuten die verschiedenen Daten aus dem Data Lake, welche für die Generierung des DZ benötigt werden. Für die Verarbeitung sind zwei APIs herangezogen worden. Zum einen die Recognition API von Sighthound[9], welche eine Fahrzeugerkennung bereitstellt. Um diese zu verwenden muss ein Account erstellt werden. Anschließend kann ein API Token generiert werden. Mit einem kostenlosen Token können pro Monat 5000 Anfragen an die API gesendet werden. Das erstellte Token muss im POST-Request als Header hinzugefügt werden. Für den API-Request kann ein Binary Stream eines Bildes oder eine URL zu einem Bild an folgenden Endpunkt gesendet werden: `https://dev.sighthoundapi.com/v1/recognition?objectType=vehicle`. Als Resultat kommt ein JSON-Objekt zurück, welches neben den Bounding-Boxes, welche angeben, wo sich die verschiedenen Fahrzeuge auf dem Foto befinden, auch Marke, Farbe und Modell auflistet [Sig20]. Nach einigen Versuchen ist entschieden worden, dass lediglich die Farbe der Autos aus dem Resultat verwendet wird, da meist falsche Resultate bezüglich der Marke und demnach auch des Modells zurück gemeldet wurden. Für die Zuordnung zwischen den realen Autos und den Karts in SuperTuxKart ist eine Liste der beliebtesten Autofarben herangezogen worden, um die große Auswahl an Farben einzuschränken. In der Untersuchung sind folgende Autofarben nach ihrer Häufigkeit aufgelistet: Grau, schwarz, weiß, blau, rot, braun, gelb, grün und orange [t-o21]. In

[9] Zu finden unter: https://docs.sighthound.com/cloud/recognition/

Wertebereich	Fahrer/Kart	Addons
Grau	Gavroche	
Schwarz		Black Shadow
Weiß		Icy
Blau		Alcador
Rot	Emule	
Braun		Cobra
Gelb	Suzanne	
Grün	Puffy	
Orange		173RX Mk2

Tabelle 7.2: Zuordnung der Karts zu Autofarben

Tabelle 7.2 ist die gewählte Zuordnung zwischen den Autofarben und den dazu passenden Karts zu erkennen. Um für jede Autofarbe ein passendes Kart zu finden, sind ebenfalls von anderen Nutzern erstellte Addons zu Rate gezogen worden. Zum Anderen ist die Weather API von OpenWeatherMap[10]. Um genau zu sein wird der Teil der API verwendet, mit welchem aktuelle Wetterdaten abgerufen werden können. Wie für die Recognition API von Sighthound, werden auch hier ein Account und API Key benötigt, welcher bei jeder Anfrage mitgesendet werden muss. Mit dem Default-Key können kostenlos bis zu 60 Anfragen pro Minute gesendet werden. Eine Anfrage, welche die aktuellen Wetterbedingungen zu bestimmten Koordinaten abruft, sieht wie folgt aus: `http://api.openweathermap.org/data/2.5/weather?lat=48.748334&lon=9.106202&units=metric&APPID=xxx`. Ein Ausschnitt aus dem zurückgelieferten Resultat, bei dem es sich um ein JSON-Objekt handelt, ist in Listing 7.5 zu betrachten, wobei hier nur der für die Analyse-Komponente relevante Teil abgebildet ist [Ope21c]. Für die weitere Verarbeitung wird der Wert von `main` unter `weather` verwendet. Der Wertebereich beinhaltet: Thunderstorm, Drizzle, Rain, Atmosphere (Mist, Smoke, Haze, Dust, Fog, Sand, Dust, Ash, Squall, Tornado), Snow, Clear, und Clouds [Ope21c]. Da die Rennstrecken in SuperTuxKart, nach Betrachtung aller Strecken im Spiel, nicht viele Wetterbedingungen darstellen, stehen nur folgende zur Verfügung: Rain, Snow, Clear, Clouds. Damit werden die Werte Thunderstorm, Drizzle, Rain und Atmosphere von OpenWeatherMap, für die weitere Klassifizierung mit Rain gleichgesetzt. Zusätzlich soll die Tageszeit mit einbezogen werden. Daraus resultiert, dass für jede Wetterbedingung zwei Rennstrecken benötigt werden, eine für den Tag und eine für die Nacht. Die ausgewählten acht Strecken und ihre Klassifizierung ist in Abbildung 7.6 zu betrachten. Wird nun das Bild aus Abbildung 7.5c an die Recognition API gesendet, Antwortet diese als Resultat mit drei Autos: Zwei schwarze und ein rotes Auto, da auf dem Foto eine kleine Ecke eines Fahrzeugs rechts zu sehen ist). In Kombination mit dem aktuellen Wetter (Clear) aus Listing 7.5 und 12 Uhr als aktuelle Tageszeit, ergibt sich aus den oben beschriebenen Klassifizierungen, folgendes Kommando zum Aufruf von SuperTuxKart: `supertuxkart.exe -N --track=hacienda --laps=1 --profile-laps=1 --aiNP=addon_black-shadow,addon_black-shadow,emule,tux`. Hierbei ist die gewählte Rennstrecke Hacienda, da das Wetter Clear und es Tag ist. Die aufgelisteten Autos entsprechen zwei schwarzen Autos (addon_black-shadow), einem roten (emule) Auto und dem Tux-Kart als Fahrzeug, welches den physischen Zwilling repräsentiert. Die restlichen Optionen sorgen dafür, dass das Rennen sofort startet, dass eine Runde gefahren und diese von Compu-

[10]Zu finden unter: https://openweathermap.org/

Listing 7.5 Relevanter Ausschnitt der Antwort auf den Request an OpenWeatherMap

```
{
  ...
  "coord": {
    "lon": 9.1062,
    "lat": 48.7483
  },
  "weather": [
  {
    "icon": "01d",
    "description": "clear sky",
    "main": "Clear",
    "id": 800
  }
  ],
  ...
}
```

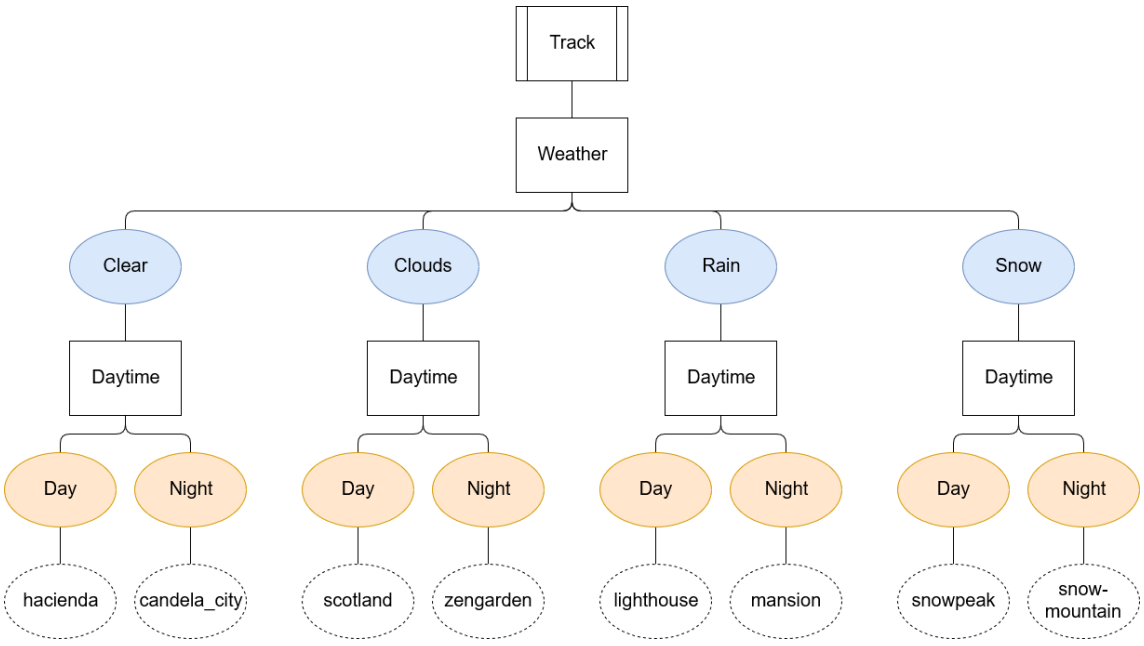


Abbildung 7.6: Klassifizierung der Rennstrecken nach Wetterbedingung und Tageszeit

ter gesteuert wird. Diese Optionen werden oben in Tabelle 7.1 beschrieben. Ist das Kommando generiert, wird es von der Analyse-Komponente in Schritt 10 der Abbildung 7.8 in einer MongoDB-Datenbank namens `digitaltwin`, unter der Collection `twins` zusammen mit dem Modell, der Marke und Fahrgestellnummer als ID abgespeichert. Dies bildet den DZ.

Der nächste Schritt wird von der Datensenke alle fünf Minuten durchgeführt. Hierbei werden für alle Autos die DZ über den DB-Accessor, zu sehen in Abbildung 7.7, aus der Datenbank geholt und die Kommandos von der DZ-Analyse extrahiert. Anschließend werden die Kommandos zusammen mit ihrer zugehörigen Fahrgestellnummer als Nachricht von einem Kafka-Producer in das `digitalTwinTopic` gesendet.

Die zuletzt aufgerufene Komponente, der SuperTuxKart Executor, erhält in Schritt 13 die im physischen Zwilling auszuführenden Kommandos über den Kafka-Consumer. Daraufhin werden, mithilfe des Command Extractors und Command Executors, zunächst die Kommandos extrahiert und dann ausgeführt.

Alle Komponenten sind in Java als dynamische Web-Projekte programmiert und es können Web Application Archives (WARs) generiert werden, welche auf einem Tomcat ausgeführt werden können. Apache Tomcat ist eine Open-Source-Implementierung verschiedener Spezifikationen der Jakarta EE Plattform, ehemals bekannt als Java EE. Ältere Versionen, wie Tomcat 9 und absteigend, unterstützen die Spezifikationen aus Java EE. Tomcat 8.5.x unterstützt unter anderem Servlet 3.1 und JavaServer Pages (JSP) 2.3. Somit ist Tomcat ein Servlet, beziehungsweise JSP Container, welcher verwendet werden kann, um Java-Web-Anwendungen auszuführen [Apa21d].

7.5 Umsetzung in Topology and Orchestration Specification for Cloud Applications

Dieser Prototyp soll nun unter Verwendung von TOSCA modelliert werden, mit dem Ziel, die ganze Applikation automatisch zu installieren und auszuführen. Statt die einzelnen Anwendungen lokal auf einem Computer zu nutzen, soll alles auf OpenStack[11] deployed werden. OpenStack ist eine Open-Source Cloud Software. Diese verwaltet verfügbare Ressourcen, wie zum Beispiel Rechenleistung und Speicher und bietet somit IaaS. Mit diesen Ressourcen kann ein Nutzer dann unter anderem VMs anlegen [Ope21a]. Auf den VMs sollen dann die einzelnen Teile des Prototyps laufen. Hierbei wird die OpenStack-Instanz der Universität verwendet[12]. Um den Ressourcenverbrauch geringer zu halten, ist für die VMs das als Abbild verfügbare `ubuntu-16.04-LTS-xenial-server-cloudimg` mit 2,2 GB Speicherbedarf ausgewählt worden. Des Weiteren sind die Varianten für die VMs möglichst klein gewählt worden. Für die WARs wird mit Variante `m1.small` eine VM mit einer VCPU, 2GB RAM und einer Größe von 20GB verwendet. Kafka, MySQL und die MongoDB laufen auf einer Instanz `m1.medium` mit zwei VCPUs, 4GB RAM und einer Größe von 40GB. Zuletzt ist für Druid eine `m1.lange`-Instanz gewählt worden, da die Datenbank unter Verwendung einer kleineren Instanz nicht funktioniert hat. Diese hat 4 VCPUs, 8GB RAM und eine Festplatte mit 80GB [Uni21b].

[11] Zu finden unter: https://www.openstack.org/
[12] Zu finden unter: https://astplos.ipvs.uni-stuttgart.de/project/

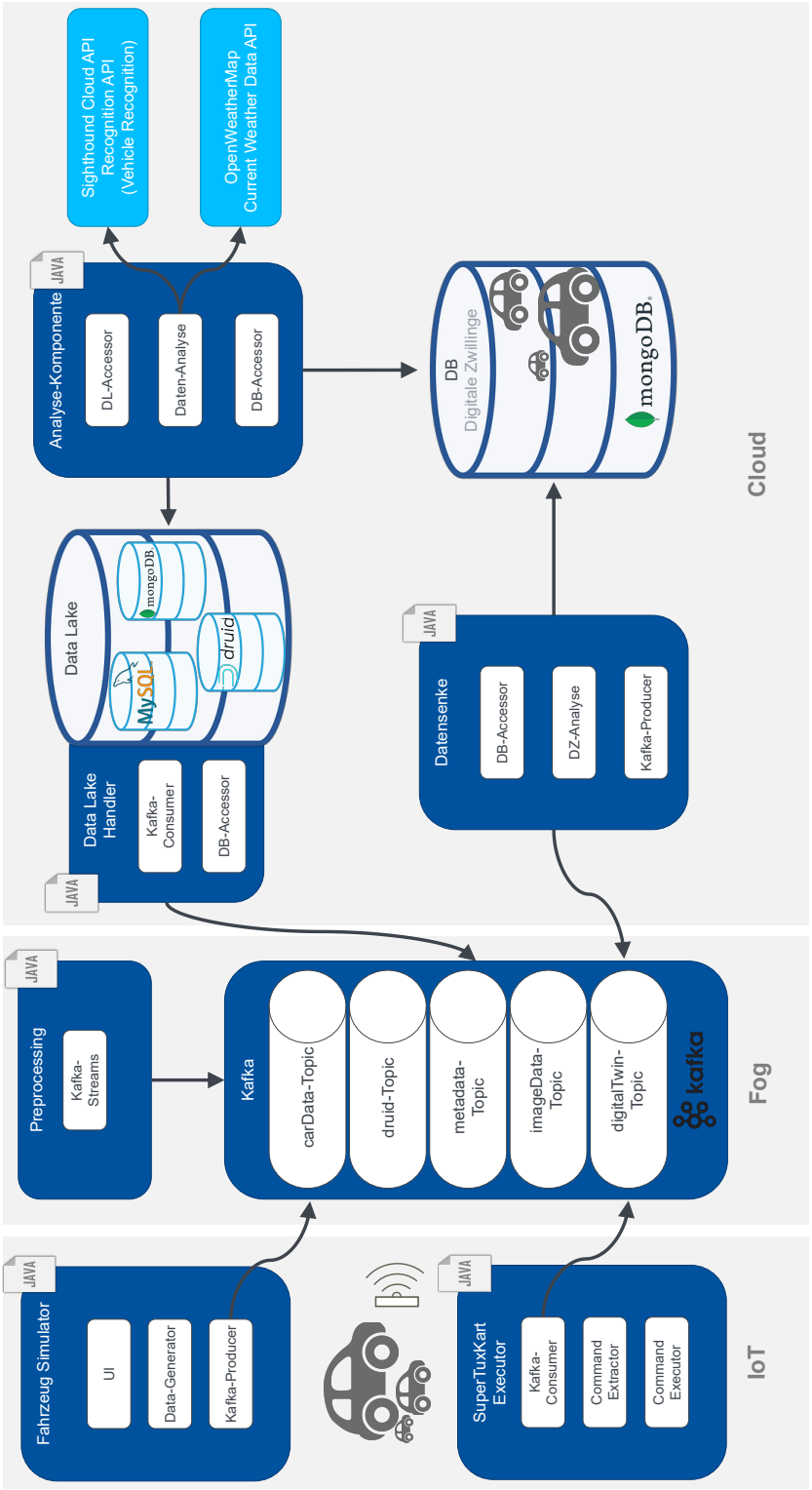


Abbildung 7.7: Architektur des finalen Prototyps für die Data Loop

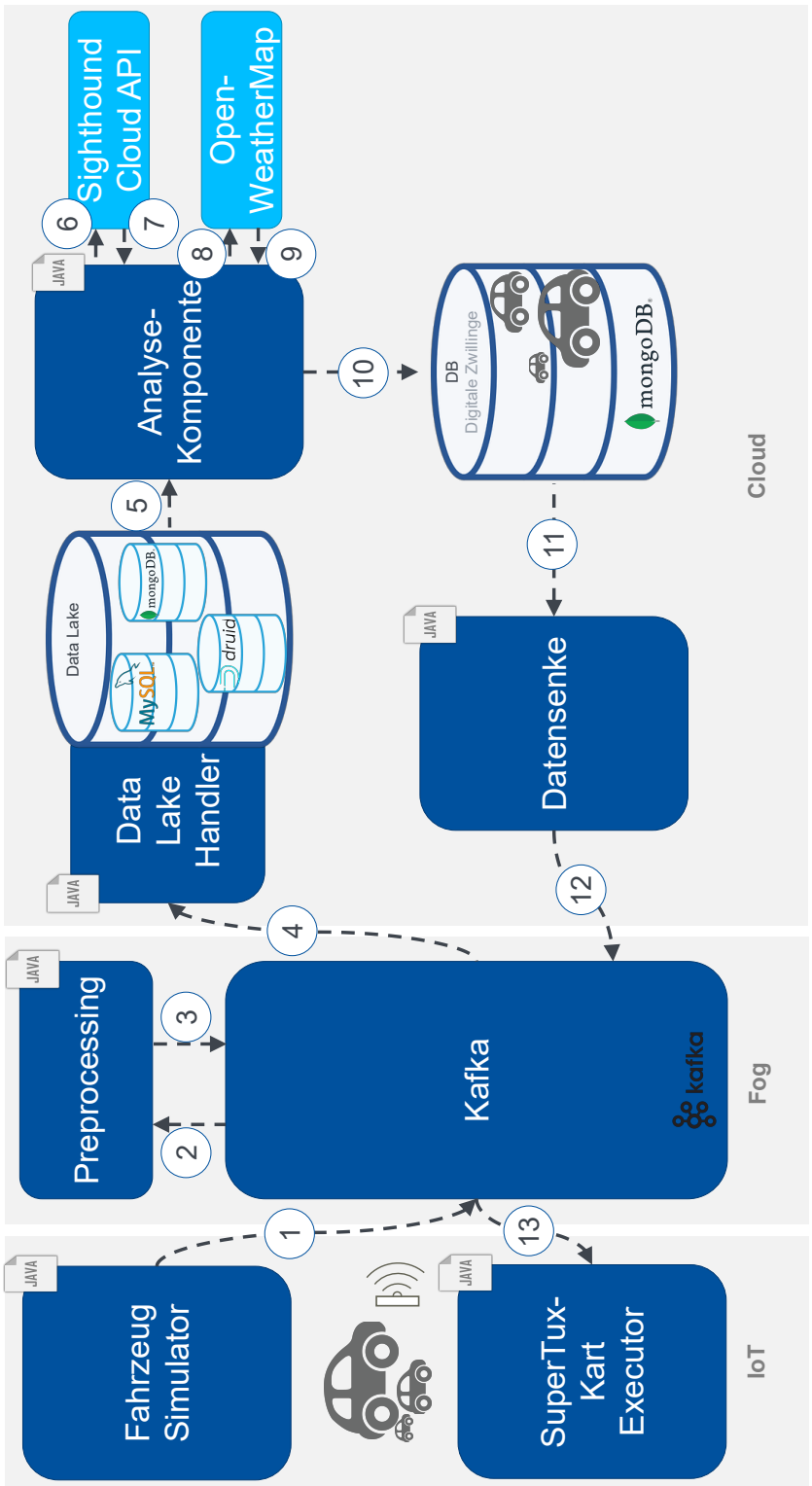


Abbildung 7.8: Data Flow des Prototyps der Data Loop

Wie oben bei dem Prototyp ist auch das Modell in TOSCA kontinuierlich weiterentwickelt worden und es sind mehrere Entwürfe entstanden, bis der Prototyp finalisiert worden ist. Im Folgenden werden nur die Entwürfe gezeigt, welche zu den oben bereits beschriebenen Implementierungen korrespondieren. Zusätzlich sind bei den ersten beiden zunächst Node Types erstellt worden, ohne eine Implementierung für das spätere Deployment zu hinterlegen. Das Modell für den ersten Entwurf des Prototyps ist in Abbildung 7.9 dargestellt. Hier sind alle der in Abbildung 7.2 zu sehenden Komponenten ebenfalls modelliert. Für die WARs ist dabei ein Java-Application Node Type angelegt worden, um diese abzubilden. Zusätzlich ist für die Speicherung der Metadaten ein MySQL-DB Node Type angelegt worden. Im Gegensatz zu den Datenbanken ist für die WARs zusätzlich bestimmt worden, dass diese auf einem Tomcat deployed werden, welcher auf einer Ubuntu 16.04 VM installiert wird. Das Deployment für die Datenbanken ist zugunsten der Übersichtlichkeit weggelassen worden, da diese ersten Entwürfe noch nicht bereitgestellt werden sollten. Auf der VM soll Java installiert werden. Dies ist durch die HostedOn-Beziehung modelliert. Damit Tomcat verwendet werden kann, wird Java benötigt [Apa21d]. Aus diesem Grund besteht eine DependsOn-Beziehung zwischen Tomcat und Java. Zuletzt hat die Java-Applikation eine Hosted-On-Beziehung zu Tomcat, da diese auf dem Tomcat-Server bereitgestellt werden soll.

Für die erweiterte Architektur in Abbildung 7.3 ist das dazugehörige Modell in Abbildung 7.10 zu betrachten. Hier ist zusätzlich eine Infrastruktur für Kafka hinzugekommen. Dabei sind in diesem Fall drei Teile modelliert worden. Zum einen Zookeeper, welcher vor dem Kafka Broker gestartet werden muss, was durch eine DependsOn-Beziehung modelliert ist, zum anderen Kafka selbst und zusätzlich die verwendeten Kafka-Topics. Da für die Verwendung von Kafka, Java 8 oder eine höhere Version benötigt wird, sind ebenfalls DependsOn-Beziehungen zwischen Zookeeper, dem Kafka Broker und dem Java-Knoten abgebildet worden [Apa21b]. Bis auf Zookeeper sind diese auch auf Abbildung 7.3 zu erkennen. Für die Darstellung der Time Series Datenbank ist in TOSCA, wie auf dem Bild des Modells zu erkennen, ein InfluxDB Node Type angelegt worden. Auch hier sind genaue Details über die Installation der Datenbanken vernachlässigt worden.

Vor der Modellierung des resultierenden Prototyps ist für die Übersichtlichkeit ein weiteres Service Template erstellt worden, welches in Abbildung 7.11 zu betrachten ist. So könnte das System auch in kleinere Teile aufgespaltet und getrennt bereitgestellt werden. Es ermöglicht den ersten Überblick über die Data Loop, wie sie in Kapitel 6 beschrieben ist. Hierfür ist zunächst der Data Lake, bestehend aus einer Java-Applikation und drei Datenbanken, zu einem Node Type namens DataLake zusammen gefasst worden. Zusätzlich ist das Messaging über eine Kafka-Komponente und das Speichern des DZ in einer MongoDB, im Vergleich zu Abbildung 7.2, ergänzt worden, da diese in Abbildung 7.3 schon vorgesehen sind. Genaue Details über das Deployment der einzelnen Komponenten, wie zum Beispiel ob dieses auf einer OpenStack-Instanz stattfindet, sind hierbei zur Generierung einer besseren Übersichtlichkeit vernachlässigt worden.

Das Service Template für den finalen Prototyp ist in Abbildung 7.7 visualisiert. Da dieses letzten Endes auch für die automatische Installation verwendet werden soll, ist auf die Verwendbarkeit der Knoten geachtet worden, was bedeutet, dass einige Node Types in kleineren Service Templates auf ihre Funktion getestet worden sind. Wie in der Abbildung zu erkennen, wird als Basis für alle Teile des Prototyps ebenfalls der Knoten `Ubuntu-VM_16.04-w1` verwendet. Dieser stellt eine Verbindung zu einer OpenStack-VM her auf welcher Ubuntu 16.04 installiert ist [Ope21b]. Wie zuvor diskutiert, soll auf jeder OpenStack-Instanz ein solches Betriebssystem verwendet werden. Für Java fällt die Wahl auf den Node Type `Java8`, da Druid nur mit Java 8 funktioniert [Apa21a]. Zusätzlich wird für Kafka ebenfalls ein Java mit mindestens Version 8 benötigt [Apa21b]. Aufgrund von

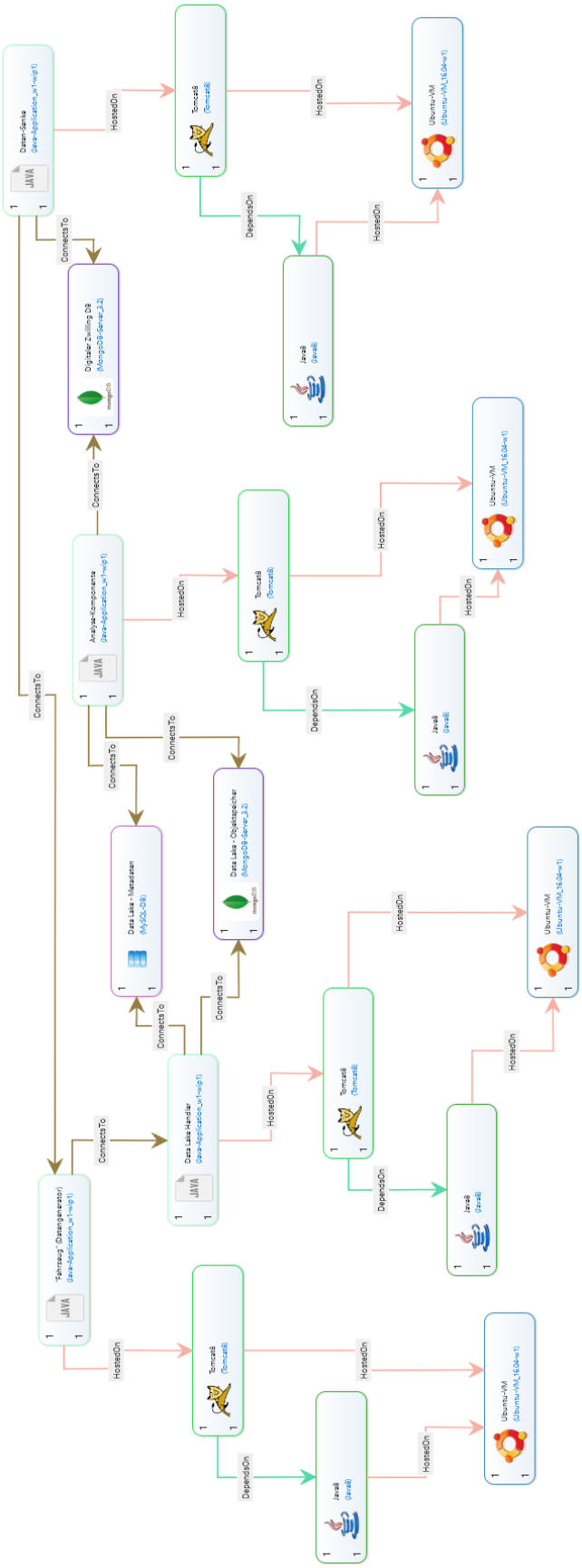


Abbildung 7.9: Modell in TOSCA für den ersten minimalen Prototyp

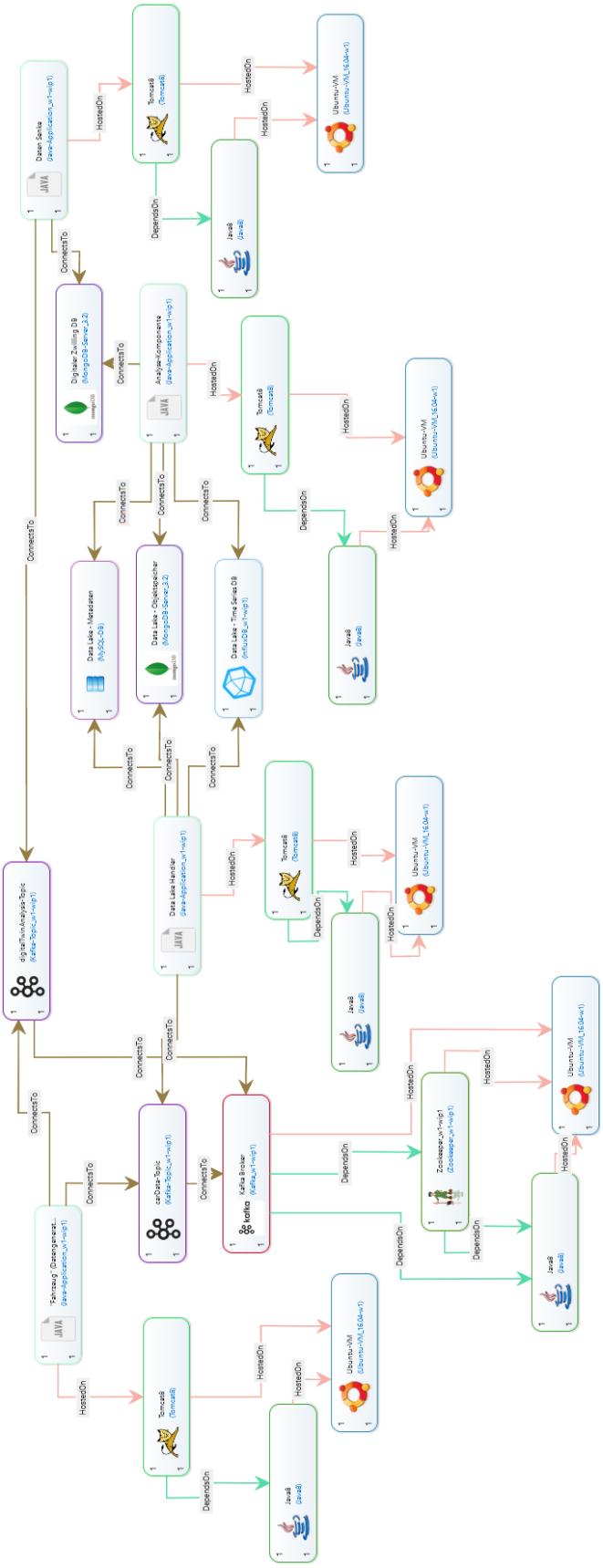


Abbildung 7.10: Modell in TOSCA für den erweiterten minimalen Prototyp

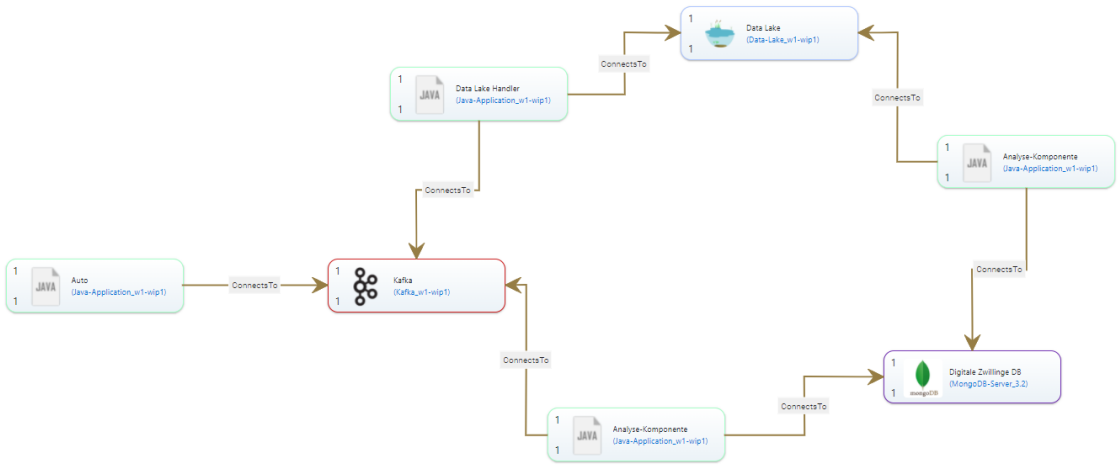


Abbildung 7.11: In TOSCA modellierter Überblick über die Data Loop

Gleichmäßigkeit wird für alle benötigten Java-Knoten `Java8` verwendet. Für Tomcat wird der Node Type `Tomcat8` verwendet. Dieser installiert einen Tomcat-Server der Version 8.5.28 und ist somit kompatibel mit Java 8 [Apa21d; Ope21b]. Um WARs auf einem Tomcat zu deployen, kann der Node Type `Java_Shop_Application` verwendet werden. Dieser nimmt eine WAR aus dem dazugehörigen Deployment Artifact und kopiert diese in den webapps-Ordner von Tomcat [Ope21b]. Befinden sich Web-Anwendungen in diesem Ordner, werden diese beim Start des Tomcat-Servers bereitgestellt [Apa21d]. Für die Bereitstellung von MySQL und MongoDB existieren ebenfalls bereits Node Types. Nach der Durchführung minimaler Tests mit dem Ziel, diese auf einer OpenStack-Instanz mit einem Ubuntu 16.04 zu deployen, ist es nicht möglich gewesen, einen Remote-Zugriff herzustellen. Somit sind einige Skripte angepasst worden, um dies zu ermöglichen. Für den Knoten `MySQL-DBMS_5.7-w1`, die Datei `configure.sh` und für `MongoDB-Server_3.2` die beiden Dateien `install.sh` und `start.sh`.

Für das automatische Deployment werden zusätzlich zwei weitere Node Types benötigt: Einer für Apache Druid und ein weiterer für Kafka. Der Node Type für Druid heißt `Druid_w1-wip1`. In der Implementation sind zwei Implementation Artifacts vom Typ Script Artifact enthalten: Eines für die Installation und eines für die Konfiguration. In Listing 7.6 ist das Skript für die Installation von Druid zu erkennen. Hierfür wird zunächst Druid heruntergeladen und danach entpackt. Daraufhin wird das Skript für die Konfiguration ausgeführt, welches in Listing 7.7 zu sehen ist. Da Druid für die Konfiguration bereits hochgefahren sein muss, gibt es kein separates SkriptArtifact für den Start von Druid. Für den Start der Datenbank wird anfangs das Verzeichnis gewechselt. Anschließend wird der Druid-Service im Hintergrund gestartet, der Output auf `/dev/null` umgelenkt und seine Prozess-Id ausgegeben. Um den in Abschnitt 7.4 beschriebenen Supervisor für das Auslesen aus einem Kafka-Topic zu konfigurieren, muss ein POST-Request an den Druid-Service gesendet werden. Bevor dies geschieht muss sichergestellt werden, dass der Service auch online ist [Apa21a]. Aufgrund dessen wird zuvor ein sleep ausgeführt. Zusätzlich muss der Output des curl-Aufrufs umgelenkt werden, da es beim Deployment über TOSCA ansonsten zu einem Fehler kommen kann. Der Supervisor wird in Form eines Strings vom Nutzer beim Deployment angegeben, da dieser als Property definiert worden ist.

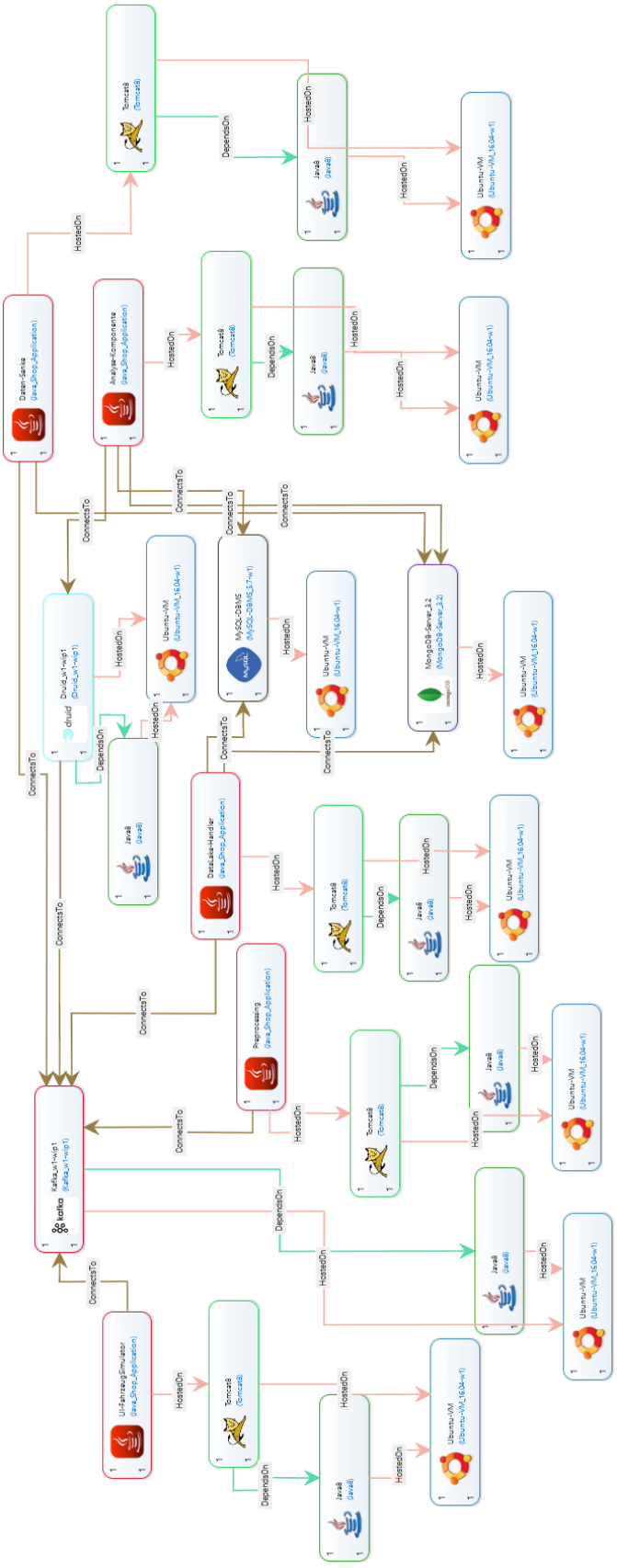


Abbildung 7.12: TOSCA-Modell für den finalen Prototyp

Listing 7.6 Script Artifact für die Installation von Druid (Nach Anleitung für eine minimale Installation [Apa21a])

```
    #!/bin/bash
    sudo sh -c "echo '127.0.0.1' $(hostname) >> /etc/hosts";
    echo " \n\n\n\n################l##############################################\n"
    echo " Setting Up Druid !!!"
    echo " \n##########################################################\n\n\n\n"
    # download druid
    sudo curl -O https://mirror.synyx.de/apache/druid/0.21.1/apache-druid-0.21.1-bin.tar.gz
    tar -xzf apache-druid-0.21.1-bin.tar.gz
```

Listing 7.7 Script Artifact für die Konfiguration von Druid [Apa21a])

```
    #!/bin/bash
    cd apache-druid-0.21.1
    # start druid in background
    ./bin/start-micro-quickstart &>/dev/null &
    echo $(pidof perl)
    # configure druid such that it loads data from a kafka topic
    # first wait for druid to have started before sending supervisor spec
    sleep 100
    cd ..
    echo $supervisor > supervisor.json
    curl -X POST -H 'Content-Type: application/json' -d @supervisor.json http://localhost
:8081/druid/indexer/v1/supervisor > curl-response.log
    echo "---------- CONFIGURED DRUID ----------"
```

Im Rahmen der Erstellung des `Kafka_w1-wip1` Node Types sind für die Installation drei Script Artifacts angelegt worden. Das Skript in Listing 7.8 lädt Kafka herunter und entpackt anschließend die Datei [Apa21a; Apa21b]. Für die Konfiguration, in 7.9, werden in `server.properties` zwei Zeilen abgeändert, sodass ein Remote-Zugriff erlaubt ist. Hierfür wird die Host-Adresse von Kafka benötigt, auf die von außen zugegriffen werden soll [Der17]. Diese muss vom TOSCA-Nutzer als VMIP-Property angegeben werden. Zuletzt wird das Start-Skript, welches in Listing 7.10 zu sehen ist, ausgeführt. Dieses startet zunächst Zookeeper, welches wie oben beschrieben für die Ausführung von Kafka benötigt wird. Abschließend wird Kafka gestartet [Apa21b]. Damit das Skript beim Deployment über TOSCA korrekt beendet wird sind zusätzlich ein `sleep 5` und ein `exit 0;` ergänzt worden.

Da die Kafka-Installation bereits eine Zookeeper-Instanz mitliefert, ist kein weiterer Node Type angelegt worden, obwohl dies zu Beginn vorgesehen war. Zusätzlich soll Kafka, laut den Entwicklern, in Zukunft auch ohne Zookeeper ausgeführt werden können [Apa21b].

Nach der Erstellung dieser Topologie in Winery, wird diese als CSAR exportiert. Anschließend kann sie in OpenTOSCA importiert und ausgeführt werden. Nach einer längeren Wartezeit werden alle Teile der Data Loop bereitgestellt und die Beispielanwendung ist, wie zuvor beschrieben, verwendbar indem zunächst über die UI Daten eingegeben werden. Nicht zu vergessen ist, dass

Listing 7.8 Script Artifact für die Installation von Kafka (Nach Anleitung [Apa21a; Apa21b])

```
#!/bin/bash
sudo sh -c "echo '127.0.0.1' $(hostname) >> /etc/hosts";
echo " \n\n\n\n################l##############################################\n"
echo " Setting Up Kafka !!!"
echo " \n##########################################################\n\n\n\n"
# download kafka
sudo curl -O https://archive.apache.org/dist/kafka/2.7.0/kafka_2.13-2.7.0.tgz
sudo tar -xzf kafka_2.13-2.7.0.tgz
```

Listing 7.9 Script Artifact für die Konfiguration von Kafka (Nach Anleitung [Der17])

```
#!/bin/bash
cd kafka_2.13-2.7.0
# configure kafka such that it can be remotely accessed
sudo sed -i 's@#listeners=PLAINTEXT://:9092@listeners=PLAINTEXT://:9092@g' config/server.
properties
sudo sed -i "s@#advertised.listeners=PLAINTEXT://your.host.name:9092@advertised.listeners=
PLAINTEXT://$VMIP:9092@g" config/server.properties
```

der in Abbildung 7.7 SuperTuxKart Executor lokal ausgeführt werden muss, damit dieser die von der Datensenke aus dem DZ extrahierten Kommandos über Kafka empfangen und anschließend SuperTuxKart mit den passenden Parametern ausführen kann.

Listing 7.10 Script Artifact für den Start von Kafka (Nach Anleitung [Apa21b])

```
#!/bin/bash
cd kafka_2.13-2.7.0
# start zookeeper and then kafka in background
nohup ./bin/zookeeper-server-start.sh config/zookeeper.properties 1>/dev/null 2>/dev/null
&
nohup ./bin/kafka-server-start.sh config/server.properties 1>/dev/null 2>/dev/null &
sleep 5
exit 0;
```

8 Evaluation des Prototyps

In diesem Kapitel wird evaluiert, ob und wie das Konzept aus Kapitel 6 und die in Kapitel 7 beschriebene prototypische Implementierung des Data Loop-Konzepts die zuvor gestellten Anforderungen aus Kapitel 5 erfüllt.

A.1 Skalierbarkeit:
Die erste Anforderung gibt an, dass die Data Loop und alle ihre Teile skalierbar sein sollen. Wie in Abschnitt 7.1.1 beschrieben, sind alle der zur Umsetzung des Data Lakes verwendeten Datenbanken skalierbar. Dazu gehören sowohl die MySQL, MongoDB und Apache Druid, welche alle horizontal skalierbar sind [Apa21a; Mon21; Ora21]. Neben den Datenbanken ist auch Kafka als das verwendete Messagingsystem skalierbar [Apa21b]. Somit erfüllen also alle verwendeten Technologien die Anforderung. Um aber die ganze Data Loop skalierbar zu machen, ist es nötig die einzelnen Java-Anwendungen mehrfach aufzusetzen und Load Balancer davor zu schalten. Diese können dann die Anfragen auf die verschiedenen Instanzen verteilen.

Die Verwendung von TOSCA für die Modellierung und das Deployment ermöglicht, wie oben bereits beschrieben, die Definition von verschiedenen Managementoperationen. Hier kann festgelegt werden, wie eine Komponente skaliert werden kann [BBKL14].

Damit kann die Anforderung A.1 zur Skalierbarkeit als erfüllt betrachtet werden.

A.2 Verarbeitung und Speicherung heterogener Echtzeitdaten:
Die Verarbeitung heterogener Echtzeitdaten wird durch die Verwendung von Kafka für das Senden und Empfangen von Nachrichten unterstützt. Kafka ist eine Event-Streaming-Plattform, was bedeutet, dass Daten in Echtzeit von ihren Quellen empfangen und weiter verarbeitet werden können [Apa21b]. Auch die Verwendung eines Data Lakes für die Speicherung der Daten hilft bei der Erfüllung dieser Anforderung, da der Data Lake nicht nur die Speicherung großer Mengen von Daten erlaubt, sondern auch Daten jeder Struktur sichert [GGH+20].

Somit kann Anforderung A.2 als erfüllt betrachtet werden.

A.3 Hohe Erweiterbarkeit und Modularisierung:
Die prototypische Implementierung der Data Loop besteht aus mehreren Teilen: Mehrere Java-Anwendungen, Kafka und der Data Lake, welcher wiederum aus einer Java-Anwendung für die Verwaltung der Daten und mehreren Datenbanken besteht. Aufgrund der Verwendung von Apache Kafka für das Messaging zwischen den verschiedenen Komponenten, können unter Berücksichtigung der existierenden Topics Anwendungen einfach ausgetauscht werden. Zusätzlich kann die Data Loop um zusätzliche Komponenten erweitert werden. Diese können Nachrichten aus bereits vorhandenen Topics verarbeiten oder es können neue Topics angelegt werden. Durch die Erstellung neuer Topics können zum Beispiel zusätzliche Analysen ergänzt werden. Diese können alle die gleichen Daten aus dem Data Lake verwenden, oder aufeinander aufbauen und eine Kette bilden.

Auch die Modellierung der Data Loop mithilfe von TOSCA unterstützt die Erweiterbarkeit des Systems. Auf diese Weise kann das Modell einfach unter Verwendung von Winery, zum Beispiel durch das Austauschen von Knoten und Anpassen der Beziehungen, aktualisiert werden. Anschließend kann das aktualisierte Modell mithilfe der Laufzeitumgebung von OpenTOSCA von neuem automatisch installiert und bereitgestellt werden.

Daraus folgt, dass auch die Anforderung A.3 nach hoher Erweiterbarkeit und Modularisierung erfüllt ist.

A.4 Automatisierbarkeit:
Für die Modellierung der Data Loop ist der Standard TOSCA verwendet worden. Das mit Winery erstellte Modell kann exportiert und dann im OpenTOSCA Container importiert werden. Daraufhin kann die Applikation automatisch bereitgestellt werden, vorausgesetzt, es wurden die entsprechenden Skripte zur Automatisierung des Deployments implementiert. Dies ist für die im Prototypen entstandenen Komponenten geschehen.

Die Anforderung A.4 der Automatisierung wird somit aufgrund der Verwendung von TOSCA erfüllt.

A.5 Verwendung von Standards und De-Facto-Standards:
Für die Modellierung und das Deployment der prototypischen Implementierung des Data Loop-Konzepts ist der Standard TOSCA verwendet worden. Zusätzlich ist für das Versenden und Empfangen von Nachrichten der De-Facto-Standard Kafka verwendet worden.

Damit ist auch die letzte Anforderung A.5 als erfüllt zu betrachten.

Insgesamt werden damit alle Anforderungen erfüllt. Durch die einfache Erweiterbarkeit besteht zusätzlich die Möglichkeit, weitere potentielle Anforderungen zu erfüllen.

Neben den Anforderungen bietet der entwickelte Ansatz weitere Vorteile. So wird durch die Fog-Architektur die fehlende Rechenkapazität der Steuergeräte in den Fahrzeugen ausgeglichen, da die Fog-Server und die Cloud weitere Ressourcen für die Verarbeitung bieten. Zusätzlich kann durch die Fog-Server das Datenvolumen reduziert werden, welches in die Cloud übertragen werden muss. Zum Beispiel durch eine Vorfilterung der Daten. Diese Vorfilterung kann auch für den Bereich Datensicherheit von Vorteil sein, da somit sensible Daten nicht in die Cloud geladen werden müssen. Des Weiteren kann durch das Verschieben einiger Analysen auf die Fog-Server die Latenzzeit reduziert werden, da diese Näher an den Fahrzeugen platziert sind. Aufgrund der Verwendung des DZ-Konzepts werden Dinge wie Softwareupdates Over-The-Air und die konstante Überwachung des Fahrzeugzustandes möglich [Krü20]. Die Struktur der Loop ermöglicht zudem die Rückmeldung der Analysen an das Fahrzeug, beziehungsweise den Fahrer. Dies macht zum Beispiel die Reaktion auf Hindernisse möglich.

9 Zusammenfassung und Ausblick

Heutzutage wird in Autos immer mehr Elektronik eingesetzt. Damit werden die Autos zunehmend vernetzter und autonomer. Die in den Autos verbauten Sensoren ermöglichen nicht nur die Kommunikation verschiedener Teile innerhalb des Autos, sondern auch die Vernetzung mit anderen Fahrzeugen. In diesem Zusammenhang wird von *Connected Cars* gesprochen. Mithilfe von Sensoren und Kommunikationstechnologien soll autonomes Fahren ermöglicht werden [CM16]. Mit dieser Technologie sollen Unfälle verhindert und die Fahrzeuginsassen sicher an ihr Ziel gebracht werden können [JG17; PKÅ+20]. Der Aufbau eines solchen Systems kann als IoT betrachtet werden. Dabei sind die Fahrzeuge, beziehungsweise deren Elektronik, wie die Sensoren, die Things [JG17].

Im Laufe der Arbeit ist ein solches IoT-Konzept in Form einer Fog-Architektur modelliert worden, welches sich Data Loop nennt. In der Data Loop werden Daten gesammelt und in einem Data Lake gespeichert. Anschließend werden die Daten verarbeitet und es wird ein sogenannter DZ generiert. Dieser spielt in einem Connected-Car-Szenario eine große Rolle. Es handelt sich dabei um ein Modell aus zwei Teilen: Der Physische Zwilling, bei welchem es sich in diesem Fall um ein oder mehrere Fahrzeuge handelt und der DZ, welcher eine digitale Repräsentation des Physischen Zwillings ist [Gri17]. Ist der DZ erstellt, so verändert sich dieser mit dem Physischen Zwilling. Genauso sollen sich Änderungen des DZ auch auf den Physischen Zwilling auswirken. Hierdurch entsteht die als Data Loop bezeichnete Schleife.

Für die Entwicklung der Data Loop ist zunächst betrachtet worden, welche Daten von Autos gesammelt werden und wie diese Daten verwendet werden könnten. Darauffolgend sind einige Anforderungen an ein solches System definiert worden. Auf dieser Basis ist zunächst das Konzept der Data Loop definiert worden. Daraufhin sind Prototypen der Data Loop entworfen worden, welche anschließend mithilfe des Modellierungsstandards TOSCA modelliert worden sind. Das finale Modell erfüllt nicht nur alle Anforderungen, sondern kann unter Verwendung von OpenTOSCA auch automatisch bereitgestellt und ausgeführt werden.

Ausblick und Diskussion

Die vorliegende Arbeit gibt einen Überblick über die gesamte Data Loop. In der Zukunft kann man weitere Arbeiten darauf aufbauen, indem man die einzelnen Teile der Data Loop genauer betrachtet. Dazu gehört zum Beispiel ein Data Lake, welcher speziell auf das Connected-Car-Szenario angepasst ist. Außerdem wäre es möglich, den DZ noch weiter auszubauen. Auch in Bezug auf weitere Analysemöglichkeiten können Nachforschungen angestellt werden.

In Bezug auf die Modellierung der Data Loop können ebenfalls weitere Möglichkeiten betrachtet werden. Hierfür könnte statt TOSCA zum Beispiel auch BPMN verwendet werden. Hierbei stellt sich dann eine weitere Herausforderung, welche untersucht werden kann: Das automatische Bereitstellen und Ausführen des Modells. Hierbei stellt sich die Frage, wie man in BPMN angeben kann, wo und wie die einzelnen Teile der Data Loop bereitgestellt werden sollen.

Wie in Kapitel 8 beschrieben, gibt die Erweiterbarkeit der Data Loop die Möglichkeit weitere potentielle Anforderungen zu erfüllen, welche nicht im Fokus dieser Arbeit standen. Dazu gehören die allgemeinen Anforderungen nach Informationssicherheit, Safety und Privacy. Diese werden im Folgenden diskutiert und können als Basis für zukünftige Arbeiten verwendet werden.

Sicherheit:
Joy et al. [JG17] beschreiben unter anderem, dass es zu Angriffen auf die Sicherheit (*Security, Cybersecurity*) kommen kann. Dazu gehören zum Beispiel DDoS-Angriffe. Solche Angriffe können im Fall des autonomen Fahrens tödlich enden. Aus diesem Grund muss jede Kommunikation im IoV sicher sein. Krüger [Krü20] beschreibt, dass die Zahl der Angriffspunkte in vernetzten Fahrzeugen steigt. Zu möglichen Angriffspunkten gehören böswillige Updates der Firmware, die Ausnutzung von Schwachstellen in Open-Source-Software, aber auch die Handys der Insassen können durch Apps eine Angriffsfläche bieten. Aus diesem Grund steigt die Bedeutung von Sicherheit im Automobilbereich.

Um die Sicherheit der vernetzen und intelligenten Fahrzeuge zu gewährleisten, werden verschiedene Standards entwickelt, welche in der Zukunft eingehalten werden müssen. Diese sind vom United Nations Economic Commision for Europe (UNECE) und genauer vom WP.29, welches das World Forum for Harmonization of Vehicle Regulations ist. Im Bereich der Cybersicherheit und Over-the-Air (OTA) gibt es zwei Regelungen, die in der Zukunft eingehalten werden sollen. OTA bedeutet dabei, dass ein Softwareupdate nicht mehr in einer Werkstatt durchgeführt werden muss, sondern *Over-the-Air*, was in der Zukunft zunehmen wird. Die Regelungen sehen vor, dass die Fahrzeughersteller die Sicherheit zu jedem Zeitpunkt sicherstellen müssen. Hierfür soll ein Software Update Management (SUMS) entwickelt werden. Unter anderem enthält das SUMS die Abhängigkeiten zwischen allen Software-Versionen und der Hardware und Software-Umgebungen. Außerdem soll es festlegen, welche Fahrzeuge ein Update benötigen und dieses sicher durchführen. Zur Erfüllung dieser Regelungen empfiehlt Krüger den DZ. Durch ihn können die Fahrzeuge durch den Hersteller überwacht und analysiert werden. Dadurch können die Integrität und Authentizität der aktuellen Software in einem Fahrzeug überprüft werden. Zusätzlich können Angriffe auf die Fahrzeuge im DZ simuliert werden, um Software-Updates zu überprüfen, bevor diese auf das Auto gespielt werden [Krü20]. Eine Erweiterung des in Kapitel 7 entwickelten und minimalen DZs hilft also bei der Erfüllung der Forderung von Sicherheit in einem vernetzen Fahrzeug.

Safety:
Ein weiterer wichtiger Aspekt, welcher bei Software auf einem Fahrzeug beachtet werden muss, ist die Sicherheit [Gri03]. Im Folgenden mit dem englischen Begriff *Safety* benannt, um eine Unterscheidung zu dem oben bereits genannten Begriff von Sicherheit, beziehungsweise Security, zu machen. Safety wird von Levenson als ein Maß festgelegt, welches angibt, inwiefern das System, sogenannte Safety-Ausfälle vermeiden kann. Ein solcher Ausfall kann zum Tod oder anderen vom Systementwickler festgelegten negativen Konsequenzen führen. Ein System, welches *safe* ist, verhindert, dass ein Zustand erreicht wird, welcher Safety-Ausfälle auslöst [Lev81]. Der Tod

eines Nutzers gehört also zu den Safety-Ausfällen. Damit kann ein oben beschriebener Angriff auf die Sicherheit also die Safety beeinflussen. Somit stehen diese beiden Anforderungen in einem Zusammenhang zueinander.

Für die Überprüfung der Safety eines Systems stellen Leveson et al. [LH83] eine Technik namens *Software Fault Tree* vor. Hierfür werden zunächst Risiken analysiert. Für die Konstruktion des Baumes wird mit einem möglichen Ereignis gestartet. Die Knoten darunter sind Voraussetzungen dafür, dass das Ereignis auftritt. Diese werden zusätzlich mit Oder- und Und-Beziehungen verknüpft. Die Ebene darunter gibt dann wieder die Vorbedingungen für die darüber liegenden Voraussetzungen an und so weiter. Ist ein Fault Tree fertig erstellt, wird dieser in Wahrheitswerte umgewandelt und ausgewertet. Auf diesem Weg können potentiell neue Fehlerszenarien entdeckt werden. Leveson [Lev86] beschreibt in einem weiteren Artikel, dass das System nicht nur anhand von Softwareanalysen safe gemacht werden kann, da die Verfahren komplex und fehleranfällig sind. Aus diesem Grund müssen die Systeme schon von Anfang an unter der Berücksichtigung von Safety entworfen werden, sodass durch Fehler oder Ausfälle keine Risiken auslösen können. Eine Serie an Standards für die Umsetzung von Safety im automobilen Bereich ist die *ISO 26262*[1].

Privacy:
Wie in Kapitel 4 beschrieben, werden im Fahrzeug viele Daten gesammelt und an den Hersteller weitergeleitet. Das Sammeln von Daten aus dem Auto, wie die Position oder der umgebenden Fahrzeuge, kann zu Datenschutzproblemen führen, da der Empfänger nicht nur die aktuelle Position kennt, sondern zum Beispiel auch auf die Fahrgewohnheiten des Fahrers schließen kann. Aus diesem Grund dürfen nur die nötigsten Daten weitergegeben werden [JG17].

Wie in Abbildung 7.12 zu sehen, befindet sich die Anwendung auf verschiedenen Ebenen. Dabei werden die Daten auf der IoT-Ebene, also in den Fahrzeugen, gesammelt. Wie oben beschrieben geht es darum, diese Daten zu schützen. Um diesen Schutz zu gewährleisten gibt es mehrere Ansätze, die umgesetzt werden könnten. Der erste und einfachste Ansatz wäre es, die Daten im Auto vorzufiltern, bevor diese über Kafka versendet werden. Somit würden nur die von der Applikation benötigten Daten nach außen getragen werden. Da die reduzierte Menge an Daten trotzdem die Privatsphäre des Datenbesitzers verletzen kann, sind weitere Ansätze notwendig, um dies zu verhindern. Hierfür beschreibt die General Data Protection Regulation (GDPR) zwei Möglichkeiten: Pseudonymisierung und Anonymisierung. Eine Pseudonymisierung der Daten beschreibt, dass Daten einer bestimmten Person so verändert werden, sodass sie einer Person gehören, welche vielleicht identifiziert wird. Die Anonymisierung hingegen sorgt dafür, dass die zu den Daten gehörige Person nicht identifiziert werden kann [Kot16]. Um eine Anonymisierung zu realisieren gibt es ebenfalls verschiedene Ansätze. Joy et al. [JG17] stellen für IoV *Haystack Privacy* vor. Unter Verwendung dieses Mechanismus, können Daten in einer Datenbank gespeichert werden, welche mehrere Kollaboratoren besitzt. Hierbei werden die Daten von den Datenbesitzern selbst privatisiert. Anschließend werden diese an die Cloud gesendet und dort aggregiert, um den nicht privatisierten Wert abzuschätzen. Dabei ist es wichtig anzumerken, dass die Daten mehrerer Datenbesitzer verarbeitet werden. Ein Beispiel für eine Frage, deren Antworten mithilfe des Haystack-Mechanismus geschützt werden können, ist, wie viele Leute sich an einem bestimmten Ort befinden. Ohne im Genauen auf die mathematischen Grundlagen einzugehen, werden die Daten privatisiert indem in diesem Fall mehrere Antworten bezüglich der Position gegeben werden, oder keine Auskunft über die aktuelle Position gegeben wird. Ein weiterer Ansatz für den Schutz privater Daten ist die Verwendung von *Differential*

[1] Zu finden unter: https://www.iso.org/standard/68383.html

Privacy. Diese sorgt für die Sicherung der Daten auch vor internen Angriffen, was bedeutet, dass Datenanalysten ihren Zugriff auf die Daten nicht ausnutzen können. Unter Verwendung des Ansatzes können Analysten eine statistische Datenbank abfragen, um Informationen über die Population einer Stichprobe zu bekommen. Dabei erhalten sie keine Informationen über die Einzelpersonen, welche Teil der Datenbank sind. Dies wird ermöglicht, indem *Noise* zum Resultat der Anfragen hinzugefügt wird, um die personenbezogenen Daten zu verschleiern [Dwo06].

Eine weitere Möglichkeit mit vertraulichen Daten umzugehen ist die Verwendung von Cloud Patterns. Strauch et al. [SBK+12] beschreiben mehrere dieser Pattern, welche verwendet werden können, wenn sich die Datenbank zum Speichern der Daten in der Cloud befindet.

Literaturverzeichnis

[ABD+16] M. Artač, T. Borovšak, E. Di Nitto, M. Guerriero, D. A. Tamburri. „Model-Driven Continuous Deployment for Quality DevOps“. In: *Proceedings of the 2nd International Workshop on Quality-Aware DevOps*. QUDOS '16. Association for Computing Machinery, Juli 2016, S. 40–41 (zitiert auf S. 32).

[ADA21] ADAC. *Diese Daten sammelt ein modernes Auto*. 2021. URL: https://www.adac.de/rund-ums-fahrzeug/ausstattung-technik-zubehoer/assistenzsysteme/daten-modernes-auto/ (zitiert auf S. 35).

[ADS+06] M. Anjanappa, K. Datta, T. Song, R. Angara, S. Li. „Introduction to Sensors and Actuators“. In: *The Mechatronics Handbook* (2006), S. 121–134 (zitiert auf S. 24).

[AIM10] L. Atzori, A. Iera, G. Morabito. „The internet of things: A survey“. In: *Computer networks* 54.15 (2010), S. 2787–2805. DOI: 10.1016/j.comnet.2010.05.010 (zitiert auf S. 15, 23).

[Ama21] Amazon Web Services, Inc. oder Tochterfirmen. *Fallstudie: BMW Group*. 2021. URL: https://aws.amazon.com/de/solutions/case-studies/bmw-group-case-study/ (zitiert auf S. 29).

[Apa21a] Apache Software Foundation. *Apache Druid - Druid | Interactive Analytics at Scale*. 2021. URL: https://druid.apache.org/ (zitiert auf S. 44, 53, 60, 63, 65–67).

[Apa21b] Apache Software Foundation. *Apache Kafka*. 2021. URL: https://kafka.apache.org/ (zitiert auf S. 45, 60, 65–67).

[Apa21c] Apache Software Foundation. *Apache Kafka - Kafka Streams*. 2021. URL: https://kafka.apache.org/documentation/streams/ (zitiert auf S. 53).

[Apa21d] Apache Tomcat Project. *Apache Tomcat - Welcome!* 2021. URL: https://tomcat.apache.org/ (zitiert auf S. 57, 60, 63).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications“. In: *International Conference on Service-Oriented Computing*. ICSOC 2013. Springer Berlin Heidelberg, Dez. 2013, S. 692–695 (zitiert auf S. 27, 28).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA“. In: *Proceedings of the IEEE International Conference on Cloud Engineering*. IEEE IC2E 2014. IEEE Computer Society, März 2014, S. 87–96 (zitiert auf S. 28).

[BBK+16] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, F. Leymann. „From Architecture Modeling to Application Provisioning for the Cloud by Combining UML and TOSCA.“ In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science*. CLOSER 2016. 2016, S. 97–108 (zitiert auf S. 32, 42).

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. In: *Advanced Web Services*. New York: Springer, Jan. 2014. Kap. TOSCA: Portable Automated Deployment and Management of Cloud Applications, S. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22 (zitiert auf S. 26, 27, 42, 67).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem - Concepts & Tools“. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* 1 (2016), S. 112–130. DOI: 10.5220/0007903201120130 (zitiert auf S. 28, 42).

[Ber16] C. Berger. „An Open Continuous Deployment Infrastructure for a Self-driving Vehicle Ecosystem“. In: *Open Source Systems: Integrating Communities*. IFIP 2016. 2016, S. 177–183. DOI: 10.1007/978-3-319-39225-7_14 (zitiert auf S. 32, 42).

[BKF17] A. Bader, O. Kopp, M. Falkenthal. „Survey and Comparison of Open Source Time Series Databases“. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband*. BTW 2017. 2017, S. 249–268 (zitiert auf S. 44).

[BKLW17] U. Breitenbücher, K. á. Képes, F. Leymann, M. Wurster. „Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?“ In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. SummerSOC 2017. 2017, S. 18–27 (zitiert auf S. 33).

[BMNZ14] F. Bonomi, R. Milito, P. Natarajan, J. Zhu. „Fog computing: A platform for internet of things and analytics“. In: *Big data and internet of things: A roadmap for smart environments*. Springer, März 2014, S. 169–186 (zitiert auf S. 25).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. „Fog computing and its role in the internet of things“. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. MCC '12. ACM, 2012, S. 13–16 (zitiert auf S. 24).

[Bon18] H. Bonnin. „The Certification Challenges of Connected and Autonomous Vehicles“. In: *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*. ERTS 2018. 2018 (zitiert auf S. 29).

[BSS+09] C. Buckl, S. Sommer, A. Scholz, A. Knoll, A. Kemper, J. Heuer, A. Schmitt. „Services to the field: An approach for resource constrained sensor/actor networks“. In: *2009 International Conference on Advanced Information Networking and Applications Workshops*. IEEE, 2009, S. 476–481 (zitiert auf S. 23, 24).

[CDKB12] G. Coulouris, J. Dollimore, T. Kindberg, G. Blair. *Distributed Systems: Concepts and Design*. Boston: Addison-Wesley Publishing Company, 2012. 1067 S. ISBN: 9780132143011 (zitiert auf S. 37).

[CM16] R. Coppola, M. Morisio. „Connected car: Technologies, Issues, Future Trends“. In: *ACM Computing Surveys (CSUR)* 49.3 (2016), S. 1–36. DOI: 10.1145/2971482 (zitiert auf S. 15, 69).

[Con21] Confluent. *Daten-Streaming-Lösung für die Automobilbranche – Confluent*. 2021. URL: https://www.confluent.de/industry-solutions/automotive/ (zitiert auf S. 31).

[DB16] A. V. Dastjerdi, R. Buyya. „Fog Computing: Helping the Internet of Things Realize Its Potential“. In: *Computer* 49.8 (Aug. 2016), S. 112–116 (zitiert auf S. 25, 40).

[Der17] Derlin. *Kafka - Unable to send a message to a remote server using Java*. 2017. URL: https://stackoverflow.com/questions/28146409/kafka-unable-to-send-a-message-to-a-remote-server-using-java (zitiert auf S. 65, 66).

[Dix10] Dixon, James. *Pentaho, Hadoop, and Data Lakes*. 2010. URL: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/ (zitiert auf S. 21).

[Dix14] Dixon, James. *Data Lakes Revisited*. 2014. URL: https://jamesdixon.wordpress.com/2014/09/25/data-lakes-revisited/ (zitiert auf S. 21).

[Doc21] Docker. *Empowering App Development for Developers | Docker*. 2021. URL: https://www.docker.com/ (zitiert auf S. 32, 42).

[Dwo06] C. Dwork. „Differential Privacy“. In: *33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006)*. Bd. 4052. Lecture Notes in Computer Science. 2006, S. 1–12 (zitiert auf S. 72).

[EQS17] H. Eichelberger, C. Qin, K. Schmid. „Experiences with the Model-based Generation of Big Data Pipelines“. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband*. BTW 2017. 2017, S. 49–56 (zitiert auf S. 32, 42).

[FBK+16] M. Falkenthal, U. Breitenbücher, K. Képes, F. Leymann, M. Zimmermann, M. Christ, J. Neuffer, N. Braun, A. W. Kempa-Liehr. „OpenTOSCA for the 4th Industrial Revolution: Automating the Provisioning of Analytics Tools based on Apache Flink“. In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT'16. 2016, S. 179–180. DOI: 10.1145/2991561.2998463 (zitiert auf S. 32, 33, 42).

[FHB+17] A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang. „Customization and provisioning of complex event processing using TOSCA“. In: *Computer Science - Research and Development* (2017), S. 1–11. DOI: 10.1007/s00450-017-0386-z (zitiert auf S. 33, 42).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014, S. 367 (zitiert auf S. 22).

[GGH+20] C. Giebler, C. Gröger, E. Hoos, R. Eichler, H. Schwarz, B. Mitschang. „Data Lakes auf den Grund gegangen“. In: *Datenbank-Spektrum* 20.1 (2020), S. 57–69. DOI: 10.1007/s13222-020-00332-0 (zitiert auf S. 21, 22, 67).

[Gre15] S. Greengard. „Automotive Systems Get Smarter“. In: *Communications of the ACM* 58.10 (2015), S. 18–20. DOI: 10.1145/2811286 (zitiert auf S. 15).

[Gri03] K. Grimm. „Software Technology in an Automotive Company - Major Challenges“. In: *25th International Conference on Software Engineering, 2003. Proceedings.* ICSE 2003. 2003, S. 498–503. DOI: 10.1109/ICSE.2003.1201228 (zitiert auf S. 70).

[Gri17] M. W. Grieves. „Digital Twin: Mitigating Unpredictable, Undesirable Emergent Behavior in Complex Systems“. In: *Transdisciplinary Perspectives on Complex Systems*. pringer International Publishing Switzerland, 2017, S. 85–113. DOI: 10.1007/978-3-319-38756-7_4 (zitiert auf S. 16, 19–21, 40, 69).

[Gri19] M. W. Grieves. „Virtually Intelligent Product Systems: Digital and Physical Twins“. In: *Complex Systems Engineering: Theory and Practice*. American Institute of Aeronautics und Astronautics, Inc., 2019, S. 175–200. DOI: 10.2514/5.9781624105654.0175.0200 (zitiert auf S. 16, 19, 20).

[HGQ16] R. Hai, S. Geisler, C. Quix. „Constance: An Intelligent Data Lake System“. In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. Association for Computing Machinery, Juni 2016, S. 2097–2100 (zitiert auf S. 21).

[HLC17] C. Huang, R. Lu, K.-K. R. Choo. „Vehicular Fog Computing: Architecture, Use Case, and Security and Forensic Challenges“. In: *IEEE Communications Magazine* 55.11 (2017), S. 105–111. DOI: 10.1109/MCOM.2017.1700322 (zitiert auf S. 41).

[Ian20] Ian Skerrett. *Why MQTT Has Become the De Facto Standard for the Connected Car*. 2020. URL: https://www.hivemq.com/blog/mqtt-standard-for-connected-car/ (zitiert auf S. 30).

[JG17] J. Joy, M. Gerla. „Internet of Vehicles and Autonomous Connected Car - Privacy and Security Issues“. In: *2017 26th International Conference on Computer Communication and Networks (ICCCN)*. ICCCN 2017. 2017, S. 1–9. DOI: 10.1109/ICCCN.2017.8038391 (zitiert auf S. 15, 36, 39, 69–71).

[JKK+14] K. Jo, J. Kim, D. Kim, C. Jang, M. Sunwoo. „Development of Autonomous Car—Part I: Distributed System Architecture and Development Process“. In: *IEEE Transactions on Industrial Electronics* 61.12 (2014), S. 7131–7140. DOI: 10.1109/TIE.2014.2321342 (zitiert auf S. 15).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications“. In: *Business Process Model and Notation*. Lecture Notes in Business Information Processing. 2012, S. 38–52. DOI: 10.1007/978-3-642-33155-8_4 (zitiert auf S. 33).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery – A Modeling Tool for TOSCA-Based Cloud Applications“. In: *International Conference on Service-Oriented Computing*. ICSOC 2013. Springer Berlin Heidelberg, Dez. 2013, S. 700–704 (zitiert auf S. 28).

[KKJ+14] W. Kastner, M. Kofler, M. Jung, G. Gridling, J. Weidinger. „Building Automation Systems Integration into the Internet of Things The IoT6 approach, its realization and validation“. In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*. IEEE, 2014, S. 1–9 (zitiert auf S. 24).

[KMC+18] S. A. Kumar, R. Madhumathi, P. R. Chelliah, L. Tao, S. Wang. „A novel digital twin-centric approach for driver intention prediction and traffic congestion avoidance“. In: *Journal of Reliable Intelligent Environments* 4.4 (2018), S. 199–209. DOI: 10.1007/s40860-018-0069-y (zitiert auf S. 30).

[KOJ11] P. Kleberger, T. Olovsson, E. Jonsson. „Security aspects of the in-vehicle network in the connected car“. In: *2011 IEEE Intelligent Vehicles Symposium (IV)*. IV 2011. 2011, S. 528–533. DOI: 10.1109/IVS.2011.5940525 (zitiert auf S. 15).

[Kot16] W. Kotschy. „The new General Data Protection Regulation -Is there sufficient pay-off for taking the trouble to anonymize or pseudonymize data?“ In: (2016) (zitiert auf S. 71).

[Krü20] J. Krüger. „Digital Twin für maximale Cyber Security: Mit digitalem Fahrzeug-Zwilling neue UNECE-Cyber-Security-Vorgaben einfacher erfüllen“. In: *Zeitschrift für wirtschaftlichen Fabrikbetriebg* 115.s1 (Apr. 2020), S. 29–31 (zitiert auf S. 16, 30, 68, 70).

[Lev81] N. G. Leveson. „Software Safety: A Definition and Some Preliminary Thoughts“. In: *ICS Technical Reports* 174 (1981), S. 1–36 (zitiert auf S. 70).

[Lev86] N. G. Leveson. „Software Safety: Why, What, and How“. In: *ACM Comput. Surv.* 18.2 (Juni 1986), S. 125–163 (zitiert auf S. 71).

[LGL+15] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. Wei, L. Sun. „Fog computing: Focusing on mobile users at the edge“. In: *CoRR* abs/1502.01815 (Feb. 2015) (zitiert auf S. 24–26, 40).

[LH83] N. Leveson, P. Harvey. „Analyzing Software Safety“. In: *IEEE Transactions on Software Engineering* SE-9.5 (Sep. 1983), S. 569–579 (zitiert auf S. 71).

[LKM+15] A. Luckow, K. Kennedy, F. Manhardt, E. Djerekarov, B. Vorster, A. Apon. „Automotive big data: Applications, workloads and infrastructures“. In: *2015 IEEE International Conference on Big Data (Big Data)*. Big Data 2015. IEEE, 2015, S. 1201–1210. DOI: 10.1109/BigData.2015.7363874 (zitiert auf S. 29).

[LL15] I. Lee, K. Lee. „The Internet of Things (IoT): Applications, investments, and challenges for enterprises“. In: *Business Horizons* 58.4 (2015), S. 431–440 (zitiert auf S. 23).

[Mar21] Margaretha Erber. *IoT Data Streaming – Warum MQTT und Kafka eine exzellente Kombination sind | Informatik Aktuell*. 2021. URL: https://www.informatik-aktuell.de/betrieb/netzwerke/iot-data-streaming-warum-mqtt-und-kafka-eine-exzellente-kombination-sind.html (zitiert auf S. 31).

[Mat17] C. Mathis. „Data Lakes“. In: *Datenbank-Spektrum* 17.3 (2017), S. 289–293. DOI: 10.1007/s13222-017-0272-7 (zitiert auf S. 21).

[Met20] Metje, Rainer and Gröger, Christoph. *Bosch Data Strategy*. 2020. URL: https://tpl.informatik.uni-stuttgart.de/wp-content/uploads/2020/10/20201006_Bosch-Data-Strategy_UniStgt_public.pdf (zitiert auf S. 29).

[MG11] P. Mell, T. Grance. „The NIST Definition of Cloud Computing“. In: *NIST special publication* 800 (2011), S. 1–3 (zitiert auf S. 22, 23).

[MK16] A. Meier, M. Kaufmann. *SQL- & NoSQL-Datenbanken*. Berlin, Heidelberg: Springer-Verlag, 2016. 258 S. ISBN: 978-3-662-47664-2 (zitiert auf S. 43).

[Mon21] MongoDB, Inc. *MongoDB - Die beliebteste Datenbank für moderne Apps*. 2021. URL: https://www.mongodb.com/de-de (zitiert auf S. 44, 67).

[MTA+16] P. Masek, M. Thulin, H. Andrade, C. Berger, O. Benderius. „Systematic Evaluation of Sandboxed Software Deployment for Real-time Software on the Example of a Self-Driving Heavy Vehicle“. In: *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*. ITSC 2016. 2016, S. 2398–2403. DOI: 10.1109/ITSC.2016.7795942 (zitiert auf S. 32, 42).

[NLJ08] D. K. Nilsson, U. E. Larson, E. Jonsson. „Creating a Secure Infrastructure for Wireless Diagnostics and Software Updates in Vehicles“. In: *Computer Safety, Reliability, and Security*. SAFECOMP 2008. Springer Berlin Heidelberg, 2008, S. 207–220. DOI: 10.1007/978-3-540-87698-4_19 (zitiert auf S. 15).

[NZM+19] F. Nargesian, E. Zhu, R. J. Miller, K. Q. Pu, P. C. Arocena. „Data Lake Management: Challenges and Opportunities“. In: *Proceedings of the VLDB Endowment* 12.12 (2019), S. 1986–1989. DOI: 10.14778/3352063.3352116 (zitiert auf S. 21).

[OAS21a] OASIS Committee Note Draft 01. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. 2021. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html (zitiert auf S. 25–27).

[OAS21b] OASIS Standard. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. 2021. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 26).

[OLe14] D. E. O'Leary. „Embedding AI and Crowdsourcing in the Big Data Lake". In: *IEEE Intelligent Systems* 29.5 (2014), S. 70–73. DOI: 10.1109/MIS.2014.82 (zitiert auf S. 21).

[Ope21a] OpenStack. *Open Source Cloud Computing Infrastructure - OpenStack*. 2021. URL: https://www.openstack.org/ (zitiert auf S. 57).

[Ope21b] OpenTOSCA. *OpenTOSCA/tosca-definitions-public: Preparation of the public TOSCA definitions repository*. 2021. URL: https://github.com/OpenTOSCA/tosca-definitions-public/ (zitiert auf S. 60, 63).

[Ope21c] OpenWeatherMap. *urrent weather and forecast - OpenWeatherMap*. 2021. URL: https://openweathermap.org/ (zitiert auf S. 55).

[Ora21] Oracle. *MySQL*. 2021. URL: https://www.mysql.com/de/ (zitiert auf S. 43, 67).

[PKÅ+20] P. Pelliccione, E. Knauss, S. M. Ågren, R. Heldal, C. Bergenhem, A. Vinel, O. Brunnegård. „Beyond connected cars: A systems of systems perspective". In: *Science of Computer Programming* 191 (Juni 2020) (zitiert auf S. 15, 39, 69).

[PVL+10] B. Piascik, J. Vickers, D. Lowry, S. Scotti, J. Stewart, A. Calomino. „Materials, Structures, Mechanical Systems, and Manufacturing Roadmap". In: *NASA Technology Area 12* (Nov. 2010), S. 1–36 (zitiert auf S. 19).

[RRSM16] F. M. Ribeiro, T. da Rocha, J. C. S. Santos, E. D. Moreno. „A Model-Driven Solution for Automatic Software Deployment in the Cloud". In: *Information Technology: New Generations*. Springer International Publishing, März 2016, S. 591–601 (zitiert auf S. 32, 42).

[RVKK19] A. Rassõlkin, T. Vaimann, A. Kallaste, V. Kuts. „Digital twin for propulsion drive of autonomous electric vehicle". In: *2019 IEEE 60th International Scientific Conference on Power and Electrical Engineering of Riga Technical University (RTUCON)*. RTUCON 2019. 2019, S. 1–4. DOI: 10.1109/RTUCON48111.2019.8982326 (zitiert auf S. 30).

[RVT+18] L. F. Rivera, N. M. Villegas, G. Tamura, M. Jiménez, H. A. Müller. „UML-driven Automated Software Deployment". In: *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*. CASCON '18. IBM Corp., Okt. 2018, S. 257–268 (zitiert auf S. 32, 42).

[SBH+17] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. „Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments". In: *In Proceedings of the 7th International Conference on Cloud Computing and Services Science*. CLOSER 2017. 2017, S. 330–339. DOI: 10.5220/0006243303580367 (zitiert auf S. 33, 42).

[SBK+12] S. Strauch, U. Breitenbuecher, O. Kopp, F. Leymann, T. Unger. „Cloud Data Patterns for Confidentiality“. In: *Proceedings of the 2nd International Conference on Cloud Computing and Services Science*. CLOSER 2012. SCITEPRESS (Science und Technology Publications, Lda.), 2012, S. 387–394. DOI: 10.5220/0003893903870394 (zitiert auf S. 72).

[SBK+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. „Opentosca for iot: automating the deployment of iot applications based on the mosquitto message broker“. In: *Proceedings of the 6th International Conference on the Internet of Things*. IoT'16. 2016, S. 181–182. DOI: 10.1145/2991561 (zitiert auf S. 33, 42).

[Sch14] E. Schicker. *Datenbanken und SQL - Eine praxisorientierte Einführung mit Anwendungen in Oracle, SQL Server und MySQL*. Wiesbaden: Springer Fachmedien, 2014. 355 S. ISBN: 978-3-8348-2185-0 (zitiert auf S. 43).

[Sch16] T. Schneider. „Achieving Cloud Scalability with Microservices and DevOps in the Connected Car Domain“. In: *CEUR Workshop Proceedings on Continuous Software Engineering*. SE-WS 2016. http://ceur-ws.org, Feb. 2016, S. 138–141 (zitiert auf S. 37).

[SD16] W. Shi, S. Dustdar. „The Promise of Edge Computing“. In: *Computer* 49.5 (Mai 2016), S. 78–81 (zitiert auf S. 24, 25).

[Sig20] Sighthound. *Recognition API - Vehicle and License Plate*. 2020. URL: https://docs.sighthound.com/cloud/recognition/ (zitiert auf S. 54).

[Sri20] Sridhar Mamella. *Big Data bei Porsche: Echtzeit-Verarbeitung von Streaming-Daten*. 2020. URL: https://newsroom.porsche.com/de/2020/innovation/porsche-daten-streaming-revolution-tool-streamzilla-sridhar-mamella-plattformmanager-22787.html (zitiert auf S. 30).

[Sup21] SuperTuxKart Team. *SuperTuxKart*. 2021. URL: https://supertuxkart.net/Main_Page (zitiert auf S. 49).

[t-o21] t-online. *Auto: Diese Farben sind in Deutschland besonders beliebt*. 2021. URL: https://docs.sighthound.com/cloud/recognition/ (zitiert auf S. 54).

[TSRC15] I. G. Terrizzano, P. M. Schwarz, M. Roth, J. E. Colino. „Data Wrangling: The Challenging Yourney from the Wild to the Lake“. In: *7th Biennial Conference on Innovative Data Systems Research*. CIDR '15. Jan. 2015, S. 1–9 (zitiert auf S. 22).

[TV07] A. S. Tanenbaum, M. Van Steen. *Distributed Systems: Principles and Paradigms*. New Jersey: Pearson Prentice Hall, 2007. 686 S. ISBN: 9783827372932 (zitiert auf S. 37).

[Uni21a] United States Department of Transportation. *Intelligent Transportation Systems - Connected Vehicles Basics*. 2021. URL: https://www.its.dot.gov/cv_basics/cv_basics_20qs.htm (zitiert auf S. 15).

[Uni21b] Universität Stuttgart. *OpenStack der Universität Stuttgart*. 2021. URL: https://astplos.ipvs.uni-stuttgart.de/project/ (zitiert auf S. 57).

[VD19] O. Veledar, G. Damjanovic-Behrendt Violeta and Macher. „Digital Twins for Dependability Improvement of Autonomous Driving“. In: *Systems, Software and Services Process Improvement*. EuroSPI 2019. Springer International Publishing, 2019, S. 415–426. DOI: 10.1007/978-3-030-28005-5_32 (zitiert auf S. 30).

[VWB+16] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, D. S. Nikolopoulos. „Challenges and Opportunities in Edge Computing“. In: *2016 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, Nov. 2016, S. 20–26 (zitiert auf S. 24).

[Wae20a] K. Waehner. „Apache Kafka in the Automotive Industry“. In: (2020), S. 1–10 (zitiert auf S. 31).

[Wae20b] Waehner, Kai and Lysak, Paul. *Confluent, MQTT, and Apache Kafka Power Real-Time IoT Use Cases*. 2020. URL: https://www.confluent.de/blog/iot-streaming-use-cases-with-kafka-mqtt-confluent-and-waterstream/ (zitiert auf S. 31).

[WBH+20a] K. Wild, U. Breitenbücher, L. Harzenetter, F. Leymann, D. Vietz, Z. Michael. „TOSCA4QC: Two Modeling Styles for TOSCA to Automate the Deployment and Orchestration of Quantum Applications“. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. EDOC 2020. 2020, S. 125–134. DOI: 10.1109/EDOC49727.2020.00024 (zitiert auf S. 32, 42).

[WBH+20b] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. „TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies.“ In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. CLOSER 2020. 2020, S. 216–226 (zitiert auf S. 33).

[WBK+18] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov. „Modeling and Automated Deployment of Serverless Applications using TOSCA“. In: *Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications (SOCA)*. SOCA 2018. 2018, S. 73–80. DOI: 10.1109/SOCA.2018.00017 (zitiert auf S. 32, 42).

[WZX+16] D. Wu, L. Zhu, X. Xu, S. Sakr, D. Sun, Q. Lu. „Building Pipelines for Heterogeneous Execution Environments for Big Data Processing“. In: *IEEE Software* 33.2 (2016), S. 60–67. DOI: 10.1109/MS.2016.35 (zitiert auf S. 31, 42).

[Xia17] Xiao, Yu and Chao Zhu. „Vehicular fog computing: Vision and challenges“. In: *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. PerCom Workshops. 2017, S. 6–9. DOI: 10.1109/PERCOMW.2017.7917508 (zitiert auf S. 41).

[YLH+18] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, X. Yang. „A Survey on the Edge Computing for the Internet of Things“. In: *IEEE Access* 6 (2018), S. 6900–6919 (zitiert auf S. 23).

[ZBL17] M. Zimmermann, U. Breitenbücher, F. Leymann. „A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications“. In: *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS*. ICEIS 2017. 2017, S. 121–131 (zitiert auf S. 33).

Alle URLs wurden zuletzt am 02. 11. 2021 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