Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Deployment und Ausführung verteilter Datenpipelines in Software-Defined-Car-Anwendungen

Marko Kovačić

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. Bernhard Mitschang
Betreuer/in:	Dr. rer. nat. Pascal Hirmer, Daniel Del Gaudio, M.Sc.
Beginn am:	5. April 2022
Beendet am:	5. Oktober 2022

Kurzfassung

Anwendungen im Bereich des Software-Defined-Car (SDC) bieten neue Möglichkeiten, um die Sicherheit und Effizienz in der Verkehrssteuerung, sowie dem Kraftstoffverbrauch zu verbessern. Die Kommunikation zwischen externen Systemen, wie der Infrastruktur, und anderen Verkehrsteilnehmern ermöglicht es Fahrzeugen nicht nur auf die eigene Sensorik, sondern auch auf Daten von außerhalb zu reagieren. Gleichzeitig ergibt sich durch den Datenaustausch das Potenzial Analysen mit Fahrzeugdaten auf externen Komponenten vorzunehmen, welche die verfügbaren Ressourcen innerhalb eines Fahrzeugs übersteigen. Durch die zunehmende Zahl an Fahrzeugen und eine stetig wachsende Menge an zu verarbeitenden Daten ergeben sich jedoch Herausforderungen für das Deployment und die Ausführung von SDC-Anwendungen. In der Literatur existieren bereits modellgetriebene Lösungsansätze für die einfache und schnelle Erstellung sowie Verwaltung solcher Anwendungen. Gleichzeitig mangelt es diesen bislang an Deployment- und Ausführungstechniken, insbesondere für solch heterogene und verteilte Domänen wie dies bei vernetzten Fahrzeugen der Fall ist. Aufbauend auf einem modellgetriebenen Lösungsansatz wird im Rahmen dieser Masterarbeit ein Konzept für das automatisierte Deployment und die Ausführung verteilter Datenpipelines in SDC-Anwendungen vorgestellt. Die Umsetzbarkeit des Konzepts wird durch einen implementierten Prototyp demonstriert, welcher gleichzeitig als Basis für eine Evaluation dient.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application Programming Interface. 25

DSRC Dedicated Short-Range Communication. 19

ECU Electrical Control Unit. 15

GSM Global System for Mobile Communications. 20

IoT Internet of Things. 15

ITS Intelligent Transportation System. 17

JSON JavaScript Object Notation. 63

Lidar Light Detection and Ranging. 15

OBU Onboard Unit. 17

OEM Original Equipment Manufacturer. 19

RSU Roadside Unit. 17

SDC Software-Defined-Car. 3

TOSCA Topology and Orchestration Specification for Cloud Applications. 29

UMTS Universal Mobile Telecommunications System. 20

V2I Vehicle-to-Infrastructure. 18

V2P Vehicle-to-Pedestrian. 18

V2V Vehicle-to-Vehicle. 18

V2X Vehicle-to-Everything. 18

VANET Vehicular Ad Hoc Network. 18

VF Vernetztes Fahrzeug. 15

VM Virtual Machine. 25

VRU Vulnerable Road User. 17

WAVE Wireless Access in Vehicular Environments. 19

1 Einleitung

> „*Neue Förderrichtlinie ‚Autonomes und vernetztes Fahren in öffentlichen Verkehren' veröffentlicht*“ [Bun22]

Mit diesem Titel eines Artikels gibt das Bundesministerium für Digitales und Verkehr im September 2022 bekannt, dass anwendungsorientierte Forschungsvorhaben für das vernetzte und autonome Fahren gefördert werden. Das Ziel des Forschungsvorhabens ist es, zur Steigerung der Verkehrssicherheit sowie einer emissionsreduzierten und effizienten Mobilität beizutragen. Diese Initiative zeigt, dass Forschungsbedarf besteht, um Fahrzeuge weiter in die bestehende Infrastruktur zu integrieren und einen Mehrwert aus den vom Fahrzeug generierten Daten zu gewährleisten.

Seit Jahren steigt die Zahl der Steuergeräte in Fahrzeugen an, wodurch zahlreiche Sicherheits- und Assistenzsysteme innerhalb eines Fahrzeugs realisiert werden können [JCB+15]. Neue Technologien wie Light Detection and Ranging (Lidar) ermöglichen die Erfassung des Umfelds, während altbewährte Technologien wie Raddrehzahlsensoren es erlauben, auf durchdrehende oder blockierende Räder zu reagieren [SMTS13; WES03]. Steuergeräte, auch Electrical Control Unit (ECU) genannt, sind mit dem Bussystem des Fahrzeugs verbunden und erhalten über dieses Sensordaten [HBC+07]. Durch die Software auf einem Steuergerät werden anhand der erhaltenen Daten Berechnungen durchgeführt, welche zu Aktionen wie dem Auslösen eines Airbags führen können [Cha02].

Ein Software-Defined-Car (SDC), im Nachfolgenden auch Vernetztes Fahrzeug (VF) genannt, unterscheidet sich von einem herkömmlichen Fahrzeug durch die Möglichkeit der bidirektionalen Kommunikation mit anderen Systemen außerhalb des Fahrzeugs [WCL09]. Daher erhält ein VF nicht nur Informationen durch die verbauten Sensoren, sondern kann auch Daten der Infrastruktur, Fußgänger, der Cloud und anderer Fahrzeuge verarbeiten. Dies ermöglicht eine Verbesserung bestehender oder die Einführung neuer Funktionalitäten, wie beispielsweise die frühzeitige Warnung eines voraus liegenden Stauendes [CRR+18]. Einen ausführlichen Einblick in das Thema der VFs, sowie deren Einordnung in das Internet of Things (IoT), behandelt Holland [Hol19] und stellt weitere Anwendungsmöglichkeiten und Potenziale vor.

Anwendungen für vernetzte Fahrzeuge gehen jedoch mit einigen Herausforderungen einher. Zur Kommunikation mit externen Komponenten, muss das Fahrzeug mit diesen Daten austauschen, was in Form von Nachrichten geschehen kann [DS17]. Durch die steigende Anzahl an Fahrzeugen auf den Straßen und die größer werdende Menge der generierten Daten, müssen Systeme in der Lage sein die hohe Zahl der anfallenden Nachrichten zu verarbeiten [CM16]. Aus diesem Grund ist es notwendig skalierbare Technologien mit hohem Durchsatz auszuwählen, um auch in Stoßzeiten mit der großen Last zurechtzukommen. Zusätzlich muss beachtet werden, dass SDCs lediglich begrenzte Ressourcen zur Verfügung stehen und daher speicher- oder rechenintensive Anwendungen möglicherweise nicht auf diesen ausgeführt werden können [SME20]. Dem kann durch den Einsatz verteilter Systeme entgegengewirkt werden, da ressourcenintensive Berechnungen auf Komponenten

außerhalb der Fahrzeuge ausgelagert werden können [GSS+18]. In Ihrem Buch beschreiben Hohpe und Woolf [HW04] verschiedene Kategorien von Herausforderungen beim Umgang mit verteilten Anwendungen und präsentieren eine Sammlung an möglichen Lösungsansätzen für diese.

Ziel dieser Arbeit ist die Erstellung eines Konzepts für die Ausführung und das Deployment von Datenpipelines in SDC-Anwendungen, welches anschließend durch eine prototypische Implementierung evaluiert wird. Als Ausgangspunkt für das Konzept werden Anforderungen definiert, welche für Anwendungen in diesem Bereich gelten sollen. Das darauf basierende Konzept erläutert in den drei folgenden Punkten, wie eine SDC-Anwendung aufgebaut sein kann:

- Modellierung der Datenpipeline-Modelle
- Deployment erstellter Datenpipeline-Modelle
- Ausführung von Datenpipelines

Bislang liegt der Fokus im Bereich der vernetzten Fahrzeuge entweder auf der Modellierung oder dem Deployment und der Ausführung von Software-Defined-Car-Anwendungen. Aus diesem Grund ist es bislang nur Nutzern mit Softwarekenntnissen möglich diese Anwendungen zu Erstellen und zu Verwalten. Um dem entgegenzuwirken, ist das Ziel dieser Arbeit die Zusammenführung von Modellierung sowie automatisierten Deployment- und Ausführungstechniken. Infolgedessen wird es einem größeren Nutzerkreis ermöglicht Anwendungen für vernetzte Fahrzeuge aufzusetzen und zu modifizieren, da Experten auf diesem Gebiet ohne Softwarekenntnisse ebenfalls befähigt werden.

Gliederung

Im Folgenden wird der Aufbau dieser Arbeit beleuchtet. Zu Beginn werden in Kapitel 2 die **Grundlagen** vorgestellt, um ein allgemeines Verständnis für die Themengebiete vernetzte Fahrzeuge, Datenpipelines und Virtualisierung zu schaffen.

Anschließend gibt Kapitel 3 einen Überblick über relevante **Verwandte Arbeiten**.

Nachfolgend werden in Kapitel 4 die definierten **Anforderungen** vorgestellt, welche für eine spätere Evaluation dienen. Zu diesen gehören Skalierbarkeit, Heterogenität, Portabilität, Erweiterbarkeit, Robustheit/Verfügbarkeit und Automatisierbarkeit.

Kapitel 5 beleuchtet das erarbeitete **Deploymentkonzept für verteilte Datenpipelines**. Dieses ist in drei Bereiche unterteilt und enthält ein entwickeltes Konzept für die Modellierung, das Deployment und die Ausführung einer Datenpipeline im Kontext von Software-Defined-Car-Anwendungen.

Darauffolgend wird in Kapitel 6 die **Prototypische Implementierung des Deploymentkonzepts** präsentiert. Das Kapitel wird mit einer Übersicht eingeleitet, in welcher ein Anwendungsszenario, verwendete Technologien und die zugrundeliegende Softwarearchitektur vorgestellt werden. Analog zum Konzept folgt anschließend eine Unterteilung der entwickelten Lösung in die Bereiche Modellierung, Deployment und Ausführung.

Die aufgestellten Anforderungen werden anhand des implementierten Prototyps in Kapitel 7 evaluiert. Die **Evaluation** setzt sich mit der kritischen Betrachtung aller Anforderungen auseinander und gibt Aufschluss über die Limitationen des Prototyps.

Zusammenfassung und Ausblick werden abschließend in Kapitel 8 behandelt.

2 Grundlagen

In diesem Kapitel werden die Grundlagen der vorliegenden Arbeit erläutert. Abschnitt 2.1 beleuchtet zunächst das Konzept des vernetzten Fahrzeugs. Anschließend werden Datenpipelines in Abschnitt 2.2 vorgestellt. Abschnitt 2.3 beschäftigt sich mit den Konzepten der Virtualisierung und gibt Einblicke in das Thema der Container-Orchestrierung.

2.1 Vernetzte Fahrzeuge

Die Komplexität und der Funktionsumfang von Kraftfahrzeugen haben sich über die Jahre stark verändert. Ursprünglich waren das Fahren und später die Sicherheit ausschlaggebende Faktoren bei der Entwicklung eines neuen Fahrzeugs. Inzwischen zählen jedoch ebenfalls Komfort, Unterhaltung und Assistenzsysteme zu den festen Bestandteilen von Neufahrzeugen. Um all diese neuen Herausforderungen bewältigen zu können, sind ECUs in Fahrzeugen verbaut. ECUs sind eingebettete Computer, welche eine oder mehrere Funktionen des Fahrzeugs steuern. Während vor einigen Jahren noch zwischen 30 und 70 ECUs in modernen Fahrzeugen verbaut waren, sind mittlerweile schon über 100 solcher Steuergeräte vorzufinden [HKGF09; SNA+13].

Die Daten, welche an den ECUs anfallen, sind jedoch nicht nur innerhalb des Fahrzeugs relevant. Mittels fortschrittlicher Kommunikations- und Informationstechnologie, ist es möglich mit den anfallenden Daten einen größeren Mehrwert zu schaffen. Innerhalb eines sogenannten Intelligent Transportation System (ITS) wird es vernetzten Fahrzeugen ermöglicht mit Komponenten wie anderen Verkehrsteilnehmern sowie der Infrastruktur zu kommunizieren [Dim11]. Das Ziel eines solchen ITS ist es diverse Verkehrsprobleme zu lösen. Dazu zählen Verkehrsüberlastung, Sicherheit, Verkehrseffizienz und Umweltschutz [FJM+01]. Um diese Probleme lösen zu können, besteht ein ITS in der Regel aus den folgenden vier Komponenten, welche im Anschluss näher erläutert werden [SLY+16]:

- Onboard Units (OBUs)
- Vulnerable Road Users (VRUs)
- Roadside Units (RSUs)
- ITS Server

Ein VF ist mit einer OBU ausgestattet. Dies ist ein eingebautes Stück Hardware mit entsprechender Firmware und deckt hauptsächlich zwei Anwendungsfälle ab [OWMW15]:

(i) Die OBU ist zuständig für die Kommunikation eines Fahrzeugs mit anderen Entitäten in einem externen Netzwerk.

(ii) Das Routing von und zu Fahrzeug-internen Geräten, wie beispielsweise den ECUs innerhalb eines Fahrzeugs, wird durch die OBU gehandhabt.

Die VRUs innerhalb eines ITS entsprechen den verwundbarsten Verkehrsteilnehmern. Dazu zählen beispielsweise Fußgänger, Radfahrer und Motorradfahrer. Sewalkar und Seitz [SS19] sehen es als notwendig an, diese Gruppe gesondert in einem ITS zu behandeln. Zur Integration von VRUs schlagen die Autoren vor, essenzielle Daten wie GPS-Koordinaten über das Smartphone des Verkehrsteilnehmers an das Netzwerk zu übermitteln. Dadurch könne die Ortungsgenauigkeit bei der Unfallerkennung zwischen Fahrzeug und VRU deutlich erhöht werden.

Den nächsten wichtigen Bestandteil eines ITS bilden RSUs. Diese Komponenten sind stationäre Geräte die an Straßen oder Wegen platziert werden und als Sender und Empfänger fungieren. Jedes VF kommuniziert mit den benachbarten RSUs. Dabei informiert es die RSU über den eigenen Zustand mit Informationen über Standort, Geschwindigkeit und Fahrtziel [OWMW15]. Umgekehrt kann aber auch das Fahrzeug von der RSU über den Zustand des nächsten Straßenabschnitts informiert werden.

Zusätzliche Informationen von VRUs und Sensoren, wie Kameras und Induktionsschleifen entlang der Fahrbahn, können ebenfalls empfangen werden. Mithilfe dessen ist die RSU laut Seo et al. [SLY+16] in der Lage die lokale Topologie zu analysieren und mit anderen Verkehrsteilnehmern zu teilen. Beispielsweise können dadurch Manöver oder Routen der umliegenden Fahrzeuge koordiniert werden, wenn dies sinnvoll ist [WMG17].

OBUs und RSUs ermöglichen demnach die gegenseitige Kommunikation zwischen Fahrzeugen, sowie zwischen Fahrzeug und Infrastruktur. Solch ein Netzwerk wird Vehicular Ad Hoc Network (VANET) genannt und bildet eine wichtige Grundlage für ITSs [LW07; YMF06]. Hartenstein und Laberteaux [HL08] erläutern in Ihrer Arbeit, dass diese hochdynamischen Netzwerke bereits eine Möglichkeit zur Erhöhung der Verkehrssicherheit und Verkehrseffizienz darstellen, das Potenzial aber noch weiter reicht.

Neben den flüchtigen Verbindungen zu den bisher genannten Komponenten im begrenzten Signalfeld einer RSU, ist diese zusätzlich mit einem zentralen ITS Server verbunden. Laut Seo et al. [SLY+16] ist es seine Aufgabe, eine zentralisierte Steuerung für alle Teilnehmer des Netzwerks sowie Verkehrs-, Straßen- und Serviceinformationen bereitzustellen.

Laut Chen et al. [CHS+17] lässt sich ein ITS in Kombination mit dem Konzept des Vehicle-to-Everything (V2X) weiter ausbauen und verbessern. Diverse Autoren unterteilen V2X in verschiedene Sub-Typen, wie in Abbildung 2.1 dargestellt ist. Darunter zählen [CHS+17; SS19; WMG17]:

- Vehicle-to-Vehicle (V2V)
- Vehicle-to-Pedestrian (V2P)
- Vehicle-to-Infrastructure (V2I)

In den folgenden Abschnitten wird ein Verständnis für diese Typen geschaffen, um besser zwischen ihnen differenzieren zu können.

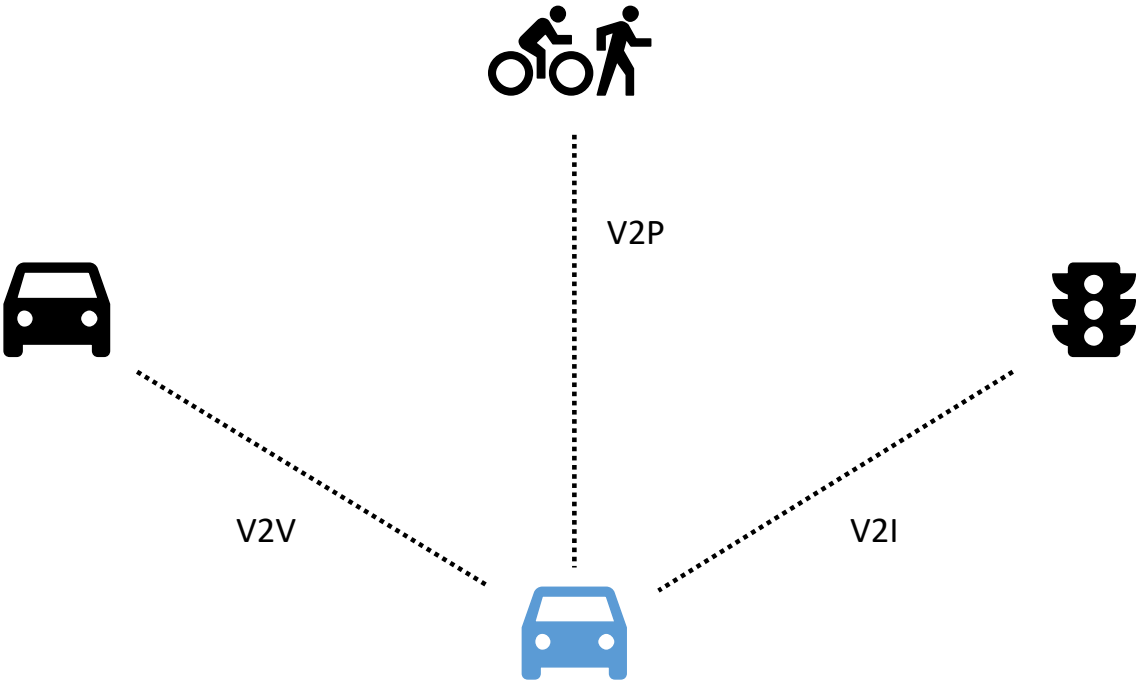


Abbildung 2.1: Darstellung verschiedener Sub-Typen der Vehicle-to-Everything-Kommunikation.

2.1.1 Vehicle-to-Vehicle

Der erste Bestandteil von V2X ist die Vehicle-to-Vehicle Kommunikation. Während Fahrzeuge verschiedene drahtlose Technologien zum Informationsaustausch mit anderen Geräten nutzen, wird für V2V häufig Dedicated Short-Range Communication (DSRC) genutzt. DSRC wurde entwickelt, um eine Vielzahl von Anwendungen auf der Grundlage von Fahrzeugkommunikation zu unterstützen [Ken11]. Für den physischen, sowie für den MAC-Layer wird hierbei IEEE 802.11p genutzt [IEE10]. Dies ist eine modifizierte Version des bekannten IEEE 802.11 (WiFi) Standards [IEE20]. Der Standard 802.11p gehört zu Wireless Access in Vehicular Environments (WAVE) [IEE19], einer Anwendung des DSRC, welche speziell an die Anforderungen eines VANET angepasst ist.

Die bislang häufig verwendeten Original Equipment Manufacturer (OEM) Lösungen nutzen eingebette Systeme wie Kameras und Sensoren, um einem Fahrzeug die Beobachtung und Analyse der Umgebung zu ermöglichen. Diese Systeme sind zwar sehr nützlich, jedoch sind die Sensoren nicht in der Lage versteckte Objekte oder abrupte Bewegungen anderer Fahrzeuge zu erkennen. Mit V2V ist es hingegen möglich, solche Informationen frühzeitig von einem zum anderen Fahrzeug weiterzuleiten. Zeadally et al. [ZGC19] sehen diese Lösung als wirksamer an, vor allem in Bezug auf Aspekte wie Unfälle, Wetter, Straßensperrungen und Verkehr. Die Autoren schreiben darüber hinaus, dass V2V dazu beitragen kann Verkehrsstaus zu verringern, sowie das Fahrerlebnis und die Sicherheit von Fahrgästen und Fahrern drastisch zu erhöhen. Die zur Zeit größten Faktoren für das schnelle Wachstum der V2V-Kommunikation sind Sicherheitsbedenken, rasche Fortschritte in der drahtlosen Technologie und eine starke Nachfrage nach Innovationen und technologischen Verbesserungen im Automobilsektor [ZGC19].

Molisch et al. [MTKM09] zufolge ist die Vision von V2V, dass alle zukünftigen Fahrzeuge Sensordaten sammeln und Informationen über die Verkehrsdynamik untereinander austauschen. Jedes Fahrzeug kann so Informationen von anderen Fahrzeugen empfangen, um die Funktionalität seines Bremssystems zu verbessern, Airbag-Funktionen zu verbessern und den Kraftstoffverbrauch, sowie die benötigte Fahrzeit zu reduzieren [MTKM09]. Eichler [Eic07] zufolge ist das Hauptziel des Einsatzes dieser Technologie in Fahrzeugen, die Sicherheit der Fahrgäste zu verbessern und die Zahl der Verkehrstoten zu verringern.

Neben diesen positiven Eigenschaften, gibt es im Feld von V2V jedoch auch einige Herausforderungen. Während vernetzte Fahrzeuge immer abhängiger von Informationstechnologie werden, steigt auch die Anfälligkeit für Angriffe und böswillige Manipulation [ZGC19]. Neben der Sicherheit, spielt auch der Kostenfaktor eine Rolle, da jedes Fahrzeug mit entsprechender Technologie ausgerüstet werden müsste.

2.1.2 Vehicle-to-Pedestrian

Ein Zusammenstoß zwischen Fahrzeug und einem VRU, beispielsweise einem Fußgänger oder Fahrradfahrer, führt häufig zu einem tödlichen Unfall. Nach Angaben der Weltgesundheitsorganisation ist die Anzahl tödlicher Verkehrsunfälle mit VRUs intolerabel hoch. Die Organisation gibt an, dass fast die Hälfte aller tödlichen Unfälle verwundbare Verkehrsteilnehmer beinhalten [Org+15]. Die Hauptursache für solch ein Unglück ist das mangelnde Vermögen der Verkehrsteilnehmer, entgegenkommende Gefahren rechtzeitig zu erkennen und wahrzunehmen, sodass Maßnahmen zur Unfallvermeidung ergriffen werden können [AMS+14].

Diese Punkte sind ein Indikator für die Notwendigkeit neuer Technologien, welche in der Lage sind VRUs besser zu schützen. Wie Gandhi und Trivedi [GT07] berichten, wurde dieses Thema in der Forschung bereits seit längerem behandelt. Speziell der Erkennung von Fußgängern und Vorhersage möglicher Kollisionen mithilfe von Sensoren und Computer-Vision wurde viel Aufmerksamkeit gewidmet [GT07]. Der Konsens ist, dass diese sensorbasierten Ansätze bei schlechten Sichtverhältnissen jedoch nicht gut abschneiden [BSN14; DF10; GT07]. Nachts, bei mangelhaften Wetterbedingungen, hoher Entfernung zwischen Fahrzeug und VRU oder bei nicht vorhandener Sichtverbindung hat dieser Ansatz seine Nachteile.

Aus diesem Grund unterscheiden Bagheri et al. [BSN14] zwischen zwei Fällen, die zur Verringerung des Unfallrisikos notwendig sind. Im ersten Fall besteht eine direkte Sichtverbindung zwischen VRU und dem vernetzten Fahrzeug. Im zweiten Fall gibt es keine direkte Sichtverbindung zwischen den beiden Entitäten, sodass das Vermeiden eines Unfalls aus Sicht des Fahrzeugs deutlich erschwert ist. Wenn ein Fußgänger beispielsweise die Straße zwischen parkenden Autos überquert, ist die Reaktionszeit für das entgegenkommende Fahrzeug deutlich geringer, als wenn vorab eine Sichtverbindung besteht.

Um die Verkehrssicherheit für VRUs zu erhöhen, schlagen diverse Autoren vor, Fußgänger mithilfe von Smartphones in das ITS einzubetten [BSN14; DF10; SNH08]. Solch ein funkbasiertes Kollisionsvermeidungssystem benötigt keine Sichtverbindung für die Kommunikation zwischen Fahrzeug und VRU. Zusätzlich kann es auf der nahezu universell verfügbaren Infrastruktur des Global System for Mobile Communications (GSM) beziehungsweise Universal Mobile Telecommunications System (UMTS) aufgebaut werden [DF10].

In der Literatur wird die V2P-Kommunikation in drei verschiedene Modi unterteilt. Fahrzeuge und VRUs können *direkt* oder *indirekt* miteinander kommunizieren, außerdem ist auch ein *hybrider* Ansatz für die Kommunikation möglich [DF10; SS19].

Im *direkten* Kommunikationsmodus erfolgt der Nachrichtenaustausch zwischen vernetztem Fahrzeug und Smartphone eines VRU ohne eine dazwischenliegende Entität. Von den drei Modi ist dies in der Regel die schnellste Möglichkeit Nachrichten auszutauschen, da nur eine Verbindung aufgebaut werden muss. Zusätzlich ist die Latenz am niedrigsten, sodass sich *direkte* Kommunikation gut für sicherheitsrelevante Anwendungsfälle eignet [SS19]. Negative Aspekte des *direkten* Kommunikationsmodus sind jedoch die niedrige Reichweite der Kommunikation, sowie der hohe Energieverbrauch [DF10; SS19]. Zusätzlich müssen alle Geräte mit der gleichen Art von Kommunikationstechnologie ausgestattet sein [SS19].

Bei der *indirekten* Kommunikation werden Nachrichten zwischen Fahrzeug und VRU über die Infrastruktur, beispielsweise eine RSU, geleitet. Im Gegensatz zum *direkten* Modus, ist diese Lösung langsamer im Austausch von Nachrichten, da die Nachricht erst vom Fahrzeug an die Infrastruktur und anschließend an einen VRU gesendet werden muss. Entsprechend erhöht sich ebenfalls die Latenz. Dieser Ansatz hat jedoch auch einige wesentliche Vorteile gegenüber der *direkten* Kommunikation. Die einzelnen Entitäten können miteinander kommunizieren, obwohl sie mit verschiedenen Kommunikationstechnologien ausgestattet sind. Beispielsweise könnte ein Fahrzeug seine Nachrichten über 802.11p oder Wi-Fi versenden, während das Mobiltelefon über das Mobilfunknetz in das ITS integriert wird. Durch die Infrastruktur würde sich auch der Kommunikationsradius erweitern, sodass Nachrichten über größere Distanz ausgetauscht werden können. Auch der Energieverbrauch würde von den Endgeräten auf die Infrastruktur verlagert werden, wo in der Regel mehr Ressourcen zur Verfügung stehen [SS19].

Der *hybride* Ansatz versucht alle positiven Aspekte der beiden vorherigen Modi zu kombinieren. Durch die *direkte* Kommunikation über ein ad-hoc Netzwerk und die *indirekte* Kommunikation via Infrastruktur, könnten Kommunikationsreichweite und Latenz optimiert werden, so Sewalkar und Seitz [SS19]. Die Autoren schreiben weiterhin, dass dieser Modus die Komplexität des Systems erhöht. Ein Grund dafür ist die Notwendigkeit alle Entitäten im ITS mit verschiedenen Kommunikationstechnologien auszustatten.

2.1.3 Vehicle-to-Infrastructure

Neben der Kommunikation zwischen VF und anderen Fahrzeugen, sowie zwischen VF und VRUs, spielt auch die Kommunikation mit der Infrastruktur bei SDCs eine wichtige Rolle. Barrachina et al. [BGF+13] zufolge, entstehen zwei große Vorteile durch das Hinzufügen von Infrastruktur in ein ITS bezogen auf die Verkehrssicherheit:

(i) Die Infrastruktur kann den Internetzugang für Fahrzeuge bereitstellen.

(ii) Komponenten der Infrastruktur sind in der Lage empfangene Nachrichten zu broadcasten, wodurch mehr Fahrzeuge mit einer Nachricht erreicht werden können.

Damit adressiert V2I eines der größten Probleme von V2V-Kommunikation, nämlich die Unregelmäßigkeit eines VANET, insbesondere wenn die Fahrzeugdichte gering ist [NRSG17].

Ebenso wie bei der V2V-Kommunikation, wird bei V2I DSRC genutzt, um die notwendigen Komponenten in das Netzwerk zu integrieren [CKP+11]. Typischerweise zählen RSUs zu diesen Komponenten. Diese Kommunikationsknoten können Verbindungen zu Fahrzeugen mit einer Entfernung bis zu einem Kilometer aufrechterhalten [CKP+11]. Eine OBU innerhalb eines vernetzten Fahrzeugs verbindet sich mit der RSU und die beiden Komponenten können Daten austauschen, solange sich das VF innerhalb der Reichweite befindet. Neben RSUs können zusätzliche Komponenten wie Ampeln und Induktionsschleifen Informationen an das ITS senden oder empfangen, wenn diese über die entsprechenden Technologien verfügen [UJ16].

Durch die Einbindung der Infrastruktur in das ITS ergeben sich einige zusätzliche Anwendungsfälle. Besonders nützlich ist die V2I-Kommunikation in den Bereichen Sicherheit, der effizienteren Nutzung von Straßen und Kreuzungen, Infotainment-Anwendungen, sowie Zahlungen (beispielsweise Maut) [NRSG17].

Popescu et al. [PSA+17] beschreiben in ihrer Arbeit eine Möglichkeit die Sicherheit auf Autobahnen durch V2I zu erhöhen. Dabei sollen Verkehrsunfälle automatisch erkannt werden, sodass nachkommende Fahrzeuge frühzeitig gewarnt werden können. Mit ihrem Konzept entwickelten die Autoren einen Prototyp und erreichten eine Reduktion der Stau-Höhepunkte, als auch die schnellere Auflösung von Staus auf den Straßen.

Milanes et al. [MVG+12] erarbeiteten einen Ansatz für die Entwicklung eines vollständigen Verkehrssteuerungssystems. In ihrer Forschung zeigten sie, dass V2I es ermöglicht den Verkehrsfluss in urbanen Gebieten zu regeln. Zusätzlich konnten die Forscher zeigen, dass ihr Ansatz zur Vermeidung von Unfällen beiträgt, indem es den Fahrer vor möglichen Kollisionen warnt. Als Infrastruktur-Komponente wurde hierbei eine RSU genutzt, welche den gesamten Bereich einer Teststrecke abdeckte und so mit den Fahrzeugen kommunizieren konnte.

Diverse Arbeiten behandeln die Verzögerung des Verkehrs, sowie erhöhten Kraftstoffverbrauch in urbanen Gebieten durch Ampelschaltungen [BI16; CWG13; UJ16]. Speziell im Feld der adaptiven Signalsteuerungsmethoden wurde viel geforscht. Durch die Beeinflussung von Ampelschaltungen können sowohl Staus reduziert, als auch die durchschnittliche Wartezeit an einer roten Ampel minimiert werden [BI16]. In ihrer Arbeit zeigten Cai et al. [CWG13] anhand einer entwickelten adaptiven Signalsteuerung, dass die Fahrtzeit und Anzahl der Stopps gegenüber Benchmark-Methoden mittels V2I verringert werden kann. Die Autoren berichten für beide Fälle eine Verbesserung von über 10% gegenüber herkömmlichen Methoden [CWG13].

Butakov und Ioannou [BI16] präsentieren in ihrer Arbeit einen alternativen Ansatz für die gleiche Herausforderung. Ihr Ziel ist es, den Kraftstoffverbrauch zu senken und die Wartezeit zu verkürzen, jedoch ohne in die Steuerung der Ampelschaltungen einzugreifen. Ihr Ansatz ist es das Problem individuell anzugehen, indem einzelne VF kontrolliert werden. V2I ermöglicht den Zugriff auf Informationen über den Standort und Zustand von Ampeln an Kreuzungen, die für die Route des Fahrers relevant sind. Durch das von den Autoren entwickelte System, kann mithilfe dieser Informationen die optimale Fahrgeschwindigkeit gefunden, und unnötige Beschleunigung oder Abbremsung minimiert werden. Mit diesem Ansatz konnten die Autoren eine durchschnittliche Verbesserung des Energieverbrauchs von 29% erzielen und damit die Wirksamkeit des Algorithmus zeigen [BI16].

Während V2I viele neue Anwendungsfälle zur Verbesserung des ITS ermöglicht, entstehen durch diese Technologie auch neue Herausforderungen. Die zuvor angesprochenen RSUs, welche für diverse Szenarien notwendig sind, stellen eine wertvolle Ressource im ITS dar. Grund dafür sind die hohen Installationskosten und die begrenzte Anzahl von RSUs in Vororten und Gebieten mit geringer Bevölkerungsdichte [BGF+13].

Weiterhin hängt die Wirksamkeit eines ITS stark davon ab, wie effizient RSUs eingesetzt werden. Neben den angesprochenen Herausforderungen ist das Deployment von RSUs selbst in urbanen Gebieten herausfordernd. Hindernisse wie Gebäude und Bäume können den empfangenen Signalpegel im Frequenzbereich von DSRC stark beeinflussen [GSB12]. Aus diesen Gründen stellt die Platzierung bzw. das Deployment von RSUs einen wichtigen Punkt in der Forschung von V2I dar und wird häufig untersucht [BGF+13; GSB12; KK10; LK10].

2.2 Datenpipelines

Eine Datenpipeline stellt eine Methode zur Verarbeitung von Daten dar. Durch eine Reihe von chronologischen Schritten können Daten hierbei verändert werden. Neben Quellen und Senken, welche die Entstehung und das Ende einer Pipeline darstellen, gibt es eine beliebige Anzahl an Zwischenschritten. Typischerweise ist die Ausgabe jedes Schrittes die Eingabe des nächsten, ausgenommen hiervon sind Datensenken, da anschließend keine weiteren Verarbeitungsschritte innerhalb der Pipeline erfolgen.

Häufig wird das Konzept von Datenpipelines im Big-Data-Bereich vorgefunden. Einem Bericht von McKinsey & Company [McK14] zufolge verarbeiteten Neufahrzeuge im Jahr 2014 bereits bis zu 25 Gigabyte an Daten. In einer einzelnen Stadt sind typischerweise mehrere tausend Fahrzeuge zur gleichen Zeit in Bewegung. Durch die steigende Anzahl an VFs und dem großen Interesse an ITS wird dieser Sektor zukünftig noch mehr Daten produzieren. Zhu et al. [ZYW+19] zufolge fallen ITSs somit in den Bereich Big Data.

Big Data lässt sich in drei Charakteristiken unterteilen [Cur16; SS13; ZYW+19]:

- Volume (Menge der Daten)
- Velocity (Geschwindigkeit der Daten)
- Variety (Bandbreite an Datentypen/-quellen)

Volume sorgt für Komplexität da die außerordentlichen Datenmengen im Big-Data-Bereich Terabyte und Petabyte übersteigen können. Die zunehmende Menge an Daten übersteigt die traditionellen Speicher- und Analysetechniken [SS13].

Velocity muss nicht nur für Big Data, sondern für alle Prozesse beachtet werden. Die Bewältigung von Datenströmen mit hoher Frequenz eingehender Echtzeit-Daten, beispielsweise Sensordaten eines VF, müssen ebenfalls spezielle Aufmerksamkeit erhalten [Cur16].

Variety sorgt dafür, dass Datentypen nicht gleich behandelt werden können und somit die Komplexität erhöht wird. Unterschieden wird hier zwischen strukturierten, semi-strukturierten und unstrukturierten Daten [SS13].

Analyseansätze aus dem Bereich Big Data skalieren in Bezug auf die Menge und Geschwindigkeit der Daten. Dies ist möglich, indem sich einzelne Verarbeitungsschritte auf ein Cluster, das heißt einen Verbund aus Maschinen für das Speichern und Rechnen, verlassen [HWCL14]. Resultate einzelner Maschinen werden anschließend aggregiert und zusammengefasst. Um verschiedenen Anwendungen abzudecken, können Analyseansätze im Big-Data-Bereich in zwei Kategorien unterteilt werden, *Batch*- und *Stream*-Analytics [AGP17].

Batch Analytics beziehen sich dabei auf historische Daten und werden verwendet, um Aktivitäten und Informationen aus der Vergangenheit zu untersuchen und Erkenntnisse daraus zu gewinnen [AGP17]. *Stream Analytics* behandeln dagegen Echtzeitdaten und werden daher für Anwendungen mit niedriger Latenz bevorzugt [AGP17]. Anstatt Daten in Batches zu laden, bewegen Streaming-Pipelines Daten kontinuierlich in Echtzeit von der Quelle zum Ziel. Beide Ansätze können dabei als Datenpipeline angesehen werden. Diese Big-Data-Pipelines zerlegen komplexe Analysen großer Datenmengen in eine Reihe von einfacheren Aufgaben mit unabhängig voneinander abgestimmten Komponenten für jede Aufgabe [RSGJ13].

Durch die Nutzung von Datenpipelines im ITS-Bereich können gezielt die einzelnen Charakteristiken von Big Data behandelt werden. Nichtsdestotrotz ergeben sich Herausforderungen im Umgang mit den Daten. Zum einen ist die Speicherkapazität nicht außer Acht zu lassen. Eine komfortable Lösung ist das Aufsetzen eines grenzenlosen Speichermediums, bspw. einer Datenbank oder eines Data Lakes, über einen Cloud-Provider [SAK+13]. Bei den anfallenden Datenmengen kann diese Option jedoch schnell sehr kostspielig werden. Falls sicherheitsrelevante Daten versendet werden, ist ein Cloud-Provider eventuell keine Möglichkeit. Eine Alternative wäre das Aufsetzen einer privaten Cloud, mit welcher die Daten sicher wären. Gleichzeitig ist die Latenz bei dieser Lösung in der Regel geringer, was bei Anwendungen mit hohem Datenverkehr von Vorteil sein kann [SAK+13]. Dafür muss die private Cloud eigenständig verwaltet und gewartet werden.

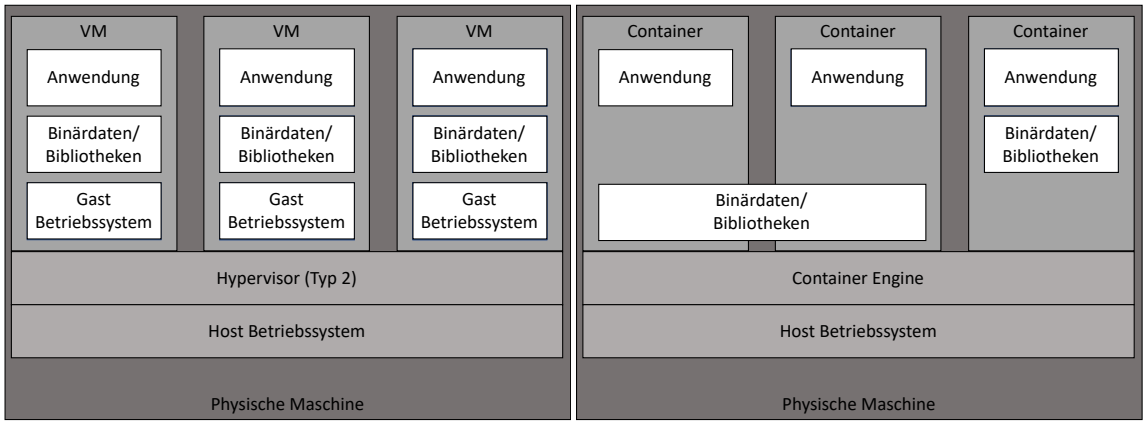
Während Datenpipelines für Big Data also einen großen Mehrwert schaffen, müssen die verschiedenen Aspekte wie Erhebung, Schutz, Speicherung und die Verarbeitung von Daten an den jeweiligen Anwendungsfall angepasst werden [ZYW+19].

2.3 Virtualisierung

Durch die Abstraktion von der Hardware bietet Virtualisierung viele Vorteile. Sie ermöglicht das einfache Migrieren, sowie eine bessere Ressourcennutzung, was zu geringeren Kosten und Energieeinsparungen führt [Ede16]. Im ersten Unterabschnitt werden die Arten von Virtualisierung vorgestellt, sowie deren Vorteile miteinander verglichen. Anschließend wird in Abschnitt 2.3.2 auf Container-Orchestrierung eingegangen. Dies ist eine Technologie zur Verwaltung von Containern, welche eine Virtualisierungsart darstellen.

2.3.1 Arten der Virtualisierung

Lange Zeit konnte der Begriff Virtualisierung mit dem Hypervisor-basierten Ansatz gleichbedeutend genutzt werden. In den letzten Jahren nahm die Virtualisierung durch Container dagegen stark zu. Während beide Technologien einige Vorteile bieten, haben sie ihre eigenen Vorzüge und Unterschiede, auf welche im Folgenden eingegangen wird.



(a) Hypervisor-basierter Virtualisierungsansatz. **(b)** Container-basierter Virtualisierungsansatz.

Abbildung 2.2: Architekturvergleich von Hypervisor- und Container-basierten Virtualisierungsansätzen, basierend auf [LKLA17].

Die Hypervisor-basierte Virtualisierung wird seit mehreren Dekaden genutzt [Ede16]. Der Hypervisor ist eine Software, welche die Abstraktion von der Hardware ermöglicht. Jedes Stück Hardware, das für die Ausführung von Software benötigt wird, muss vom Hypervisor emuliert werden [Ede16]. Diese Art von Technologie wird in Cloud Anwendungen häufig verwendet, wodurch virtuelle Maschinen als Service zur Verfügung gestellt werden können [XYP14]. Dabei verwaltet der Hypervisor die physischen Rechenressourcen und erstellt isolierte Teile der Hardware für die Erstellung von Virtual Machines (VMs) [Mer+14]. Da dieser Ansatz nur Zugriff auf die physischen Ressourcen erlaubt, benötigt jede einzelne VM die komplette Implementierung eines Gast-Betriebssystems, einschließlich Binärdateien und Bibliotheken für Anwendungen [Ber14]. Dies ist in Abbildung 2.2a dargestellt. Prominente Beispiele für die Virtualisierung mit dem Hypervisor-basierten Ansatz sind Hyper-V[1], VMware[2] und VirtualBox[3].

Um den Overhead der Hypervisor-basierten Virtualisierung zu verringern, haben Forscher und Praktiker vor kurzem begonnen, eine alternative und leichtgewichtige Lösung zu fördern, nämlich die Container-basierte Virtualisierung [LKLA17]. Im Gegensatz zur Lösung auf Hardware-Ebene realisieren Container die Virtualisierung auf Betriebssystemebene und nutzen isolierte Teile des Host-Betriebssystems [Ber14]. Dadurch müssen Container keine Hardware emulieren wie beim Hypervisor-Ansatz, sondern nutzen die Hardware des Host-Systems [Ede16]. Wie man in Abbildung 2.2b sieht, greifen Anwendungen auf derselben physischen Maschine mit diesem Ansatz auf das gleiche Betriebssystem zu. Zusätzlich ist es möglich, dass verschiedene Anwendungen auf die gleichen Binärdateien und Bibliotheken zugreifen. Docker[4] ist zur beliebtesten Container-Lösung geworden, da es die Verwaltung von Containern mithilfe eines einheitlichen Tool-Satzes und eines Application Programming Interface (API) erheblich erleichtert [Pah15].

[1] Hyper-V: https://docs.microsoft.com/de-de/virtualization/hyper-v-on-windows/
[2] VMware: https://www.vmware.com/
[3] VirtualBox: https://www.virtualbox.org/
[4] Docker: https://www.docker.com/

Sowohl der Hypervisor-, als auch der Container-Ansatz bieten ihre Vorteile. Beide ermöglichen es, bestehende virtuelle Maschinen oder Container einfach zu migrieren. Der größte Unterschied hierbei ist, dass die Hypervisor-basierte Virtualisierung es ermöglicht einen gesamten Computer zu emulieren. Dadurch kann ein Rechner mit einem anderen Betriebssystem als dem zugrundeliegenden betrieben werden, was speziell für die Ausführung von Anwendungen auf anderen Plattformen von großem Vorteil sein kann. Im Gegensatz hierzu, nutzen Container-basierte Lösungen Funktionen des Host-Betriebssystem Kernels, um eine isolierte Umgebung für Prozesse zu schaffen [Ede16]. Aus diesem Grund ist Software innerhalb eines Containers abhängig von dem zugrunde liegenden Betriebssystem und der CPU-Architektur. Da jedoch kein Gast-Betriebssystem notwendig ist, werden initial weniger Platz und CPU-Ressourcen von einem Container benötigt [SHM+14]. Zusätzlich entfällt auch der Bedarf eines Hypervisors, sodass Container gegenüber VMs ressourcen- und zeit-effizienter sind [LKLA17]. Da keine Hardware emuliert und kein komplettes Betriebssystem gebootet werden muss, können Container in wenigen Millisekunden starten und sind effizienter als klassische virtuelle Maschinen.

2.3.2 Container-Orchestrierung

In der heutigen Zeit bestehen große Anwendungen bereits aus mehreren hunderten von Microservices. Eine Möglichkeit diese bereitzustellen ist die Virtualisierung mittels Container-Technologie. Container-Lösungen in solch großem Rahmen anzuwenden kann sich jedoch als zeitaufwändig herausstellen, da jeder einzelne Container verwaltet werden muss. Aus diesem Grund wurde eine neue Ebene in der Containerisierung eingeführt, nämlich die Container-Orchestrierung.

Eine Container-Orchestration-Engine automatisiert die Bereitstellung, Skalierung und Verwaltung von Containern, einschließlich der Planung von Ressourcen, der Koordination von und der Kommunikation zwischen Microservices [JBB+19]. Zu den am weitesten verbreiteten Beispielen für solch eine Engine zählen Docker Swarm[5], Kubernetes[6] und Mesos[7] [HBW+17; TLP+19].

Jawarneh et al. [JBB+19] zufolge, lässt sich die Architektur eines Container-Orchestrator in drei verschiedene Ebenen unterteilen, nämlich:

- Ressourcenmanagementebene
- Planungsebene
- Servicemanagementebene

Die Ressourcenmanagementebene verwaltet Ressourcen auf der niedrigsten Ebene. Dazu zählen beispielsweise Speicher, CPU/GPU, Festplattenspeicher, Datenträger, Port und IP. Das Hauptziel dieser Ebene ist es, die Auslastung zu maximieren und Konflikte zwischen Containern, die um Ressourcen konkurrieren, zu minimieren.

In der Planungsebene geht es darum, Cluster-Ressourcen effizient zu nutzen. Durch den Nutzer spezifizierte Angaben werden als Eingabe genommen, mithilfe welcher der Orchestrator die Container entsprechend deployen kann. Der Nutzer kann zum Beispiel die Platzierung eines

[5] Docker Swarm: https://docs.docker.com/engine/swarm/
[6] Kubernetes: https://kubernetes.io/
[7] Mesos: https://mesos.apache.org/

Containers beeinflussen und die Anzahl der Replicas eines Containers angeben. Über *readiness-checks* kann dafür gesorgt werden, Container nur zu beachten, wenn diese bereit sind. Außerdem ist es möglich, das *Rescheduling* zu kontrollieren, womit man Container im Fehlerfall automatisch erneut starten kann.

Die letzte der drei Ebenen beinhaltet die Funktionalität für das Verwalten von Ressourcen der höchsten Ebene. Mithilfe von *Labels* können Metadaten an Container angeheftet werden. Durch Gruppen beziehungsweise Namespaces verfügt ein Orchestrator über die Möglichkeit Container zu isolieren oder Multi-tenacy zu erlauben. Abhängigkeiten zwischen Containern, sowie die Lastverteilung für eingehende Nachrichten lassen sich durch die Verwendung von Container-Orchestration-Engines ebenfalls umsetzen.

3 Verwandte Arbeiten

In diesem Kapitel werden relevante wissenschaftliche Arbeiten vorgestellt, welche als Grundlage für das entwickelte Konzept dienen. Hierzu werden verschiedene Bereiche von verwandten Arbeiten betrachtet. Von Bedeutung sind für diese Arbeit speziell die Modellierung von Datenpipelines, Deploymentstrategien sowie Kommunikationsmöglichkeiten zwischen Services.

Hirmer [Hir18] behandelt in seiner Dissertation die anforderungsbasierte Modellierung und Ausführung von Datenflussmodellen. Für diesen Zweck stellt der Autor verschiedene Konzepte auf. Dazu zählt ein entwickeltes Modell für den Entwurf von Kontroll- und Datenfluss, welches auf dem Pipes-and-Filters-Muster basiert. Eine Erweiterung dieser Funktionalität schafft der Autor anhand der Einführung einer Anforderungspriorisierung. Zusätzlich wurde ein Konzept für die automatische Provisionierung der Ausführungsumgebung entwickelt. Dieses basiert auf der Topology and Orchestration Specification for Cloud Applications (TOSCA) und ermöglicht die Bereitstellung anwendungsspezifischer Komponenten, welche anschließend durch Plattform- und Infrastrukturkomponenten ergänzt werden. Darauf aufbauend entwickelte Hirmer ein Konzept für das Anschließen von Datenquellen. Durch die Nutzung einer Middlewarekomponente werden der automatische Anschluss der Datenquellen ermöglicht und zugleich der Datenzugriff vereinfacht. Als letzten Punkt beinhaltet die Dissertation ein Konzept für die automatisierte Ausführung der Datenverarbeitung und deren Optimierung.

Als Bestandteil von Hirmers Dissertation entwickelten Hirmer und Mitschang [HM16] eine Methode zur automatischen Provisionierung von Data-Mashups mit dem Namen *TOSCA4Mashups*. Diese basiert auf OpenTosca[1], einer TOSCA Laufzeitumgebung für das Provisionieren und Verwalten von Cloud-Anwendungen. Anhand der verwendeten Methode entstehen Vorteile wie eine bessere Skalierbarkeit, Verfügbarkeit und eine Kostenersparnis gegenüber bisherigen Ansätzen. Durch die Implementierung eines Prototyps konnten die Autoren außerdem eine generische Anwendbarkeit ihres Konzepts zeigen.

In seiner Masterarbeit entwickelte Kenzler [Ken21] ein Konzept für die Modellierung von Datenpipelines in Software-Defined-Car-Anwendungen und setzte dieses anhand eines Prototyps um. Der Fokus des Konzepts liegt auf der Modellierung von Datenflüssen in solchen Anwendungen und orientiert sich an der vorgestellten Dissertation von Hirmer. Kenzler setzt den Blickpunkt auf Datenquellen und Filter, also Datenverarbeitungsoperationen, während Datensenken vernachlässigt werden. Der resultierende Prototyp wurde in Form einer Webanwendung entwickelt. Diese erlaubt es Nutzern Modelle zu erstellen, indem Datenquellen und Filter erstellt und anschließend durch gerichtete Kanten verbunden werden, welche den Datenfluss zwischen Komponenten darstellen. Dieser Ansatz kann durch das Hinzufügen von Datensenken erweitert und als Basis für das Deployment und die anschließende Ausführung von Pipelines genutzt werden.

[1] OpenTosca: https://www.opentosca.org/

Sami et al. [SME20] stellen in Ihrer Arbeit ein Konzept für das Deployment containerisierter Micro-Services auf Fahrzeug-Bordcomputern vor. Das Ziel des Konzepts ist es, VFs mit der Cloud zu verbinden, um mit Komponenten außerhalb des Fahrzeugs kommunizieren zu können. Dies ist notwendig, da OBUs noch nicht über die erforderlichen Ressourcen verfügen, um VMs effizient herunterzuladen und zu starten. Um diese Limitation zu überwinden, schlagen die Autoren das Ausführen von containerisierten Anwendungen vor und nutzen Docker als Technologie. Für die Verwaltung von Fahrzeugflotten, bestehend aus OBUs mit laufenden Docker Containern, nutzen die Autoren Kubernetes als Orchestrator.

Das Deployment containerisierter Anwendungen mittels Orchestrator ist auch in weiteren verwandten Arbeiten die gewählte Lösung. So entwickelten Yang et al. [YSCH19] einen Ansatz zur Berechnung von kurzfristigen Verkehrsvorhersagen durch die Verwendung von Docker und Kubernetes. In einem Experiment mit mehreren VFs und einer RSU berechnet der resultierende Prototyp Vorhersagen über die Fahrzeuggeschwindigkeit und Standzeit an roten Ampeln. Verglichen mit bisher verwendeten Methoden zur Verkehrsvorhersage kann das Experiment nahezu gleichwertige Genauigkeiten erzielen, benötigt jedoch eine deutlich niedrigere Rechenkomplexität.

Eine weitere Herausforderung stellt die Kommunikation zwischen Komponenten innerhalb der Datenpipeline dar. In Ihrer Arbeit stellen Del Gaudio und Hirmer [DH19] einen Ansatz für die dezentrale Datenverarbeitung im IoT durch die Nutzung eines selbst entwickelten Nachrichtenübermittlungssystems vor. Anstatt Daten von einem IoT-Gerät an die Cloud zu schicken, diese dort zu verarbeiten und das Resultat anschließend wieder zurückzusenden, schlagen die Autoren eine alternative Methode für zeitkritische Anwendungen vor. Dabei sollen Daten möglichst nah an einem IoT-Gerät verarbeitet werden, wodurch sowohl Latenz als auch Kosten reduziert werden können. Durch eine prototypische Implementierung ihres Ansatzes konnten die Autoren zeigen, dass die dezentralisierte Datenverarbeitung umsetzbar ist und speziell in zeitkritischen Anwendungen für eine schnellere sowie robustere Verarbeitung sorgt.

Einige Lösungen nutzen MQTT[2] als Kommunikationstechnologie, welches häufig als der Standard für IoT-Messaging bezeichnet wird. Dhall und Solanki [DS17] stellen in Ihrer Arbeit einen IoT-basierten Ansatz für die vorausschauende Wartung von vernetzten Fahrzeugen vor. In Ihrem Konzept stellen die Autoren ein Anwendungsszenario vor, bei welchem Fahrzeuge als Produzenten und die Cloud als Konsument verschickter Daten fungieren. Daten welche ausgetauscht werden können den Standort und die Geschwindigkeit des Fahrzeugs, sowie den Status verschiedener Bauteile beinhalten. Anhand dieser Informationen werden anschließend analytische Berechnungen innerhalb der Cloud vorgenommen. Die Kommunikation zwischen Produzenten und Konsumenten wird im vorgeschlagenen Lösungsansatz mittels MQTT gehandhabt, da sich dieses für eingeschränkte Umgebungen eignet, in denen Geräte über begrenzte Verarbeitungs- und Speicherressourcen, sowie eine geringe Bandbreite verfügen. Weiterhin beschreiben die Autoren die Eignung von MQTT durch das leichtgewichtige Nachrichtenübermittlungsprotokoll sowie die Möglichkeit zur Nutzung des Publish-Subscribe-Musters.

Chowdhury et al. [CRR+18] reflektieren in Ihrer Arbeit über die Erkenntnisse einer deployten Testumgebung für VFs. Auf dem Campus der Clemson University setzten die Autoren eine kontrollierte Umgebung, bestehend aus drei RSUs und drei VFs auf. In einer Fallstudie wurden eine Anwendung zur Kollisionsvermeidung, sowie eine zweite Anwendung zur Prävention von Auffahrunfällen

[2] MQTT: https://mqtt.org/

durch die Erkennung eines Stauendes deployt. Die Grundlage beider Anwendungen bildet die Echtzeitanalyse anfallender Fahrzeugdaten. Um diese zu versenden, entschieden sich die Autoren ebenfalls für den Einsatz von MQTT als Messaging-Technologie. Diese Entscheidung begründen Sie durch den hohen Durchsatz, den niedrigen Stromverbrauch, die geringe Netzwerkauslastung sowie der niedrigen Latenz, welche mit MQTT einhergeht.

Eine Alternative zur Kommunikation mittels MQTT stellt die Verwendung von Apache Kafka dar. In Ihrer Arbeit stellen Du et al. [DCR+18] eine Strategie zur Erstellung eines effizienten und latenzarmen verteilten Nachrichtenübermittlungssystems für VF-Anwendungen vor. Diese Strategie ermöglicht die Aufnahme, Verarbeitung und Umwandlung von großen Datenmengen. Anhand eines Prototyps wurde die entwickelte Strategie anschließend evaluiert. Als verteiltes Nachrichtenübermittlungssystem wählten die Autoren Kafka, eine frei zugängliche Message-Broker-Plattform. Mithilfe des Prototyps konnten Du et al. zeigen, dass die durchschnittliche Latenz der Nachrichtenübertragung mit den minimalen Anforderungen des Verkehrsministeriums der Vereinigten Staaten für VF-Anwendungen übereinstimmt.

Einen ähnlichen Ansatz verfolgen Amini et al. [AGP17]. In Ihrer Arbeit entwickeln sie einen Entwurf für eine umfassende und flexible Architektur für die Echtzeit-Verkehrssteuerung. Als Grundlage definierten die Autoren eine Liste an Anforderungen, abgeleitet von bestehenden Verkehrssteuerungssystemen. Um die vorgeschlagene Architektur zu evaluieren, entwickelten die Autoren eine Prototyp-Plattform, bei welcher die Kommunikation zwischen den verschiedenen Bestandteilen ebenfalls mittels Kafka umgesetzt ist. Innerhalb einer Fallstudie betrieben die Autoren den Prototyp, um das Verhalten anhand einer Verkehrssimulation zu testen. Durch die Nutzung von Kafka konnten die Autoren zeigen, dass die Skalierbarkeit basierend auf steigenden Datenquellen und Datensenken umsetzbar ist, wodurch die Plattform eine große Bandbreite an Daten verarbeiten kann. Zusätzlich garantiert die gewählte Kommunikationstechnologie weitere Vorteile, wie die at-least-once Semantik bei der Nachrichtenübermittlung, das heißt die mindestens einmalige Zustellung von Nachrichten, ebenso wie das Tolerieren von Ausfällen einzelner Rechnerknoten durch Replikationsmechanismen.

Akbar et al. [AKCM17] nutzen ebenfalls Kafka als Kommunikationstechnologie für die Analyse komplexer IoT-Datenflüsse. Im Vergleich zu anderen verfügbaren Messaging-Systemen ist Kafka in der persistenten Speicherung der versendeten Nachrichten einzigartig. Darüber hinaus verfügt das System über eine außerordentliche Leistung in Anwendungsszenarien mit hohem Durchsatz [YQC+19]. In Ihrer Arbeit vergleichen Fu et al. [FZY21] bestehende Messaging-Systeme und wählen Kafka als beste Technologie für die Umsetzung von Anwendungen mit umfangreicher Datenerhebung und -analyse. Da diese Aspekte in VF-Anwendungen von großer Relevanz sind, soll Kafka in dieser Arbeit als Kommunikationstechnologie für Datenpipelines verwendet werden.

Zusammenfassend lässt sich festhalten, dass der momentane Stand der Forschung gewisse Lücken aufweist. Modellierungsansätze für verteilte Anwendungen konnten gefunden werden, ebenso wie Deployment- und Kommunikationstechniken. Bislang fehlt es jedoch an einheitlichen Lösungen und Standards in all diesen Bereichen für VF-Anwendungen. Darüber hinaus konnten keine Quellen gefunden werden, die einen Ansatz von der Modellierung bis zur Ausführung solcher Applikationen behandeln, was der Fokus dieser Arbeit ist. Stattdessen fokussiert sich die bisherige Literatur entweder auf die Modellierung, oder auf bestimmte Deployment- und Ausführungstechniken.

4 Anforderungen

Für das Deployment und die Ausführung verteilter Datenpipelines in Software-Defined-Car-Anwendungen sollten einige Anforderungen erfüllt werden, um Effektivität und Nutzbarkeit zu gewährleisten. In diesem Kapitel wird zunächst auf Anforderungen für Systeme aus verwandten Bereichen eingegangen. Anschließend werden in den folgenden Abschnitten die Anforderungen genannt, welche den Rahmen dieser Arbeit bilden.

Der erste relevante Bereich, um Anforderungen abzuleiten, bilden die in Abschnitt 2.2 vorgestellten Datenpipelines. In ihrer Arbeit stellen Renesse et al. [RBV03] *Astrolabe* vor, eine Anwendung für verteilte Systemüberwachung, Management und Data Mining. Die Autoren listen *Skalierbarkeit*, *Flexibilität*, *Robustheit* und *Sicherheit* als notwendige Anforderungen für solch eine Applikation.

Im Bereich der vernetzten Fahrzeuge stellen Wilhelm et al. [WSM+15] eine Plattform für die Abbildung von Transportsystemen namens *Cloudthink* vor. Durch das Sammeln von Telemetriedaten zielt die Plattform darauf ab den Energieverbrauch, die Kosten, sowie die Umweltbelastung zu senken und die Sicherheit der Fahrer zu erhöhen. Als Anforderungen definieren die Autoren *Skalierbarkeit*, *Erweiterbarkeit*, sowie *Portabilität*.

Abhängig vom konkreten Anwendungsfall einer Software-Defined-Car-Anwendung können die Anforderungen jedoch stark variieren [DCR+18]. Sicherheitsrelevante Aspekte, die häufig eine Thematik in der V2P-Kommunikation darstellen, können sich von V2I-Anforderungen unterscheiden. Für Anwendungen im Bereich VRU-Sicherheit muss beispielsweise eine große Priorität auf die schnelle Datenübertragung gesetzt werden. Nur mithilfe dieser Anforderung ist es möglich zeitnah zu reagieren, indem beispielsweise ein Ausweichmanöver eingeleitet wird. Im Gegensatz dazu können analytische Anwendungsfälle im V2I-Bereich möglicherweise auf diese Anforderung verzichten. Um Daten über den Kraftstoffverbrauch und Emissionswerte zu sammeln, ist Skalierbarkeit eine häufig vorzufindende Anforderung. Obwohl die Sicherheit in Applikationen für Fahrzeuge typischerweise eine große Rolle spielt, wird diese aus zeitlichen Gründen und aufgrund des begrenzten Umfangs dieser Arbeit nicht als Anforderung definiert.

Aus diesen Gründen bilden folgende sechs Anforderungen die Grundlage für diese Arbeit und werden in den nachfolgenden Abschnitten erklärt:

- Skalierbarkeit
- Heterogenität
- Portabilität
- Erweiterbarkeit
- Robustheit/Verfügbarkeit
- Automatisierbarkeit

4.1 Skalierbarkeit

Aufgrund der hohen Mobilität von Fahrzeugen und der stark wechselnden Auslastung auf den Straßen bildet Skalierbarkeit eine wichtige Anforderung. Durch das Skalieren einzelner Komponenten ist es möglich Ressourcen zu sparen und kosteneffizienter die gleiche Leistung zu erbringen [Bon00].

Tägliche Ereignisse wie die Hauptverkehrszeit oder seltener eintreffende Anlässe, wie die erhöhte Reisekapazität während der Schulferien, können durch Skalierbarkeit adressiert werden. Da in diesen Zeiten ein erhöhtes Verkehrsaufkommen herrscht, ist es notwendig mehr Ressourcen zur Verfügung zu stellen. Durch das erhöhte Verkehrsaufkommen werden mehr Daten von den vernetzten Fahrzeugen an RSUs gesendet. Folglich benötigen diese mehr Rechenleistung und Arbeitsspeicher, um die ankommenden Nachrichten zu verarbeiten. Zusätzlich zum Erhöhen der Ressourcen kann ebenfalls eine Verteilung der Arbeitslast stattfinden. Dieses Konzept nennt sich „Load Balancing“ und sorgt für eine bessere Beanspruchung einzelner Anwendungsbestandteile.

Neben den soeben erwähnten Stoßzeiten gibt es aber auch Zeiten mit sehr niedrigem Verkehrsaufkommen. Nachts sind die Straßen beispielsweise leerer als während der Hauptverkehrszeit. In diesem Zeitraum ist also auch weniger Rechenleistung notwendig. Durch Elastizität bzw. dynamische Skalierbarkeit kann in allen Fällen gewährleistet werden, dass genug Ressourcen zur Verfügung stehen. Gleichzeitig werden aber nur so viele Ressourcen wie notwendig genutzt.

4.2 Heterogenität

Alshuqayran et al. [AAE16] benennen technologische Heterogenität als eines der wichtigen Qualitätsattribute von Microservices. Eine Datenpipeline kann gleichermaßen als heterogenes System angesehen werden. Dabei entspricht jeder Baustein der Datenpipeline einer in sich geschlossenen Komponente, ähnlich wie ein Microservice. Durch lose Kopplung zwischen den einzelnen Schritten der Datenpipeline ist es möglich auch technische Heterogenität in den Datenverarbeitungsschritten der Pipeline zu erlangen.

Dadurch ist der Entwickler in der Lage, die am besten passende Programmiersprache und Technologien auszuwählen, ohne andere Teile der Pipeline direkt zu beeinflussen. Im Zuge dessen wird die Anwendung zur gleichen Zeit zukunftssicherer. Falls die Unterstützung einer Programmiersprache oder einzelner Bibliotheken nicht mehr fortgeführt wird, müssen nur die Bausteine mit der entsprechenden Technologie modifiziert werden. Ohne Heterogenität wäre es stattdessen notwendig die gesamte Anwendung zu verändern.

4.3 Portabilität

Portabilität ist die Anforderung an eine Anwendung ohne großen Aufwand, optimalerweise ohne jegliche Veränderung des Quellcodes, auf unterschiedlichen Plattformen betrieben werden zu können. Diese Anforderung hängt mit der technischen Heterogenität zusammen. Wenn diese gewährleistet werden soll, ist es sinnvoll die einzelnen Elemente einer Pipeline auch portabel zu gestalten.

Durch die Vielfalt an Herstellern und Plattformen, auf denen Teile der Datenpipeline deployed werden müssen, können mittels portabler Software somit deutlich mehr Daten gesammelt werden. Perspektivisch wäre es sinnvoll so viele unterschiedliche vernetzte Fahrzeuge und RSUs wie möglich in die Anwendung miteinzubeziehen. Durch Portabilität einzelner Komponenten und schließlich auch der gesamten Datenpipeline wäre dies gewährleistet.

4.4 Erweiterbarkeit

Anforderungen für ITS steigen jährlich, daher ist Erweiterbarkeit im Umfeld von Software-Defined-Car-Anwendungen ein weiterer wichtiger Aspekt [LWM17]. Da künftiges Wachstum vorgesehen ist, macht es Sinn auch die zugrunde liegende Anwendung so zu gestalten, dass diese mit der Zeit erweitert werden kann, um neue Funktionalitäten möglichst einfach in die bestehende Plattform integrieren zu können. Durch die passende Architektur kann diese Anforderung bereits frühzeitig adressiert werden, wodurch späteres Hinzufügen von neuer Funktionalität erleichtert wird.

Weiterhin kann Erweiterbarkeit durch die Verwendung von Standards und De-facto-Standards erreicht werden. Hier macht es Sinn, weit verbreitete Technologien, Frameworks und Werkzeuge zu nutzen. Dadurch wird es zukünftigen Entwicklern leichter ermöglicht die Software zu modifizieren.

Neben der Wahl von Technologien die zur Erweiterbarkeit beitragen, ist auch die Qualität der entwickelten Software ein wichtiger Bestandteil dieser Anforderung. Durch das Hinzufügen von Kommentaren innerhalb des Quellcodes und die Dokumentation von Software und Schnittstellen werden zukünftige Modifikationen ebenfalls deutlich vereinfacht.

4.5 Robustheit/Verfügbarkeit

Neben den anderen Anforderungen sollte die Anwendung zusätzlich verfügbar und robust sein. Robustheit beschreibt den Grad, in dem ein System oder eine Komponente bei ungültigen Eingaben oder störenden Umgebungsbedingungen korrekt funktionieren kann [IEE90]. Verfügbarkeit ist mit der Robustheit verwandt und beschreibt den Grad, zu welchem ein System oder eine Komponente betriebsbereit und erreichbar ist, wenn diese Ressource gebraucht wird [IEE90]. Das notwendige Vorgehen um ein robustes und verfügbares System zu gewährleisten lässt sich in die zwei Punkte *Monitoring* und *Recovery* unterteilen.

Beim Monitoring geht es um die Überwachung des Zustandes einer Anwendung. Durch unvorhergesehene Ausnahmen wie beispielsweise fehlerhafte Daten kann es zu Sonderfallbehandlungen im Quellcode der einzelnen Datenverarbeitungsschritte kommen. Im schlimmsten Fall kann ein unbehandelter Ausnahmefall innerhalb einer Anwendung dazu führen, dass diese frühzeitig terminiert wird und damit nicht mehr funktionsfähig ist.

Bei verteilten Systemen ist es sinnvoll Monitoring nicht nur innerhalb einzelner Programmfragmente zu betreiben, sondern die Überwachung über die gesamte Software zu erstrecken. Damit ist es möglich auch externe Einflüsse, wie Hardwarefehler, im Notfall abzufangen. Ein Beispiel dafür wäre das Abstürzen eines Servers. Dieser Zustand muss zunächst vom System erkannt werden, um anschließend entsprechend reagieren zu können.

Im Falle einer unvorhergesehenen Terminierung eines Services ist es möglich, einen kompletten Ausfall der Anwendung durch Recovery zu verhindern. Optimalerweise kann ein Absturz des Programms frühzeitig verhindert werden, indem Sonderfälle abgefangen werden. Wenn dies nicht möglich ist, greifen Recovery-Mechanismen.

4.6 Automatisierbarkeit

Beim Modellieren einer Datenpipeline für eine Software-Defined-Car-Anwendung müssen verschiedene Bestandteile wie vernetzte Fahrzeuge, RSUs und weitere Elemente eines ITS abgebildet werden. Selbst bei einem kleinen Modell kann solch eine Datenpipeline komplex werden, besonders wenn man weitere Bausteine wie datenverarbeitende Zwischenschritte beachtet.

Durch die Anforderung der Automatisierbarkeit ist es möglich die Modellierung gesondert von Deployment und Ausführung zu betrachten. Sind sowohl Deployment und Ausführung automatisiert, können Experten eine Pipeline modellieren ohne zwingend Softwarekenntnisse zu haben. Des Weiteren wäre bei komplexen Modellen ein enormer manueller Aufwand notwendig. Bei Pipelines mit mehreren datenverarbeitenden Zwischenschritten und hunderten von RSUs wäre ein manuelles Deployment nicht praktikabel.

Neben diesen Argumenten ist die Automatisierbarkeit mit zuvor gelisteten Anforderungen verknüpft. Beispielsweise wäre es prinzipiell möglich, Skalierbarkeit manuell umzusetzen. Durch die Automatisierung wird eine Skalierung größerer Systeme jedoch erst praktisch durchführbar.

Ähnlich verhält es sich mit der Anforderung an Robustheit/Verfügbarkeit. Das Monitoring muss hier ebenfalls automatisiert ablaufen, um Fehlerzustände so früh wie möglich zu erkennen. Mit der Automatisierung der Recovery wird gleichzeitig gewährleistet, dass das System schnellstmöglich wieder in einen fehlerfreien Zustand überführt wird.

5 Deploymentkonzept für verteilte Datenpipelines

Für das Deployment und die Ausführung von Datenpiplines in Software-Defined-Car-Anwendungen wurde ein Konzept entwickelt, welches in diesem Kapitel vorgestellt wird. Das Konzept lässt sich dabei in drei Bestandteile unterteilen. Zunächst wird in Abschnitt 5.1 die Modellierung behandelt. Diese bildet die Grundlage für ein Deployment und behandelt grundlegende Konzepte für die Erstellung eines Modells für Datenpipelines. Anschließend werden der Deploymentprozess und verschiedene Ansätze für diesen in Abschnitt 5.2 betrachtet. Dafür wird ein allgemeines Konzept vorgestellt, welches beschreibt wie eine modellierte Datenpipeline zu einer ausführbaren Anwendung transformiert wird. Den Abschluss bildet ein Konzept für die Ausführung, welches in Abschnitt 5.3 behandelt wird und einen möglichen Architekturentwurf, sowie weitere laufzeitspezifische Merkmale wie Kommunikation und Monitoring beinhaltet.

5.1 Modellierung

Um eine Datenpipeline zu deployen, können verschiedene Vorgehen gewählt werden. Während es zwar möglich ist, die einzelnen Bestandteile der Datenpipeline händisch zu deployen, ist dieses Vorgehen nicht sonderlich gut für den zugrundeliegenden Anwendungsfall geeignet. Wie in Abschnitt 2.1 erklärt wurde, kann eine Datenpipeline im Bereich der VFs sehr komplex werden. Ein Grund hierfür sind die verschiedenen Entitäten in einem ITS. Hierzu zählen VRUs, VFs, RSUs, sowie weitere Sensoren und Services. Jede dieser Entitäten kann sich aus mehreren Instanzen in der reellen Welt zusammensetzen. Typischerweise ist es notwendig Daten von einer Vielzahl an VFs zu sammeln, statt nur einzelne Instanzen zu betrachten. Gleichzeitig ist es sinnvoll durch mehrere RSUs eine größere Fläche abzudecken, statt nur eine dieser Komponenten zu nutzen. Schlussfolgernd kann man festhalten, dass ein händisches Deployment deshalb typischerweise mit enormen Mehrkosten einhergeht. Doch nicht nur das initiale Deployment, auch Änderungen und das Undeployment solch einer Datenpipeline würde sehr zeitintensiv und fehleranfällig sein.

Als Alternative zum händischen Deployment ist es möglich die Pipeline zunächst zu modellieren und das entstehende Modell für ein automatisiertes Deployment zu nutzen. Das Modell bildet dabei eine Abstraktionsebene der Pipeline und ermöglicht es dem Nutzer die notwendigen Informationen für ein Deployment im Voraus anzugeben. Dieser Ansatz ist wiederverwendbar und kann auch von Personen ohne Informatikkenntnisse durchgeführt werden. Des Weiteren ist der zeitliche Aufwand deutlich geringer als bei einem händischen Deployment, da sowohl das initiale Modellieren, Änderungen, als auch das Undeployment eines Modells effizienter gestaltet werden können.

Als Grundlage für das Konzept und die Implementierung der Modellierung dient die Arbeit von Kenzler [Ken21]. In seiner Forschung stellte er bereits ein Konzept sowie einen Prototyp für diesen Anwendungsfall vor und orientierte sich dabei an dem Pipes-and-Filters-Architekturmuster.

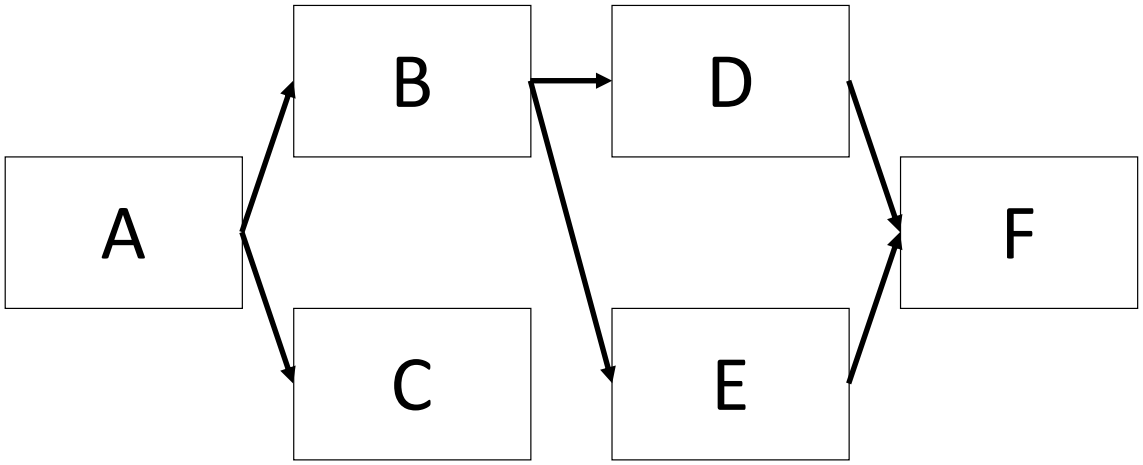


Abbildung 5.1: Abstraktion einer Datenpipeline bestehend aus einer Datenquelle (A), drei Filtern (B,D,E) zwei Datensenken (C,F) und verschiedenen Abhängigkeiten.

Dieses eignet sich Avgeriou und Zdun [AZ05] zufolge sehr gut für die stromorientierte Datenflussverarbeitung. Um möglichst vielen Nutzern den Umgang mit der Modellierungsumgebung zu ermöglichen, sind eine leichte Verständlichkeit und Bedienung wichtig. Aus diesen Gründen eignet sich eine flussdiagrammorientierte Modellierung mittels grafischer Benutzeroberfläche [Ken21].

5.1.1 Bestandteile des Modells

Dieser Abschnitt befasst sich mit den Bestandteilen, welche für die Erstellung eines Modells notwendig sind. Modelle sind an das Pipes-and-Filters-Architekturmuster angelehnt und stellen die Grundlage für Datenpipelines dar. Eine Gesamtübersicht über alle notwendigen Bestandteile für solch ein Modell sind in Abbildung 5.1 zu sehen. Das Gegenstück zu einer *Pipe* im Pipes-and-Filters-Architekturmuster bilden die Verbindungen zwischen den Komponenten. Datenquellen, Filter, sowie Datensenken entsprechen hingegen einem *Filter*. Die einzelnen Komponenten und Verbindungen werden in den folgenden Unterabschnitten nochmal näher erklärt und dabei im Kontext eines ITS betrachtet.

Datenquellen

Datenquellen bilden den Startpunkt einer Datenpipeline. Sie stellen die Komponenten dar, welche Daten erfassen und zur Verarbeitung an nachfolgende Komponenten versenden. Aus diesen Gründen besitzt eine Datenquelle keine eingehenden Verbindungen. Sie kann dafür jedoch beliebig viele ausgehende Verbindungen besitzen. Dabei sollte sie mindestens eine Verbindung zu einer anderen Komponente besitzen, da sie sonst keine Daten an die Pipeline übermitteln kann.

Im Umfeld eines ITS kann eine Datenquelle als Abstraktion für eine Vielzahl von Elementen angesehen werden. Typischerweise entspricht ein einzelnes VF einer Datenquelle. Dieses produziert Daten, welche durch nachfolgende Komponenten verarbeitet und schließlich abgelegt werden sollen. Andere Beispiele für Datenquellen im Kontext des ITS sind Sensoren wie Induktionsschleifen und Ampelschaltungen, oder auch externe Services wie Wetterdienste.

Filter

Filter sind datenverarbeitende Komponenten innerhalb einer Datenpipeline. Genauso wie Datenquellen, können Filter beliebig viele ausgehende Verbindungen besitzen, mindestens jedoch eine. Gleichzeitig können Filter auch beliebig viele eingehende Datenströme besitzen. Auch hier ist ein Minimum von einer Verbindung notwendig. Daten, welche den Filter über diese Verbindungen erreichen, werden bearbeitet und anschließend an die folgende Komponente in der Datenpipeline weitergeleitet. Zu den datenverarbeitenden Operationen zählen beispielsweise die Aggregation oder Filterung der Daten.

Ein Beispiel für einen Filter im Bereich des ITS bildet eine Aggregation. Typischerweise besitzt solch ein Filter viele eingehende Verbindungen, bei welchen die verbundenen Fahrzeuge einzelnen Datenquellen entsprechen. Dadurch ist er in der Lage Nachrichten von verschiedenen VFs zu erhalten und diese zu aggregieren. Je nach Anwendungsfall werden die eingehenden Informationen unterschiedlich verarbeitet und anschließend von dem Filter weitergesendet.

Datensenken

Datensenken bilden das Ende einer Datenpipeline. Während sie keine ausgehenden Verbindungen besitzen, sind beliebig viele eingehende Verbindungen möglich. Eine Datenbank oder ein Data Lake stellen ein typisches Anwendungsszenario für eine Datensenke im ITS-Bereich dar. Hier werden die Informationen gesammelt, welche zuvor durch die datenverarbeitenden Schritte in das richtige Format gebracht wurden. Diese Datensenken ermöglichen dann wiederum eine genauere Analyse der gesammelten Fahrzeugdaten.

Verbindungen

Verbindungen zwischen Komponenten entsprechen den Pipes des Pipes-and-Filters-Architekturmusters. Diese gerichteten Kanten abstrahieren den Datenstrom von einer Komponente zur nächsten und ermöglichen es somit den Datenfluss zu modellieren. Ein Beispiel hierfür ist die Verbindung zwischen einer Datenquelle auf einem VF und einem Filter auf einer RSU. In späteren Schritten wie dem Deployment oder der Ausführung ist die Angabe der Verbindung notwendig, um eine Kommunikation zwischen den Komponenten zu etablieren. Einzelne Bestandteile der Pipeline können jedoch auch mehrere eingehende und ausgehende Verbindungen besitzen. Ein Beispiel hierfür wäre ein externer Wetterdienst. Beim Beziehen von Wetterinformationen können verschiedene Filter an den gleichen Service angebunden sein.

5.1.2 Erweiterungen

Neben den Hauptbestandteilen des Modells wird im Folgenden auf verschiedene Erweiterungen eingegangen. Während die zuvor genannten Komponenten zwar ausreichen, um eine Datenpipeline zu erstellen, sind zusätzliche Informationen sinnvoll. Im Folgenden werden diese vorgestellt. Dabei wird ein Bezug zur Notwendigkeit hergestellt. Während einige Erweiterungen für eine bessere Bedienbarkeit sorgen, sind andere notwendig, um ein automatisches Deployment zu ermöglichen.

Metadaten

Zur leichteren Handhabung eines Modells ist es sinnvoll Metadaten zum Modell festhalten zu können. Auf Modellebene ist das Vergeben eines Namens denkbar. Dadurch ist es einfacher Modelle zuzuordnen und zu organisieren. Innerhalb der einzelnen Komponenten eines Modells sind Metadaten ebenfalls eine Möglichkeit die Funktionalität der Anwendung zu erweitern. Zusätzliche Informationen für das Deployment einer Pipeline können für die Automatisierbarkeit essenziell sein, wie beispielsweise die IP-Adresse des Rechners, auf welchen deployed werden soll.

Templates

Analog zur Zeitersparnis durch das Erstellen eines Modells gegenüber dem händischen Deployment, können Templates ebenfalls den benötigten Aufwand verringern. Als Templates werden Schablonen für die Erstellung einzelner Komponenten bezeichnet. Die Idee ist es, die benötigte Zeit für die Modellierung durch Wiederverwendbarkeit der Datenquellen, Filter und Datensenken zu minimieren. Ein Template definiert dabei die Anzahl an Ein- und Ausgängen der Komponente und benötigt weitere Informationen wie Namen und Typ, um später richtig zugeordnet werden zu können. Die zum Template gehörenden Informationen sind schließlich für alle Instanzen einer Komponente, die auf diesem Template basieren, gültig. Weiterhin soll es nach wie vor die Möglichkeit geben, Metadaten an einzelne Instanzen eines Templates anheften zu können, um ein automatisiertes Deployment zu garantieren.

Multi-Deployment

Ein weiteres Konzept zur Vereinfachung der Modellierung stellen Multi-Deployments dar. Bei der Modellierung eines ITS ist die Aufteilung der Komponenten typischerweise quantitativ sehr unterschiedlich. Während es sehr viele VFs und Sensoren geben kann, gibt es normalerweise weniger RSUs. Um beispielsweise jedes Fahrzeug einer Flotte in ein Modell einzubetten, wäre ein enormer manueller Aufwand notwendig. Multi-Deployments sollen dieses Problem lösen, indem eine Instanz nur einmalig modelliert werden muss, aber auf beliebig viele unterschiedliche Rechner deployed wird.

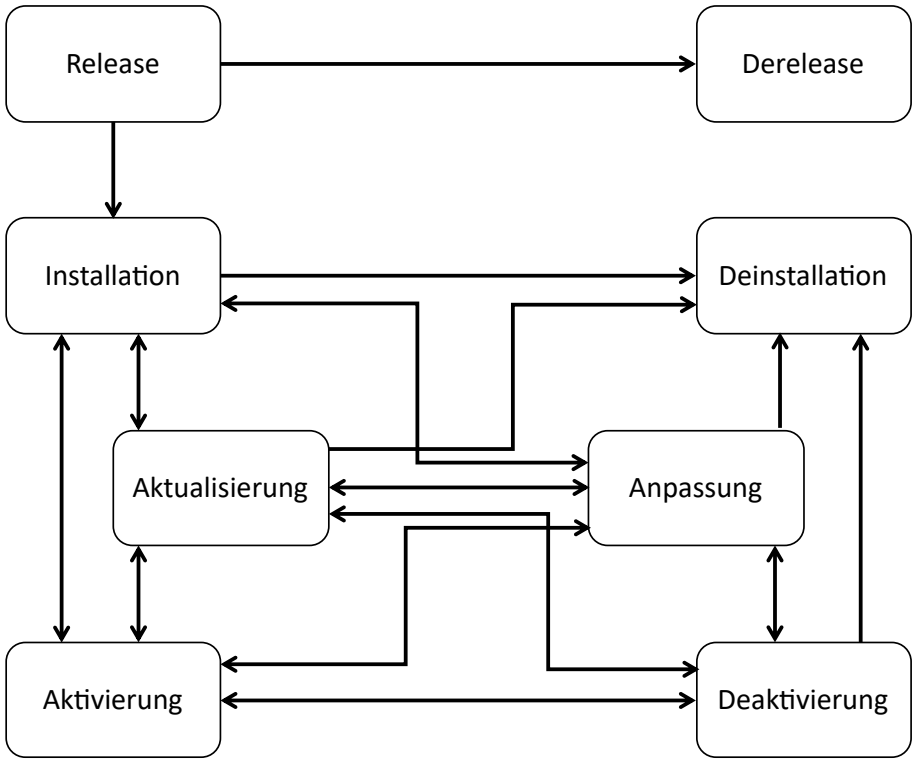


Abbildung 5.2: Aktivitäten im Software-Deployment-Prozess und deren Abhängigkeiten, nach [CFH+98].

5.2 Deployment

Da eine Datenpipeline ein verteiltes System darstellt, ist ein Konzept für das Deployment der einzelnen Verarbeitungsschritte notwendig. Das Deployment bezieht sich dabei auf die Aktivitäten, die das Softwaresystem für die Nutzung verfügbar machen [CFH+98]. In Abschnitt 5.2.1 wird das Konzept eines Deployment-Prozesses in seinen Bestandteilen vorgestellt. Anschließend werden in Abschnitt 5.2.2 verschiedene Ansätze für Deployments betrachtet.

5.2.1 Software-Deployment-Prozess

Der Prozess eines Software-Deployments umfasst eine Reihe von Aktivitäten, welche den Lebenszyklus eines Softwaresystems beeinflussen [MM16]. Ein generischer Deployment-Prozess lässt sich in der Regel durch eine gewisse Reihe von eindeutigen Aktivitäten identifizieren. Die Vorgehensweisen und Verfahren innerhalb dieser Aktivitäten lassen sich jedoch nicht präzise definieren. Grund dafür ist die starke Abhängigkeit von der Art der Software die ausgeliefert werden soll, sowie von den Eigenschaften und Anforderungen der Hersteller und Verbraucher [CFH+98]. Carzaniga et al. [CFH+98] unterteilen den Software Deployment-Prozess in acht Aktivitäten, welche in Abbildung 5.2 dargestellt sind und in den nachfolgenden Unterabschnitten erklärt werden:

Release/Derelease

Der Release beschreibt die Tätigkeit der Software-Paketierung für die Auslieferung an den Endnutzer. Dieser Schritt umfasst alle Vorgänge, die erforderlich sind, um ein System für die Installation und die Auslieferung an den Verbraucher vorzubereiten. Dazu gehören Prozesse wie die Einbeziehung der Abhängigkeiten von externen Komponenten, zum Beispiel Bibliotheken und Anwendungen.

Der Derelease stellt das Gegenstück zu einem Release dar. Wenn ein System obsolet wird kann dieses durch den Derelease als letzte Aktivität des Deployment-Prozesses endgültig entfernt werden.

Installation/Deinstallation

Aufbauend auf einem Release, kann die Software im Installations-Schritt in einen Konsumenten eingefügt werden. Hierfür muss die zu installierende Software vom Produzenten an den Konsumenten transferiert werden. Anschließend werden notwendige Konfigurationsoperationen ausgeführt, um das System für die Aktivierung vorzubereiten.

Wenn die Software auf der Konsumentenseite nicht länger benötigt wird, kann die Deinstallation durchgeführt werden. Dafür wird die Deaktivierung der Software vorausgesetzt. Bei der Deinstallation werden die installierten Konfigurationen, welche für eine Aktivierung notwendig sind entfernt.

Aktivierung/Deaktivierung

Als Aktivierung wird das Starten der ausführbaren Komponenten eines Systems bezeichnet. Grundvoraussetzung hierfür ist die Installation und Konfiguration des Konsumenten, auf welchem die Software aktiviert werden soll. Weiterhin ist es möglich, dass die Aktivierung einer Software von anderen aktivierten Systemen abhängig ist. In solch einem Fall muss sichergestellt werden, dass ein rekursives Deployment dieser ausgeführt wird und alle notwendigen Anforderungen für ein System vor dessen Aktivierung vorhanden sind.

Die Deaktivierung ist das Inverse einer Aktivierung. Diese Aktivität bezeichnet das Stoppen aller ausführbaren Anwendungen eines installierten Systems. Für viele Aktivitäten im Deployment-Prozess muss zunächst die Deaktivierung durchgeführt werden. Dies ist beispielsweise beim Durchführen einer Aktualisierung notwendig.

Aktualisierung

Die Aktualisierung eines Systems stellt eine besondere Art der Installation dar, die sich meistens auf Ereignisse innerhalb des Systems bezieht. Dazu zählen zum Beispiel Änderungen der Konfiguration, auf welche reagiert werden muss. Typischerweise wird die Aktualisierung so gehandhabt, dass ein System zunächst deaktiviert wird, eine neue Version installiert und anschließend aktiviert wird. Bei gewissen Applikationen ist dies nicht notwendig, sodass eine Aktualisierung ohne vorherige Deaktivierung stattfinden kann. Wie bei der Installation beinhaltet dieser Schritt den Transfer und die Konfiguration des zu aktualisierenden Systems.

	Physisch	Virtual Machine	Container
Ladezeit	-	0	+
Komplexität	-	0	+
Skalierbarkeit	-	0	+
Performance	+	-	0
Betriebssystemunabhängigkeit	-	+	-

Tabelle 5.1: Übersicht der Deploymentansätze und Bewertung verschiedener Charakteristiken. Dabei steht ein „+“ für „gut“, „0“ für „durchschnittlich“ und „-“ für „schlecht“.

Anpassung

Ebenfalls wie die Aktualisierung beinhaltet die Anpassung das Verändern eines zuvor installierten Systems. Der Unterschied zwischen Anpassung und Aktualisierung hängt vom Ursprung des Ereignisses ab. Anpassungen werden typischerweise durch Ereignisse aus anderen Systemen ausgelöst, zum Beispiel dem Release eines Softwareupdates. Eine Anpassungsmaßnahme wird möglicherweise eingeleitet, um Korrekturmaßnahmen zur Aufrechterhaltung der Funktionsfähigkeit des installierten Softwaresystems zu gewährleisten.

5.2.2 Deploymentansätze

Ein wichtiger Teil des Deploymentkonzepts beinhaltet die Auswahl eines passenden Deploymentansatzes. Dabei kann man zwischen *physischen* und *virtuellen* Deploymentansätzen unterscheiden [KSV17]. Beim virtuellen Deploymentansatz ist es weiterhin möglich, die ausgelieferte Software in Form von VMs oder Containern zu deployen, wie zuvor in Abschnitt 2.3 vorgestellt. Bei einem physischen Deployment wird die Software ohne Virtualisierung auf einem Rechner aufgesetzt.

In den folgenden Unterabschnitten werden relevante Charakteristiken für die verschiedenen Deploymentansätze erläutert. So kann das Konzept dabei unterstützen den passendsten Deploymentansatz für die Erstellung eines Prototyps zu wählen. Eine Übersicht aller Charakteristiken und Deploymentansätze ist in Tabelle 5.1 dargestellt. Abschließend wird ein Gesamtfazit der Deploymentansätze im letzten Unterabschnitt vorgestellt.

Ladezeit

Einen relevanten Aspekt für den Vergleich der verschiedenen Ansätze stellt die Ladezeit dar. Diese Charakteristik beschreibt die Zeit die benötigt wird, bis die Software verfügbar ist und von einem Endnutzer verwendet werden kann. Abhängig von der Nutzungszeit einer Software kann diese Charakteristik wichtig oder unwichtig sein. Wird eine Anwendung beispielsweise nur selten deployt und hat eine sehr lange Laufzeit, so ist es möglich diese Charakteristik zu vernachlässigen [KSV17].

In einem Vergleich der drei Deploymentansätze konnten Kominos et al. [KSV17] zeigen, dass der physische Deploymentansatz die mit Abstand längsten Ladezeiten benötigt. Beide Virtualisierungsmöglichkeiten sind im Vergleich deutlich schneller. Der Container-basierte Ansatz schneidet jedoch besser ab, als das Deployment mittels VM. Der Hintergrund hierfür ist, dass eine VM ein gesamtes Betriebssystem betreibt und typischerweise mehr Speicherplatz benötigt als ein Container.

Komplexität

Die Komplexität kann für ein Deployment ebenfalls eine wichtige Rolle spielen. Je nachdem wie häufig ein neues Deployment notwendig ist, kann diese Charakteristik unterschiedlich stark gewichtet werden. Als Komplexität wird dabei der Aufwand bezeichnet, um einen Konsumenten für ein Deployment vorzubereiten.

Um ein System mittels physischem Ansatz auf einem Rechner zu deployen, ist es notwendig zunächst alle Anforderungen wie Bibliotheken und weitere Abhängigkeiten bereitzustellen. Beim physischen Deployment müssen diese zunächst installiert und aufgesetzt werden. Eine VM muss ebenfalls vorbereitet werden, indem ein entsprechendes Gast-Betriebssystem aufgespielt wird und notwendige Abhängigkeiten verfügbar gemacht werden. Um diese Schritte zu automatisieren, gibt es Lösungen wie Vagrant[1] oder Puppet[2]. Beim Container-basierten Ansatz hingegen werden sogenannte „Images“ genutzt. Diese unterscheiden sich von den vorherigen beiden Ansätzen, da ein Host-System vorausgesetzt wird. Dadurch benötigt ein Image lediglich sämtliche Dateien, von denen die zu deployende Anwendung abhängig ist. Speziell bei sicherheitsrelevanten Applikationen wie dem Deployment von Anwendungen im VF-Bereich, eignet sich eine geringere Komplexität besser, um Fehler und Bugs schneller beheben zu können. Aus diesem Grund eignen sich die beiden Virtualisierungsmöglichkeiten besser als das physische Deployment, welches die größte Komplexität mit sich bringt. Dabei ist das Deployment einer VM tendenziell komplexer, da ein Betriebssystem mit den notwendigen Abhängigkeiten bereitgestellt werden muss.

Skalierbarkeit

Bei großen verteilten Systemen mit mehreren Komponenten kann die Skalierbarkeit eine wichtige Rolle bei der Wahl des Deploymentansatzes spielen. Für diese Anwendungen sind die beiden virtuellen Ansätze besser geeignet als ein physisches Deployment. Um die verfügbaren Ressourcen zu erhöhen, wäre es beim physischen Ansatz notwendig Komponenten des Rechners zu verändern oder einen neuen Rechner bzw. Server einzurichten. Im Gegensatz dazu ist das Erweitern verfügbarer Ressourcen in virtuellen Umgebungen deutlich weniger aufwändig. Bestehenden Containern bzw. VMs können entweder mehr Ressourcen zugewiesen werden, oder sie können auf anderen Rechnern repliziert werden. Container sind hierfür jedoch noch besser geeignet als VMs, da der geringe Ressourcenverbrauch eine bessere Leistung in kleinem und größerem Maßstab ermöglicht [Ede16].

[1] Vagrant: https://www.vagrantup.com/
[2] Puppet: https://puppet.com/

Performance

Li et al. [LKLA17] zufolge ist die durchschnittliche Leistung eines Containers im Allgemeinen besser als die der VM und bei vielen Funktionen sogar vergleichbar mit der physischen Maschine. In ihrem Vergleich zwischen Container und VM konnten Potdar et al. [PGKM20] ebenfalls feststellen, dass Container eine besser Leistung erzielen. Untersucht wurden CPU-Leistung, Speicherdurchsatz, Festplatten-E/A Operationen, Lasttests und Messungen der Verarbeitungsgeschwindigkeit. Den Autoren zufolge konnte der Container in jeder Kategorie bessere Ergebnisse als die VM erzielen.

Betriebssystemunabhängigkeit

Abhängig vom Anwendungsfall kann es notwendig sein das gleiche System auf unterschiedlichen Betriebssystemen zu deployen. Ein Beispiel hierfür könnte das Testen eines User Interfaces auf verschiedenen Mobilgeräten sein. Beim physischen Deployment besteht eine direkte Abhängigkeit zwischen Software und darunterliegendem Betriebssystem, sowie der CPU-Architektur. Das Gleiche gilt für ein Container-basiertes Deployment, da dieses ebenfalls vom Betriebssystem des Rechners abhängig ist. Mit dem VM-Ansatz ist es möglich ein beliebiges Gast-Betriebssystem zu wählen, welches sich vom Host-Betriebssystem unterscheidet, wodurch dieser Ansatz als einziger eine komplette Unabhängigkeit gewährleistet.

Gesamtfazit

Zusammenfassend lässt sich festhalten, dass sich ein physisches Deployment für sicherheitsrelevante Anwendungen, wie in der VF-Domäne, am wenigsten eignet. Neben der langsamen Ladezeit und der höheren Komplexität stellt auch die Skalierbarkeit hier ein Problem dar. Beide Virtualisierungstechniken eignen sich in diesen Charakteristiken besser für das Deployment einer VF-Anwendung, bieten dafür aber eine niedrigere Performance als das physische Deployment. Der größte Unterschied zwischen einem VM- und einem Container-basierten Deployment stellt die Betriebssystemunabhängigkeit dar. Ist diese notwendig, eignet sich der VM-basierte Ansatz besser. Falls die Anwendung auf gängigen Betriebssystemen wie Windows, Linux oder macOS deployt werden soll, ist der Container-basierte Ansatz in den Punkten Ladezeit, Komplexität, Skalierbarkeit und Performance jedoch besser geeignet.

5.3 Ausführung

Da verteilte Systeme komplex werden können ist neben der Modellierung und dem Deployment zusätzlich ein Konzept für die Ausführung notwendig. Aus diesem Grund wird in Abschnitt 5.3.1 eine mögliche Architektur für eine Anwendung aufgezeigt, welche für das Deployment und die Ausführung von Datenpipelines in Software-Defined-Car-Anwendungen genutzt werden kann. Ein weiterer wichtiger Aspekt ist die Kommunikation zwischen den einzelnen Komponenten. Da der Datenaustausch bei Systemen dieser Art sehr heterogen sein kann, werden Möglichkeiten in Abschnitt 5.3.2 aufgezeigt. Um bestmöglich auf Veränderungen und Probleme reagieren zu können, werden in Abschnitt 5.3.3 notwendige Bestandteile der Systemüberwachung präsentiert.

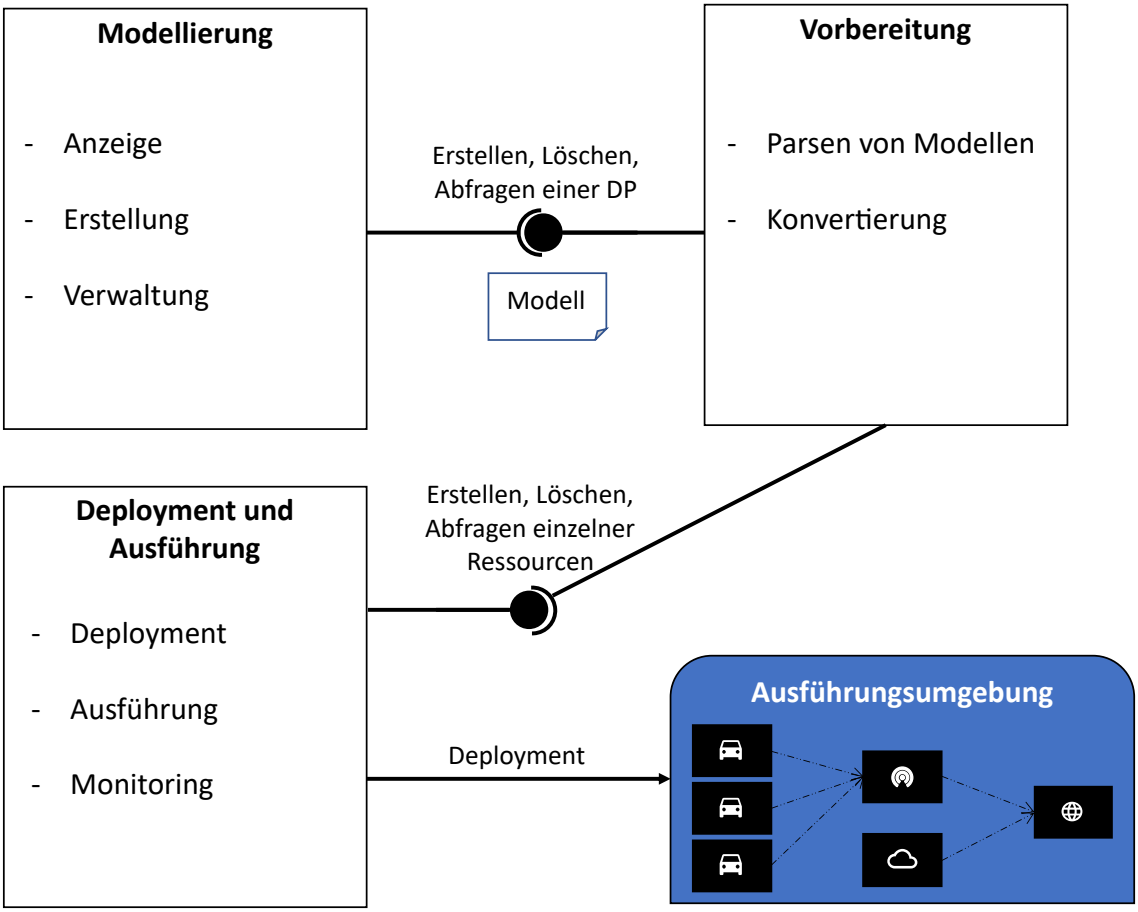


Abbildung 5.3: Architektur einer Software für das Deployment und die Ausführung von Datenpipelines bestehend aus drei Komponenten.

5.3.1 Architektur

Um eine effiziente Ausführung der Anwendung zu ermöglichen, wird in diesem Abschnitt ein möglicher Architekturentwurf behandelt. Hierfür wird zunächst die Architektur betrachtet. Anschließend werden die einzelnen Komponenten und deren Zweck in den folgenden Unterabschnitten genauer erklärt. Das Ziel des Architekturentwurfs ist es, als Basis für die spätere Implementierung eines Prototyps zu dienen.

Abbildung 5.3 zeigt eine mögliche Architektur der Anwendung. Dabei wird die Funktionalität auf drei Komponenten aufgeteilt. Mit diesem Ansatz sind die Funktionalitäten für die Modellierung, Deploymentvorbereitung und die Durchführung des Deployments sowie die Ausführung voneinander abgekapselt. Durch die lose Kopplung zwischen den Komponenten ist es einfacher möglich Änderungen an ihnen vorzunehmen. Darüber hinaus kann eine einzelne Komponente ausgetauscht werden, ohne dass dies einen Einfluss auf das Gesamtsystem hat.

Modellierung

Die Modellierungskomponente bildet im vorgestellten Architekturentwurf das Frontend der Anwendung. Als eigenständige Komponente und durch die lose Kopplung mit anderen Bestandteilen der Anwendung bieten sich so mehrere Möglichkeiten für die Umsetzung der Modellierung. Eine Option wäre die Verwendung eines bereits bestehenden Modellierungswerkzeugs, wodurch der Implementierungsaufwand des Systems gesenkt werden könnte. Alternativ kann eine individuelle Modellierungsumgebung entwickelt werden, wodurch die Funktionalität maximal an die eigenen Anforderungen angepasst werden kann.

Das Ziel dieser Komponente ist es, dem Nutzer eine Oberfläche für die Erstellung eines Modells für die Datenpipeline zu bieten, welche anschließend deployed werden soll. Die Hauptfunktionalitäten sind dabei die Anzeige, Erstellung und Verwaltung von Datenpipelines. Wie man Abbildung 5.3 entnehmen kann, ist eine Verbindung zwischen der Modellierung und der Vorbereitung dargestellt. Diese soll verdeutlichen, dass ein Datenfluss zwischen den beiden Komponenten notwendig ist. Der Hauptbestandteil dieser Kommunikation bildet das vom Nutzer erstellte Modell, welches in ein repräsentatives Datenformat umgewandelt werden muss und anschließend an die nächste Komponente übergeben werden kann, wo es anschließend weiterverarbeitet wird. Durch diese Trennung kann weiterhin gewährleistet werden, dass das Frontend möglichst wenige Berechnungen durchführen muss, da der Großteil der Logik in andere Komponenten ausgelagert wird. So kann das Frontend performant gestaltet werden und beispielsweise als Webanwendung Client-seitig ausgeführt werden.

Mit diesem Ansatz ist es dementsprechend möglich die Anzahl der Frontend-Instanzen beliebig zu erhöhen. Dadurch können eine Vielzahl von Endanwendern gleichzeitig das Frontend nutzen. Typischerweise ist die Modellierung eine zeitintensive Aufgabe, wodurch diese Komponente vermutlich am längsten genutzt wird. Das Deployment eines fertigen Modells findet im Vergleich deutlich seltener statt. Auch hier bietet die vorgeschlagene Lösung eine gute Möglichkeit die einzelnen Komponenten unterschiedlich zu handhaben und je nach Auslastung mehr oder weniger Ressourcen zur Verfügung zu stellen.

Vorbereitung

Die Vorbereitung ist der Teil der Anwendung, welcher den Großteil der Logik beinhaltet und rechenintensive Operationen ausführt. Den mitunter wichtigsten Anwendungsfall stellt die Entgegennahme des Modells und die Weiterverarbeitung der Daten dar. Somit ist es sinnvoll eine Schnittstelle für die eingehenden Daten zu definieren, welche vom Frontend genutzt werden kann.

Um ein empfangenes Modell zu deployen sind verschiedene Schritte notwendig. Diese orientieren sich am gewählten Deploymentansatz und sind daher variabel. Grundsätzlich ist es jedoch notwendig zunächst einige vorbereitende Schritte für das Deployment zu treffen. Ein entscheidender Teil ist dabei das Parsen des Modells und der Aufbau einer Topologie. Durch die Verbindungen zwischen den Datenverarbeitungsschritten innerhalb der Pipeline, lassen sich einzelne Komponenten so als Datenquellen, Filter und Datensenken identifizieren, was als eine Konvertierung der Datenhaltung angesehen werden kann. Gleichzeitig ist es notwendig die Nachbarn eines Knotens ausfindig zu machen, um den Datenfluss zwischen den einzelnen Schritten zu ermöglichen.

Wenn diese Schritte abgeschlossen sind, ist das vom Frontend empfangene Datenformat in eine interne Darstellung des Modells umgewandelt worden und kann schließlich bereitgestellt werden. Das eigentliche Deployment wird dabei aber lediglich von der Vorbereitungskomponente koordiniert. Die Provisionierung sowie Ausführung der Datenpipeline werden anschließend durch eine weitere Komponente umgesetzt, was durch die Verbindung zwischen den beiden in Abbildung 5.3 verdeutlicht wird.

Deployment und Ausführung

Die letzte Komponente der Anwendung ermöglicht das Deployment, die Ausführung und das Monitoring von Datenpipelines. Dafür wird eine Schnittstelle bereitgestellt, welche von der Vorbereitungskomponente genutzt wird, um die einzelnen Bestandteile wie Datenquellen, Filter und Datensenken, sowie ergänzende Metainformationen zu übermitteln.

Ähnlich wie beim Frontend ist diese Komponente als gesonderte Einheit vorgesehen. Dadurch lässt sich eine maximale Flexibilität für die gewählte Art der Umsetzung gewährleisten. Abhängig vom gewählten Deploymentansatz können hier unterschiedliche Lösungen für die Implementierung gewählt werden. Für den Container-basierten Ansatz könnte sich der Einsatz eines Container-Orchestrators eignen. Alternativ ist es beim VM-basierten Ansatz möglich auf Werkzeuge für das Deployment von VMs zurückzugreifen. In der Regel beinhalten die vorgestellten Lösungen sehr allgemeine Schnittstellen für die Erstellung, Verwaltung und Überwachung von VMs oder Containern. Aufgrund der vielfältigen Funktionen von bereits bestehenden Lösungen ist eine eigene Implementierung für diese Komponente eine sehr aufwändige Aufgabe.

5.3.2 Kommunikation

Die Kommunikation spielt bei verteilten Systemen eine wichtige Rolle, da es verschiedene Möglichkeiten gibt diese umzusetzen. Je nach Anforderung kann der Einsatz der gewählten Kommunikationsmuster variieren. Es ist also nicht möglich eine allgemeingültige Lösung zu wählen, sondern es ist notwendig diese Entscheidung auf Basis des Anwendungsszenarios zu treffen.

Im Folgenden werden zwei Optionen für die Handhabung der Kommunikation zwischen den verschiedenen Komponenten dargestellt. Dafür wird zunächst das Kommunikationsmuster Request-Reply im folgenden Unterabschnitt aufgegriffen. Anschließend wird der Publish-Subscribe-Ansatz im darauffolgenden Unterabschnitt im Kontext einer ITS-Anwendung betrachtet.

Request-Reply

Eines der am meisten verbreiteten Kommunikationsmuster ist Request-Reply. Dieses Verfahren ermöglicht es zwei Entitäten in einem Netzwerk direkt miteinander zu kommunizieren. Wie in Abbildung 5.4 dargestellt, schickt die erste Komponente eine Anfrage an die zweite, woraufhin die zweite Komponente eine Antwort zurücksendet. Ein Beispiel für die Nutzung dieses Musters innerhalb eines ITS könnte die Abfrage der momentanen Ampelphase sein. Dabei würde ein VF eine Anfrage an die Ampel senden und anschließend eine Antwort mit den benötigten Daten erhalten.

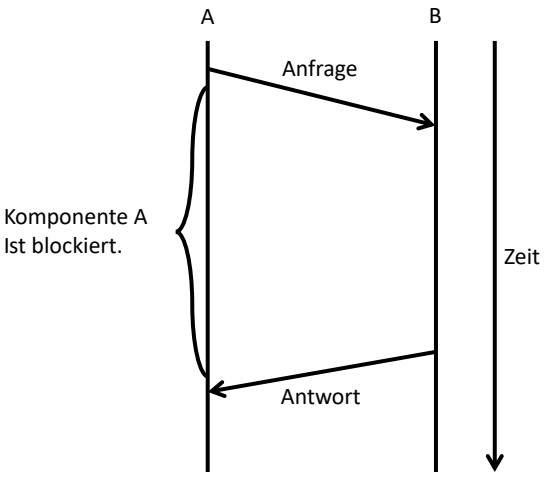


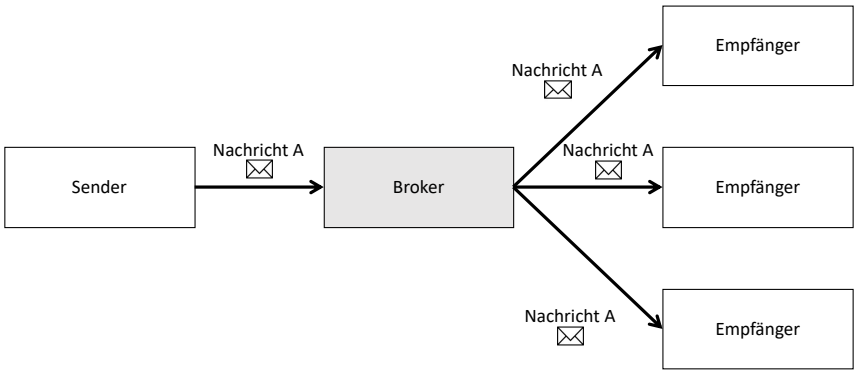
Abbildung 5.4: Veranschaulichung der asynchronen Kommunikation zwischen zwei Komponenten bei Verwendung des Request-Reply-Musters.

Während sich das Request-Reply-Muster gut für CRUD (Create, Read, Update, Delete) Operationen eignet, kann die synchrone Arbeitsweise ein Nachteil sein. Bei diesem Muster blockiert der Absender die Verarbeitung typischerweise so lange, bis er eine Antwort bekommt und führt anschließend erst die Ausführung weiter fort [KSA09]. Bei dem vorherigen Beispiel würde dies bedeuten, dass ein VF erst nötige Maßnahmen einleiten kann, wenn es eine Anfrage versendet und die entsprechende Antwort erhalten hat. Für solch einen sicherheitskritischen Anwendungsfall kann diese Limitation des Request-Reply-Musters einen großen Nachteil darstellen. Ist die Wartezeit für das Erhalten einer Antwort zu lang, kann dies lebensgefährliche Folgen haben, da das VF nicht früh genug reagieren kann. Eine hohe Geschwindigkeit des Fahrzeugs oder eine Überlastung der Ampel durch eine hohe Zahl von Anfragen könnten Ursachen für fatale Auswirkungen sein.

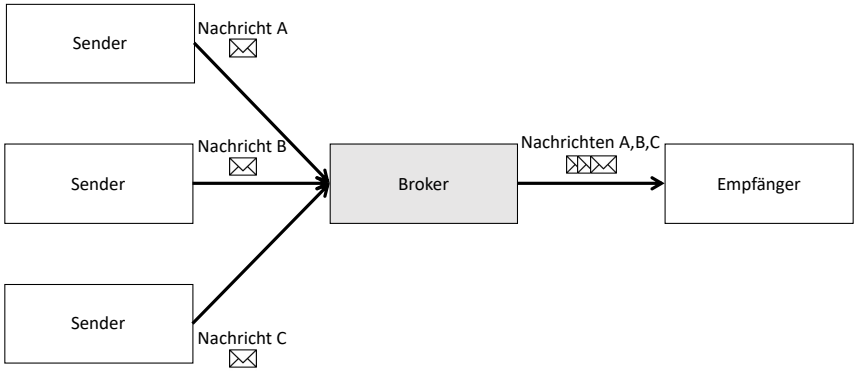
Weiterhin ist es notwendig ein Konzept für das Deployment einer Datenpipeline zu erstellen, wenn Request-Reply zwischen Komponenten innerhalb der Pipeline genutzt wird. Dabei müssen Knoten mit mehreren ausgehenden Verbindungen gesondert betrachtet werden. Für jede Verbindung mittels Request-Reply-Muster ist es nämlich notwendig die Adresse des Knotens zu kennen, welchen die Anfrage erreichen soll. Besitzt eine Datenquelle oder ein Filter viele ausgehende Kanten, entspricht jede dieser Kanten auch einer Adresse, welche im Voraus bekannt sein muss. Weiterhin muss diese Information von Außen in die Komponente hineingegeben werden, damit eine funktionierende Kommunikation während der Laufzeit gewährleistet werden kann.

Publish-Subscribe

Speziell für verteilte Echtzeitanwendungen konnte das Publish-Subscribe-Kommunikationsmuster eine hohe Popularität erzielen. Im Gegensatz zu Point-to-Point-Verbindungen, wie bei dem zuvor vorgestellten Request-Reply-Ansatz, werden der Sender und Empfänger hier entkoppelt. Diese Entkopplung geschieht typischerweise entweder durch die Verwendung von Topic-basierter oder Inhalt-basierter Nachrichtenübermittlung. Um den Umfang einzugrenzen, wird im Folgenden nur der Topic-basierte Ansatz betrachtet. Mit diesem Vorgehen werden Nachrichten von Sendern an Topics



(a) Publish-Subscribe Beispiel mit einem Sender und drei Empfängern.



(b) Publish-Subscribe Beispiel mit drei Sendern und einem Empfänger.

Abbildung 5.5: Übersicht des Publish-Subscribe-Kommunikationsmusters anhand von zwei Fallunterscheidungen. (a) Zeigt dabei ein Szenario mit einem Sender und drei Empfängern, welche alle dieselbe Nachricht erhalten. (b) Schildert eine Anwendung mit drei verschiedenen Sendern und einem Empfänger, der alle gesendeten Nachrichten empfängt.

geschickt. Empfänger können sich für Topics registrieren und erhalten somit alle Nachrichten, welche unter den registrierten Topics veröffentlicht werden. Für die erfolgreiche Nachrichtenübermittlung existieren sogenannte Broker. Nachrichten werden über diese Komponente geleitet und entsprechend der Topics an die registrierten Interessenten verschickt.

Durch diesen Ansatz bringt das Publish-Subscribe-Muster einige Vorteile mit sich [OKF10]. Direkt nach dem Vorkommen eines Ereignisses kann dieses von den Empfängern erhalten werden, wodurch sich Publish-Subscribe gut für Echtzeitanwendungen eignet. Weiterhin ist die asynchrone Arbeitsweise ein großer Vorteil, da der Sender nicht auf eine Bestätigung des Empfängers warten muss. Zusätzlich eignet sich Publish-Subscribe sehr gut für Knoten mit vielen ein- oder ausgehenden Verbindungen. Dies ist in Abbildung 5.5 dargestellt. In Abbildung 5.5a ist ein Knoten mit mehreren ausgehenden Verbindungen abgebildet. Der Sender könnte die Ampelschaltung eines ITS sein, während die Empfänger verschiedene VFs darstellen. In diesem Fall reicht das Senden einer einzigen Nachricht aus, um alle Empfänger zu erreichen, welche auf das entsprechende Topic hören. Abbildung 5.5b zeigt ein umgekehrtes Szenario, also einen Knoten mit vielen eingehenden Verbindungen. Ein Beispiel hierfür ist ein Filter, welcher auf einer RSU deployt ist, und Daten

von verschiedenen VFs sammelt. In diesem Fall profitiert die Anwendung durch ein vereinfachtes Deployment im Gegensatz zur Realisierung mittels Request-Reply-Muster. Um Nachrichten von verschiedenen Empfängern zu erhalten, genügt es mit diesem Ansatz ein Topic zu erstellen. Wenn die Sender ihre Nachrichten an das Topic senden, kann der Empfänger dieses abonnieren und so Nachrichten von beliebig vielen Sendern erhalten.

5.3.3 Monitoring

Verteilte Systeme wie die behandelten Datenpipelines werden typischerweise auf verschiedenen Rechnern ausgeführt. Mit diesem Ansatz können Komponenten von mehr verfügbaren Ressourcen profitieren. Grund dafür ist die einfachere Skalierung eines Systems auf mehrere Rechner. Die Kosten für die Erhöhung der bestehenden Ressourcen, also der Erweiterung des Systems um einen neuen Rechner, ist in der Regel günstiger als die Erweiterung bestehender Ressourcen für einen einzelnen Rechner. Um Redundanz zu erzeugen und dadurch eine bessere Verfügbarkeit einer Komponente zu gewährleisten können zusätzliche Mechanismen wie die Replikation genutzt werden. In solch einem Szenario kann die Anzahl an Rechnern und deployten Instanzen des Gesamtsystems schnell groß und unübersichtlich werden.

Beim Deployment von Datenpipelines für Software-Defined-Car-Anwendungen kann es dadurch sehr aufwändig werden einzelne Komponenten zu überwachen. Bei Systemen bestehend aus wenigen Komponenten ist dies händisch möglich, indem auf die entsprechende Maschine zugegriffen wird und der Zustand manuell abgefragt wird. Für ein System welches aus mehreren Rechnerknoten und einer Vielzahl an deployten Instanzen verfügt ist dieser Ansatz nicht mehr effizient. Aus diesem Grund ist es sinnvoll ein automatisiertes Monitoring zur Zustandsüberwachung der verschiedenen Systembestandteile zur Gesamtanwendung hinzuzufügen.

Während Monitoring je nach Bedürfnis beliebig erweitert werden kann, sind zwei Bestandteile für das Deployment von Pipelines notwendig, um eine korrekte Ausführung gewährleisten zu können. Aus diesem Grund wird das Monitoring für die **Verfügbarkeit** von Komponenten im folgenden Unterabschnitt behandelt, gefolgt von der **Terminierung**.

Verfügbarkeit

Die Verfügbarkeit einzelner Komponenten spielt für das korrekte Deployment eine wichtige Rolle. Dabei muss man zwischen zwei Arten der Verfügbarkeit unterscheiden. Der erste Punkt ist die Überwachung der Verfügbarkeit der Deploymentumgebung. Als Deploymentumgebung wird der Container oder die VM bezeichnet, welche für das Deployment einer Anwendung notwendig ist. Der zweite Punkt gilt der Verfügbarkeit der Anwendung. Eine Anwendung ist verfügbar, wenn diese betriebsbereit ist und ihre Funktionalität in Anspruch genommen werden kann.

Für das korrekte Deployment einer Datenpipeline ist es notwendig die Abhängigkeiten zwischen den Komponenten im Prozess zu beachten. Durch das Monitoring der Verfügbarkeit wird es ermöglicht eine Reihenfolge für das Deployment erzeugen. Typischerweise sind Filter und Datensenken abhängig von zuvor deployten Komponenten, durch welche sie mit Daten versorgt werden. Ein Monitoring der Verfügbarkeit würde es somit ermöglichen, die einzelnen Bestandteile einer Datenpipeline in der

korrekten Reihenfolge zu initiieren. Um dies zu realisieren könnte das Deployment einer Komponente mit eingehenden Abhängigkeiten so ablaufen, dass alle in der Hierarchie davorstehenden Knoten deployed und verfügbar sein müssen, bevor die Komponente selbst provisioniert wird.

Terminierung

Ähnlich wie die Verfügbarkeit spielt auch die Terminierung eine wichtige Rolle für das Monitoring der Anwendung. Terminierung bezeichnet dabei den Übergang einer Komponente vom verfügbaren Zustand zu einem nicht mehr verfügbaren Zustand. Analog zur Verfügbarkeit kann auch dieser Aspekt entweder im Kontext einer Anwendung oder einer Deploymentumgebung betrachtet werden.

Eine Möglichkeit der Terminierung von Anwendung, sowie Deploymentumgebung, stellt das Löschen einer gesamten Datenpipeline dar. In diesem Fall bietet das Monitoring eine Möglichkeit die Korrektheit des angestoßenen Löschvorgangs zu verifizieren. Sind alle Anwendungen und Container oder VMs entfernt, ist die Datenpipeline auch erfolgreich entfernt worden. Je nach Anwendungsfall kann es notwendig sein die Reihenfolge der Terminierung aufgrund von Abhängigkeiten zwischen deployten Komponenten zu beeinflussen. Durch das Monitoring kann die Reihenfolge der terminierten Komponenten analog zur vorher beschrieben Verfügbarkeit beeinflusst werden, um Probleme innerhalb der noch laufenden Komponenten zu minimieren.

Weiterhin bietet das Monitoring der Terminierung auch Möglichkeiten zur Erkennung und Behebung von unvorhergesehenen Fehlerzuständen. Auch dies kann auf Ebene der Anwendung oder Deploymentumgebung stattfinden. Erkennt das System den Ausfall einer deployten Anwendung, können Maßnahmen ergriffen werden, um diesen Fehlerzustand zu beheben. Eine Möglichkeit wäre der Aufruf einer Rückruffunktion, welche auf solch ein Ereignis reagiert, und die Anwendung neu startet. Die unvorhergesehene Terminierung der Deploymentumgebung stellt ebenfalls ein schwerwiegendes Problem dar. Ein Container bzw. eine VM könnte beispielsweise durch den Neustart des Rechners, auf welcher sie deployt ist, hervorgerufen werden. Tritt dieser Fall ein, ermöglicht das Monitoring entsprechend auf das Ereignis zu reagieren, indem zum Beispiel ein neuer Container deployt wird, um so den alten zu ersetzen.

6 Prototypische Implementierung des Deploymentkonzepts

In diesem Kapitel wird die Implementierung des Prototyps geschildert, welcher als verteiltes System entwickelt wurde und das zuvor erarbeitete Konzept umsetzt. Abschnitt 6.1 beinhaltet eine Übersicht des Prototyps. Innerhalb des Abschnitts werden das zugrundeliegende Anwendungsszenario, die verwendeten Technologien und eine Übersicht der Architektur vorgestellt. Anschließend wird in Abschnitt 6.2 das Modellierungswerkzeug erläutert, welches es dem Nutzer ermöglicht Datenpipeline-Modelle zu erstellen. In Abschnitt 6.3 werden der Prozess des Deployments und der dafür verwendete Container-Orchestrator betrachtet. Abschließend geht es in Abschnitt 6.4 um verschiedene Funktionalitäten, die während der Ausführung genutzt werden. Dazu zählt die Kommunikation, das Monitoring und die Funktionsweise der deployten Datenpipeline.

6.1 Übersicht

In diesem Abschnitt wird der Prototyp zunächst allgemein vorgestellt. Hierfür wird in Abschnitt 6.1.1 das gewählte Anwendungsszenario erläutert, welches als Basis für die Implementierung dient. Abschnitt 6.1.2 beschäftigt sich mit den ausgewählten Technologien. Abschließend wird die Gesamtarchitektur des entwickelten Prototyps in Abschnitt 6.1.3 erläutert, um ein besseres Bild über die einzelnen Komponenten und deren Zusammenhänge zu schaffen.

6.1.1 Anwendungsszenario

Da es im Bereich der VFs viele verschiedene Anwendungsfälle für Datenpipelines gibt, wurde ein Anwendungsszenario als Basis für die Implementierung des Prototyps festgelegt, welches in diesem Abschnitt näher vorgestellt wird. Das Ziel des Anwendungsszenarios ist es, den Kraftstoffverbrauch vorbeifahrender VFs zu messen. Durch die gesammelten Daten können Maßnahmen zur Reduzierung des Verbrauchs besser geplant werden. Zu diesem Anwendungsszenario gehören verschiedene Teilnehmer, welche in Abbildung 6.1 dargestellt sind. In diesem Szenario empfängt eine RSU Daten von verschiedenen Fahrzeugen sowie von einem Wetterdienst. Empfängt die RSU Nachrichten der Teilnehmer, sendet sie diese im Anschluss an einen Cloud-Service weiter.

Eine Datenpipeline mit diesen Teilnehmern wurde als Anwendungsszenario ausgewählt, um nähere Informationen über Verbrauchswerte der vorbeifahrenden Fahrzeuge sammeln zu können. Als Datenquelle für diese Pipeline wurden VFs ausgewählt. Diese schicken relevante Daten an die RSU, welche sich in ihrem Kommunikationsradius befindet. Zu diesen Daten gehören Informationen zur Identifizierung des Fahrzeugs, sowie Informationen zum aktuellen Zustand der Fortbewegung. Da Emissionsdaten von externen Faktoren abhängen können, beinhaltet das Anwendungsszenario eine

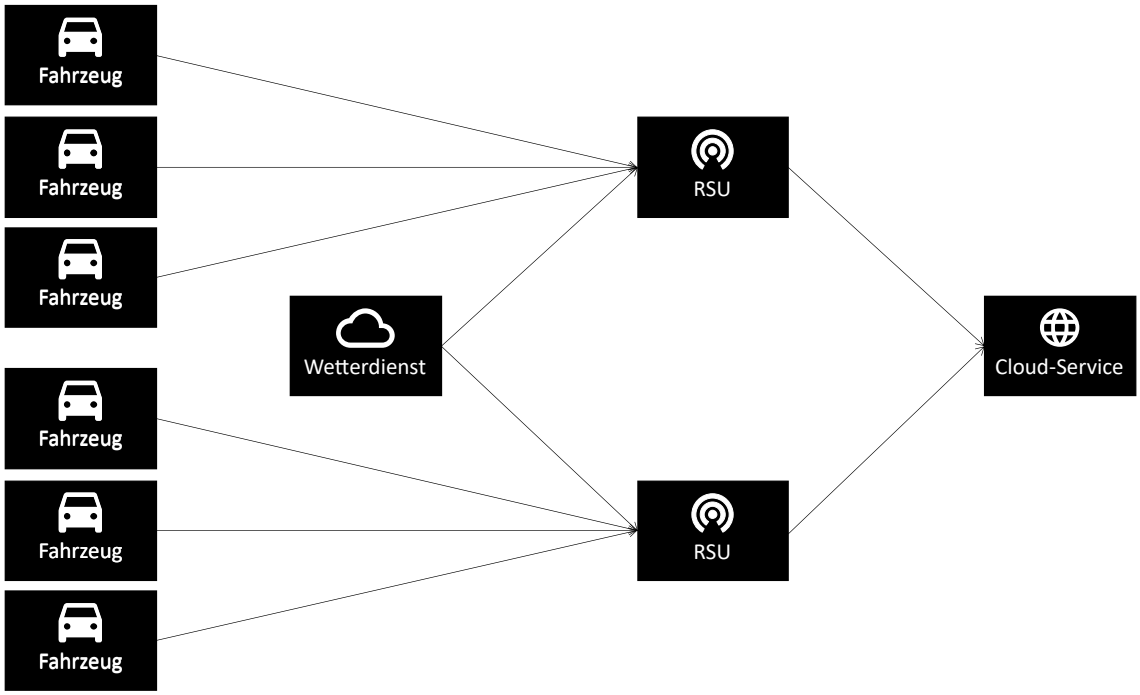


Abbildung 6.1: Übersicht der beteiligten Teilnehmer und deren Kommunikationsverbindungen im Anwendungsszenario.

weitere Datenquelle, den Wetterdienst. Dieser versorgt die RSU mit aktuellen Wetterdaten, wie der momentanen Temperatur und Windgeschwindigkeiten, welche sich auf den Verbrauch der VFs auswirken können.

Auf den verschiedenen RSUs innerhalb des Systems soll eine Software deployed werden, welche für die Datenfilterung zuständig ist. Dadurch bildet diese Instanz einen sogenannten Filter. Typischerweise fallen sehr viele Informationen von einem Software-Defined-Car an, welche für die Berechnung des Verbrauchs nicht notwendig sind. Ebenso verhält es sich mit dem Wetterdienst. Durch die Filterung innerhalb des Knotens ist es möglich, den weiteren Datenstrom zu reduzieren und nur relevante Informationen weiterzuleiten. Das Resultat dieser Datenverarbeitung wird anschließend an den letzten Teilnehmer innerhalb des Anwendungsszenarios gesendet, den Cloud-Service. Dieser stellt eine Datensenke dar und soll die Informationen der diversen Fahrzeuge und Filter aggregieren. Dadurch ist es möglich die Verbrauchswerte von sämtlichen Fahrzeugen innerhalb des deployten Systems zu betrachten.

Die gesammelten Informationen könnten anschließend für verschiedene Parteien von Interesse sein. Durch die steigende Anzahl an Fahrzeugen auf den Straßen und die dadurch erhöhten CO_2-Emissionen, bietet sich die Möglichkeit Städte oder Straßenabschnitte im Hinblick auf die dort vorherrschenden Emissionen zu beobachten. Diese Informationen könnten für die Stadt- und Verkehrsplanung relevant sein, beispielsweise um Emissions-Hotspots zu identifizieren und Maßnahmen für Verbesserungen herzuleiten. Ebenfalls besteht die Option mit diesem Wissen innerhalb der Automobilbranche einen Mehrwert zu generieren. Mithilfe des vorgeschlagenen Ansatzes wäre es möglich, den Herstellern einen Zugriff auf die Emissionsdaten zu gewähren. Durch die Überwachung

der hergestellten Fahrzeuge könnten ursprünglich aufwändige Verbrauchsmessungen vereinfacht werden. Zusätzlich bietet sich die Möglichkeit zur Betrachtung des gesamten Flottenverbrauchs eines Herstellers unter reellen Umständen, wodurch beispielsweise das Bundes-Immissionsschutzgesetz besser verfolgt werden kann.

6.1.2 Verwendete Technologien

Bei der Implementierung des Prototyps wurden diverse Technologien eingesetzt, welche in diesem Abschnitt vorgestellt werden. Als Modellierungswerkzeug wurde ein bereits bestehender Prototyp von Kenzler [Ken21] genutzt und erweitert. Dieser wurde als Browseranwendung mittels Javascript[1] und React[2] umgesetzt. Die Zeichenfläche innerhalb dieser Anwendung wurde mithilfe des Frameworks Rete.js[3] implementiert und dient der Darstellung des Modells. Für die Zustandsverwaltung wurde auf eine persistente Datenspeicherung verzichtet. Stattdessen wurde die „Store“ Komponente des Redux[4] Frameworks genutzt, um den Zustand innerhalb des Hauptspeichers zu verwalten.

Die Logik zur Verarbeitung des Modells, als auch die Vorbereitung des Deployments, wurden mittels eines eigens entwickelten Services umgesetzt, welcher auf Typescript[5] und Express[6] basiert. Diese Anwendung kommuniziert mit einem Kubernetes Cluster, welches lokal mittels Minikube[7] aufgesetzt wurde. Kubernetes ist ein Container-Orchestrierungswerkzeug, mit welchem die modellierte Datenpipeline deployed, überwacht und verwaltet werden kann. Die einzelnen Container, welche die Datenquellen, Filter und Datensenken darstellen, wurden mittels Docker erstellt. Diese datenverarbeitenden Schritte wurden ebenfalls mit Typescript und Express implementiert. Aufgrund ihrer geringen Ressourcen und einfachen Handhabung wurden gebündelte Softwarepakete, auch Images genannt, mit Docker erstellt und in Docker Hub[8] hochgeladen.

Für die Kommunikation wurden zwei verschiedene Kommunikationsmuster gewählt. Sämtliche Komponenten und externe Services, welche notwendig sind, um eine Datenpipeline zu deployen, zu überwachen und zu verwalten, kommunizieren mittels Request-Reply. Innerhalb der deployten Datenpipeline wird die Kommunikation hingegen mittels Apache Kafka[9] gehandhabt, einer Open-Source-Plattform für verteiltes Ereignis-Streaming. Kafka funktioniert dabei nach dem Publish-Subscribe-Kommunikationsmuster, bei dem Konsumenten die Nachrichten der Produzenten über sogenannte „Topics“ abonnieren können. Für das Betreiben von Kafka-spezifischen Komponenten wurde Apache Zookeeper[10] genutzt. Zookeeper dient dabei als zentraler Steuerungsknoten für die Verwaltung aller Metadaten über Kafka-Produzenten, -Broker und -Konsumenten.

[1] Javascript: https://262.ecma-international.org/12.0/
[2] React: https://reactjs.org/
[3] Rete.js: https://rete.js.org/
[4] Redux: https://redux.js.org/
[5] Typescript: https://www.typescriptlang.org/
[6] Express: https://expressjs.com/de/
[7] Minikube: https://minikube.sigs.k8s.io/docs/
[8] Docker Hub: https://hub.docker.com/
[9] Apache Kafka: https://kafka.apache.org/
[10] Apache Zookeeper: https://zookeeper.apache.org/

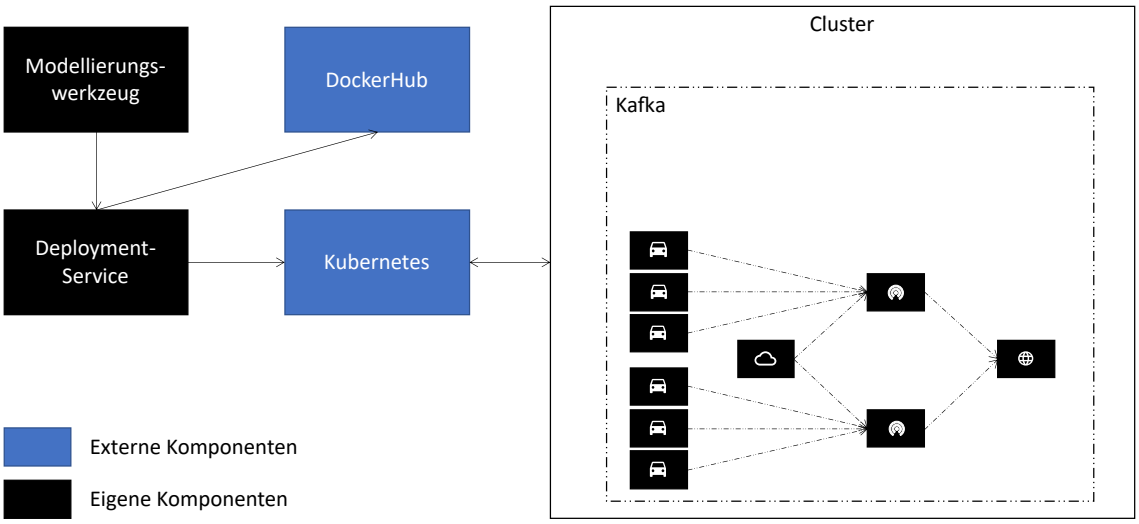


Abbildung 6.2: Übersicht der Architektur des implementierten Prototyps. Unterscheidung zwischen externen und selbst entwickelten Komponenten und Veranschaulichung der Kommunikation.

6.1.3 Architektur

In diesem Abschnitt wird die Architektur des entwickelten Prototyps dargestellt, um ein besseres Verständnis für die einzelnen Komponenten und deren Zusammenspiel zu entwickeln. Eine Übersicht dessen ist in Abbildung 6.2 zu finden. Um ein besseres Verständnis zu schaffen werden im Folgenden die einzelnen Bestandteile der Abbildung näher erläutert.

Das Modellierungswerkzeug ist eine Webanwendung, welche es dem Nutzer ermöglicht Modelle für Datenpipelines zu erstellen. Dies geschieht über eine Modellierungsoberfläche, mittels welcher Datenquellen, Filter und Datensenken erzeugt, sowie mit Metadaten ergänzt werden können. Die Logik innerhalb dieser Komponente fokussiert sich auf die grafische Darstellung eines Modells und ist von der Business-Logik für das Deployment entkoppelt. Aus diesem Grund besitzt das Modellierungswerkzeug eine Verbindung zum Deployment-Service.

Der Deployment-Service beinhaltet die Logik, welche für die Überführung von Modell zu deployter Pipeline notwendig ist. Hierfür besitzt der Service eine API, durch welche das Modellierungswerkzeug mit ihm interagieren kann. Wird ein Modell empfangen, wird dieses innerhalb der Komponente verarbeitet und anschließend mithilfe von Kubernetes, einer externen Komponente, deployed.

Für solch ein Deployment werden Docker Images genutzt. Diese ermöglichen es portable Softwarepakete zu erstellen, wodurch ein Deployment mittels Kubernetes simplifiziert wird. Um Informationen über verfügbare Images zu erhalten, besitzt der Deployment-Service eine weitere Schnittstelle, über welche DockerHub angebunden wird. Dieser externe Service ermöglicht es, Informationen zu bestehenden Images abzufragen, um diese im Modellierungswerkzeug anzeigen zu können.

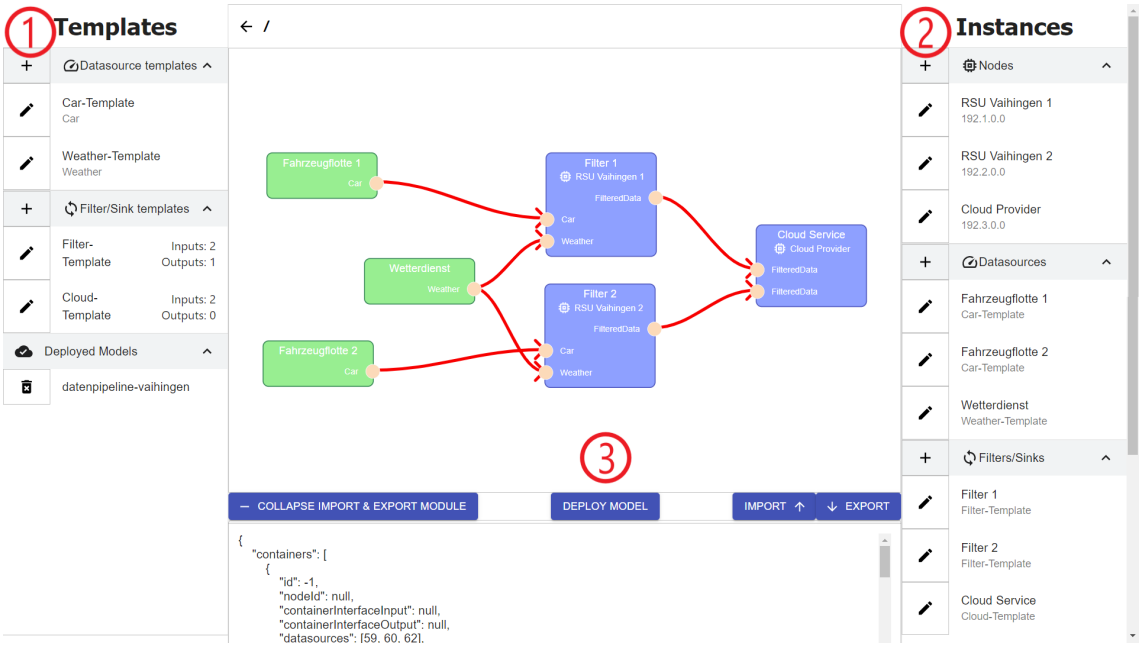


Abbildung 6.3: Übersicht des Modellierungswerkzeugs mit verschiedenen Markierungen und einer Modellierung des Anwendungsszenarios als Datenpipeline-Modell.

Um eine Datenpipeline zu deployen wird das empfangene Modell verarbeitet und anschließend mittels der Kubernetes-API deployt. Zusätzlich bietet die API Möglichkeiten Zustände von Modellbestandteilen abzufragen, wodurch das Monitoring realisiert werden kann. Für das Deployment der Pipeline wird mittels Kubernetes ein sogenanntes Cluster erstellt, eine Gruppe von Knoten, auf denen containerisierte Anwendungen laufen.

Die Kommunikation zwischen den bisher genannten Komponenten wird mittels Request-Reply-Muster umgesetzt. Innerhalb des Kubernetes-Clusters, in welchem die Datenpipeline deployt wird, kommunizieren die Komponenten hingegen mittels Kafka, was durch die gestrichelten Linien in Abbildung 6.2 dargestellt ist.

6.2 Modellierung

Dieser Abschnitt befasst sich mit der Modellierung. Für die Realisierung wurde eine Umsetzung als Webanwendung gewählt, welche es Nutzern erleichtern soll Modelle zu erstellen. Als Grundlage dient hierfür der Prototyp von Kenzler [Ken21], welcher angepasst und durch gewisse Funktionalitäten erweitert wurde, welche für ein automatisiertes Deployment notwendig waren. Im Folgenden wird das Modellierungswerkzeug vorgestellt, welches für die Modellierung von Datenpipelines entwickelt wurde. Zusätzlich werden die Anpassungen und Erweiterungen erklärt, welche durch den Autor an der bereits bestehenden Software vorgenommen wurden.

Wie in Abbildung 6.3 zu sehen ist, besteht die Anwendung aus drei Teilen. Diese sind mit den Markierungen ① bis ③ gekennzeichnet. Die Bestandteile werden in den folgenden Unterabschnitten vertieft, wobei sich der letzte Unterabschnitt mit den funktionalen Änderungen beschäftigt, die notwendig waren, um die Kommunikation mit dem Deployment-Service zu ermöglichen.

6.2.1 Templates

In der linken Spalte des Modellierungswerkzeugs, markiert mit dem Symbol ①, können drei verschiedene Menüs vorgefunden werden. Die ersten zwei Menüs ermöglichen es dem Nutzer Templates zu erstellen und zu bearbeiten. Templates ermöglichen es Wiederverwendbarkeit für die Erstellung von Datenquellen, Filtern und Datensenken zu erzeugen. Bei der Erstellung eines Templates werden gewisse Metadaten vorgegeben, welche für alle Instanzen des Templates gelten. Durch einen Klick auf den +-Knopf eines Menüs ist es möglich ein neues Template anzulegen. Bearbeiten lässt sich solch ein Template durch die Schaltfläche links neben dem entsprechenden Listenelement. Das dritte Menü in diesem Bereich beinhaltet die bereits deployten Datenpipelines und wurde zusätzlich zum bereits bestehenden Prototypen hinzugefügt. Die einzelnen Menüs und ihre Funktionalitäten werden in den folgenden Unterabschnitten vertieft.

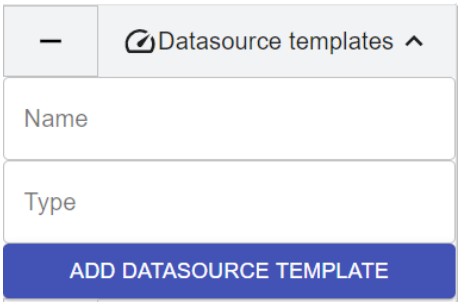
Templates für Datenquellen

Das „Datasource Templates“ Menü bietet die Möglichkeit Templates für Datenquellen zu erstellen und zu bearbeiten. In Abbildung 6.3 sind zwei Templates sichtbar, das Car-Template und das Weather-Template, welche in einer Liste angezeigt werden. Diese stellen wiederverwendbare Schablonen für die Instanz eines VF beziehungsweise eines Wetterdienstes dar, welche die Datenquellen aus dem Anwendungsszenario widerspiegeln. Neben dem Namen des Templates wird zusätzlich der Datentyp angezeigt. Des Weiteren bietet die Schaltfläche eine Möglichkeit zur Bearbeitung oder Erstellung eines Templates, wie in Abbildung 6.4 gezeigt wird.

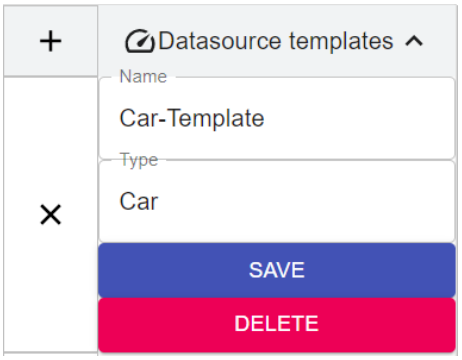
In Abbildung 6.4a wird die Maske für das Erstellen eines Templates für Datenquellen gezeigt. Für die Erstellung muss ein Name und ein Typ für das Template angegeben werden. Nach dem Hinzufügen wird eine ID generiert, durch welche das Template später zu Instanzen zugeordnet werden kann. Abbildung 6.4b zeigt ein zuvor erstelltes Template im Bearbeitungsmodus. In dieser Ansicht können zuvor hinterlegte Informationen geändert, oder das Template gelöscht werden.

Templates für Filter

Der zweite Menüpunkt bezieht sich auf das Erstellen und Verwalten von Templates für Filter und Datensenken. Beim Erstellen eines solchen Templates müssen ein Name, sowie die Verbindungen spezifiziert werden, wie in Abbildung 6.5a dargestellt ist. Der *Add Input*-Knopf bietet die Möglichkeit eingehende Datenverbindungen anzulegen, während der *Add Output*-Knopf das Anlegen ausgehender Verbindungen ermöglicht. Optional bietet sich die Möglichkeit den Filter als Container zu deklarieren. Mit dieser Option können Komponenten innerhalb anderer Komponenten geschachtelt werden. Dadurch ist es möglich die Anzeige des Modells in verschiedenen Ebenen darzustellen und somit Abstraktionen für komplexe Bestandteile zu ermöglichen.

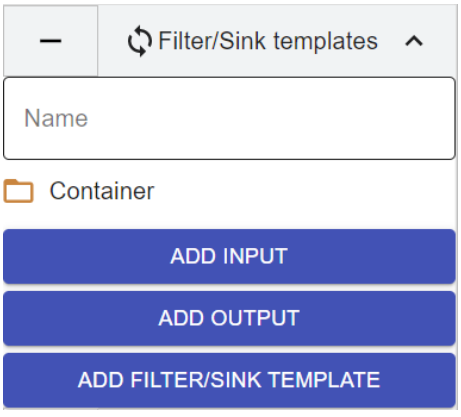


(a) Menü zum Hinzufügen eines neuen Templates für Datenquellen.

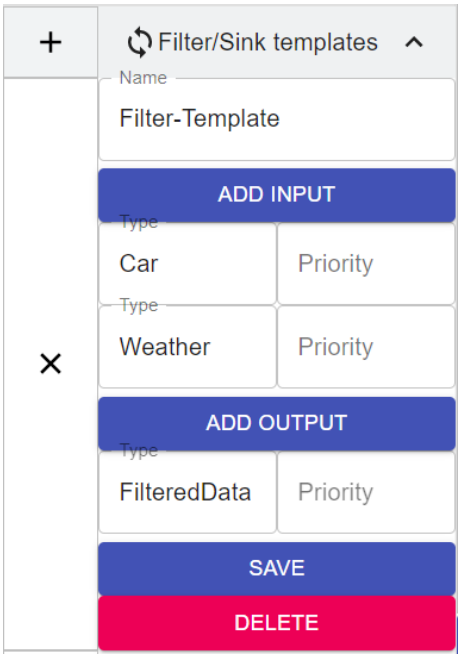


(b) Menü zum Editieren eines Templates für Datenquellen.

Abbildung 6.4: Übersicht über das Hinzufügen und Editieren von Templates für Datenquellen.

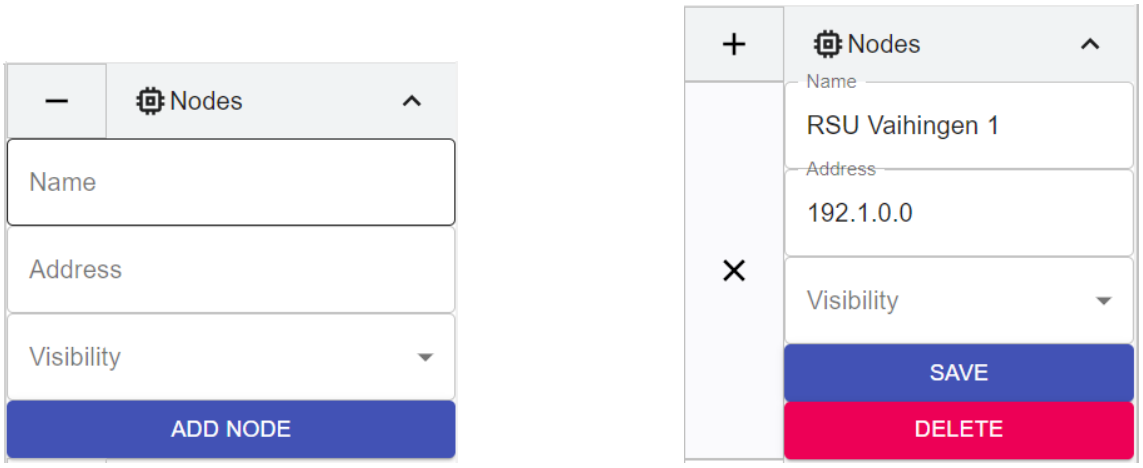


(a) Menü zum Hinzufügen eines neuen Templates für Filter und Datensenken.



(b) Menü zum Editieren eines Templates für Filter und Datensenken.

Abbildung 6.5: Übersicht über das Hinzufügen und Editieren von Templates für Filter und Datensenken.



(a) Menü zum Hinzufügen eines neuen Knotens.

(b) Menü zum Editieren eines bestehenden Knotens.

Abbildung 6.6: Übersicht über das Hinzufügen und Editieren von Knoten.

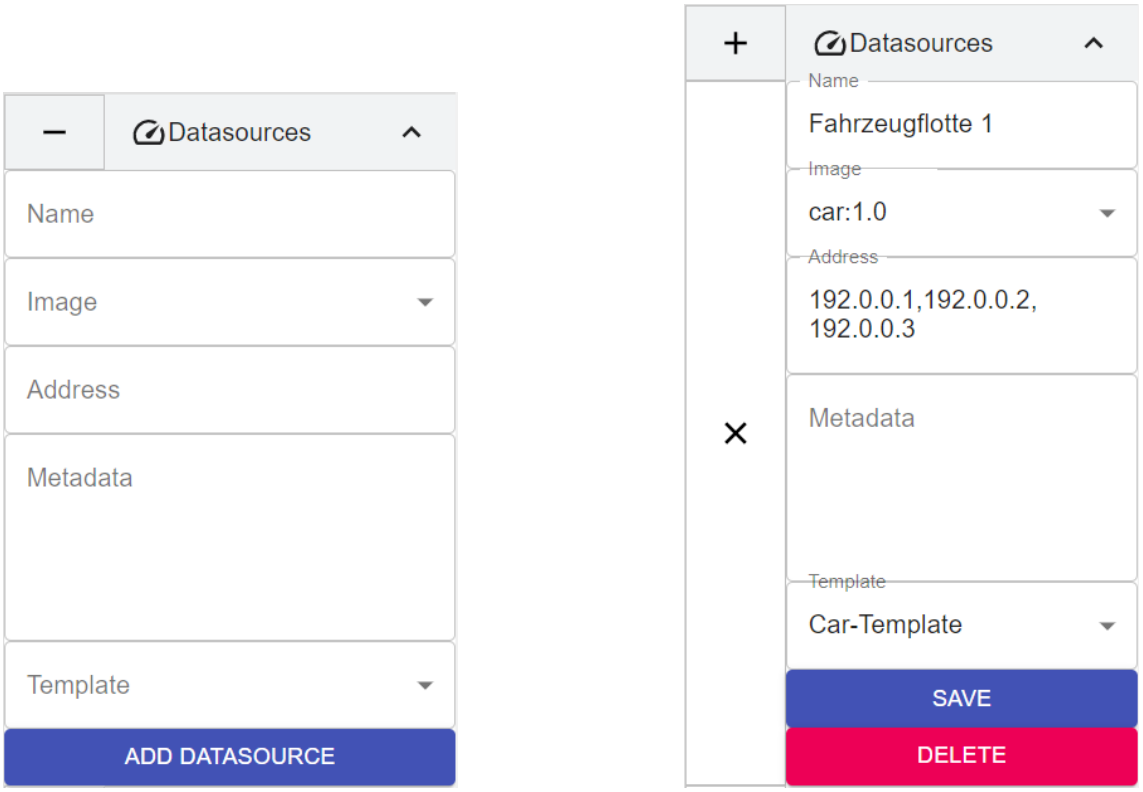
Der größte Unterschied im Hinblick auf Datenquellen-Templates ist die Möglichkeit der Spezifizierung für eingehende Kanten. Abbildung 6.5b zeigt ein bestehendes Filter-Template mit insgesamt drei Verbindungen. Dabei ist spezifiziert, dass Eingänge vom Typ „Car“ und „Weather“ akzeptiert sind, während die ausgehende Verbindung als „FilteredData“ typisiert ist. Einer Verbindung kann außerdem eine Priorität zugewiesen werden. Da Prioritäten innerhalb des Anwendungsszenarios keine Rolle spielen, war eine Prioritätensetzung in dieser Arbeit jedoch nicht notwendig.

Deployte Modelle

Der letzte Menüpunkt stellt eine der Veränderungen innerhalb des Modellierungswerkzeugs dar, welches für die Verwaltung von deployten Datenpipelines neu implementiert wurde. Diese Liste stellt eine Übersicht über sämtliche Datenpipelines dar, welche momentan deployt sind. Durch die Kommunikation mit dem Deployment-Service wird das automatische Aktualisieren dieser ermöglicht. Der Knopf, welcher sich neben einem deployten Modell befindet, bietet die Möglichkeit eine Datenpipeline zu löschen. Im Hintergrund wird für diese Aktion eine Anfrage an den Deployment-Service gesendet. In der Zwischenzeit verändert sich das Symbol, um einen Ladezustand zu signalisieren. Nach einem erfolgreichen Entfernen der Datenpipeline aus dem Kubernetes Cluster wird das Modell aus der Liste entfernt.

6.2.2 Instanzen

In diesem Abschnitt geht es um die drei verschiedenen Menüpunkte, welche die rechte Spalte des Modellierungswerkzeugs bilden und mit dem Symbol ② in Abbildung 6.3 gekennzeichnet sind. Hier kann der Nutzer die für das Deployment notwendigen Komponenten wie Knoten, Datenquellen, Filter und Datensenken erstellen. Die einzelnen Menüpunkte werden in den folgenden Abschnitten näher beschrieben.



(a) Menü zum Hinzufügen einer neuen Datenquelle.

(b) Menü zum Editieren einer bestehenden Datenquelle.

Abbildung 6.7: Übersicht über das Hinzufügen und Editieren von Datenquellen.

Knoten

Ein Knoten bezeichnet eine physische Ressource auf die eine Komponente der Datenpipeline deployt werden soll. Durch die Angabe eines Knotens ist es somit möglich, einzelne Verarbeitungsschritte auf verschiedene Rechner aufzuteilen, oder mehrere Komponenten innerhalb des selben Knotens zu deployen. Diese Funktionalität ist nur für Filter und Datensenken vorgesehen, da Datenquellen gesondert behandelt werden.

Abbildung 6.6a zeigt die Maske für die Erstellung eines solchen Knotens. Neben dem Namen ist es notwendig eine Adresse zu spezifizieren, wodurch das Deployment auf dem korrekten Rechner durchgeführt werden kann. Optional kann eine Sichtbarkeit ausgewählt werden, wodurch ein Knoten als öffentlich oder privat deklariert wird. Im Editierungsmodus, abgebildet in Abbildung 6.6b, lassen sich die zuvor eingetragenen Werte ändern. Ebenso kann der Knoten hier entfernt werden.

Datenquellen

Datenquellen sind Komponenten der Datenpipeline, welche das System mit Daten versorgen. Im vorgestellten Anwendungsszenario wurden VFs und ein Wetterdienst als solche gewählt. Abbildung 6.7a zeigt welche Informationen für das Deployment einer Datenquelle notwendig sind.

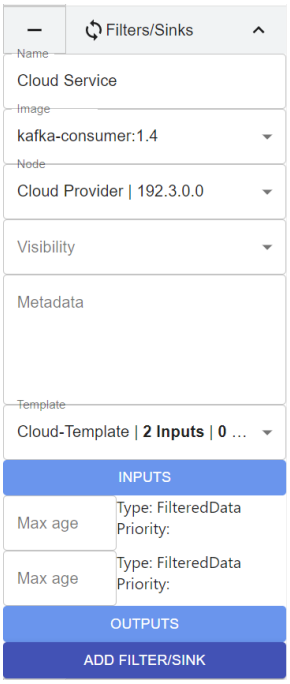
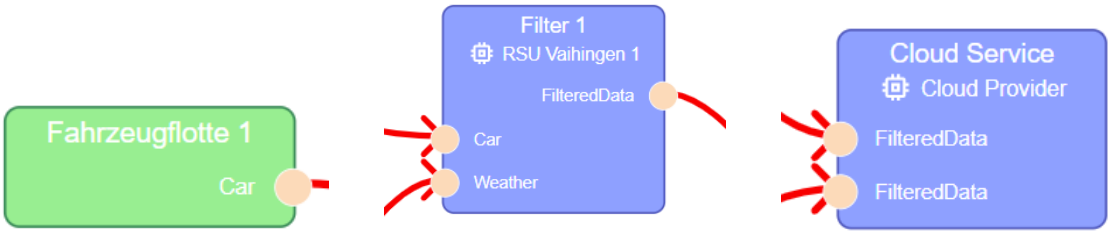


Abbildung 6.8: Menü für das Hinzufügen eines Filters oder einer Datensenke.

Zur Identifizierung kann ein Name vergeben werden. Mit der Wahl eines Templates muss der Datentyp nur einmal für mehrere Datenquellen deklariert werden. Das Spezifizieren einer Adresse ermöglicht es, die Datenquelle unter der richtigen Adresse zu deployen. Während dieses Feld bereits gegeben war, wurde der Input-Typ verändert, wodurch eine Komma-separierte Liste von Adressen übergeben werden kann. Dies ist in Abbildung 6.7b abgebildet und ermöglicht dem Nutzer das mehrfache Deployment von Datenquellen. Während es typischerweise wenige Datensenken und einige Filter gibt, können ITS-Systeme sehr viele Datenquellen besitzen. Durch dieses Konzept konnte das n-malige Modellieren eines Fahrzeugs vermieden werden. Eine weitere Ergänzung der Implementierung fand durch das Hinzufügen der Image-Auswahl statt. Hier kann der Nutzer spezifizieren, welches Docker-Image unter der angegebenen Adresse deployt werden soll. Das Dropdown-Menü ist dabei mit dem Deployment-Service verknüpft, welcher das Modellierungswerkzeug mit einer Liste von aktuellen Docker-Images versorgt. Mittels optionaler Metadaten können weitere Informationen für eine Datenquelle hinterlegt werden.

Filter und Datensenken

Der unterste Menüpunkt der rechten Spalte bietet Funktionalitäten zur Erstellung und Verwaltung von Filtern und Datensenken an. Abbildung 6.8 zeigt die Erstellung einer Datensenke und alle möglichen Eingabewerte. Für das Deployment sind die Eingabe eines Namens, die Auswahl eines Images, eines Knotens und eines Templates notwendig. Wie bei den Datenquellen wurde die Auswahl eines Images als Zusatz implementiert, um ein deploybares Softwarepaket spezifizieren zu



(a) Beispiel einer modellierten Datenquelle.

(b) Beispiel eines modellierten Filters.

(c) Beispiel einer modellierten Datensenke.

Abbildung 6.9: Beispiele für verschiedene modellierte Instanzen.

können. Durch die Auswahl eines Knotens kann das Deployment zusätzlich beeinflusst werden. Darüber hinaus ist es möglich eine Sichtbarkeit einzustellen, Metadaten zu hinterlegen und die Verbindungen durch das Spezifizieren eines maximalen Alters für Nachrichten zu beeinflussen.

6.2.3 Editor

Der Editor befindet sich in der Mitte der Anwendung und ist mit dem Symbol ③ in Abbildung 6.3 markiert. Im Folgenden werden die einzelnen Bestandteile des Editors vorgestellt. Hierfür wird zunächst die Zeichenfläche vorgestellt, gefolgt von der Import/Export Funktionalität. Abschließend wird die neu hinzugefügte Funktionalität für das Anstoßen eines Deployments erläutert.

Zeichenfläche

Die Zeichenfläche stellt eine grafische Repräsentation des erstellten Modells dar und besteht aus Knoten und Kanten. Knoten setzen sich dabei aus Informationen der erstellten Templates und Instanzen zusammen. Dabei werden Komponenten als abgerundete Rechtecke dargestellt, welche den hinterlegten Namen als Überschrift besitzen. Für Filter und Datensenken werden zusätzlich die gewählten Knoten angezeigt. Alle Komponenten besitzen zudem eine grafische Darstellung der Datentypen für ihre ein- und ausgehenden Datenverbindungen, was in Abbildung 6.9 abgebildet ist. Verbindungen werden innerhalb der Zeichenfläche als gerichtete Pfeile dargestellt und können durch den Nutzer angelegt werden. Sämtliche dargestellte Informationen basieren auf der Grundlage des Redux-Stores. Dieser erlaubt es, den Zustand innerhalb des Hauptspeichers zu speichern und auf Veränderungen des Zustands zu reagieren. Werden Änderungen an den Daten vorgenommen, wird die interne Repräsentation des Modells verändert und automatisch in der Zeichenfläche angepasst.

Import/Export

Um eine Wiederverwendbarkeit zu gewährleisten, existieren eine Import- und Exportfunktion. Die Exportfunktion ermöglicht es durch das Klicken des entsprechenden Feldes die interne Darstellung des Modells im JavaScript Object Notation (JSON)-Format ausgeben zu lassen. Das Modell im

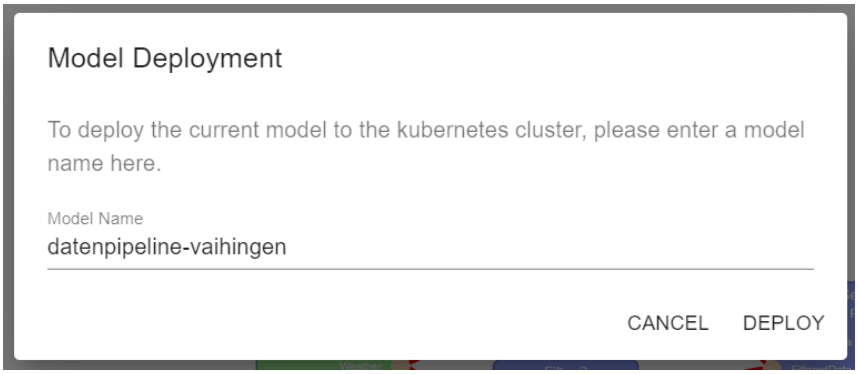


Abbildung 6.10: Popup nach dem Klick auf den *Deployment*-Knopf.

JSON-Format kann anschließend vom Benutzer gespeichert werden. Durch die Importfunktion wird es dem Nutzer ermöglicht, ein bereits bestehendes Modell im JSON Format als Eingabe zu definieren. Dadurch kann das gespeicherte Modell geladen und anschließend angezeigt werden.

Deployment

Um das erstellte Modell zu deployen wurde innerhalb dieser Arbeit ein neues Popup-Fenster implementiert, welches in Abbildung 6.10 abgebildet ist. Nachdem die für ein Deployment notwendigen Informationen erstellt wurden, kann der Nutzer diese durch den Klick der *Deployment*-Schaltfläche öffnen. Hier ist es möglich dem Deployment einen Namen zu geben. Dieser ist notwendig, um das deployte Modell später wieder identifizieren und entfernen zu können. Hat der Nutzer einen Namen ausgewählt und sich entschieden fortzufahren, wird mit einer Anfrage an den Deployment-Service das Deployment gestartet.

6.2.4 Funktionale Änderungen

Dieser Abschnitt beschäftigt sich mit neu eingeführten Funktionalitäten, welche für ein Deployment notwendig sind. Um diese einzuführen, wurde das bestehende Projekt an diversen Stellen abgeändert. Hierbei wurde der Quellcode betroffener Dateien von Javascript zu Typescript konvertiert. Der größte Mehrwert war dabei die Möglichkeit der Typisierung und die Einführung von Klassen und Interfaces. Durch diese Anpassung wird eine leichtere Wartung und Erweiterbarkeit geschaffen, da Linter und Compiler besser genutzt werden können, wodurch ein besseres Debugging ermöglicht wird. Listing 6.1 zeigt ein Beispiel für die Typisierung, welches für die Erstellung von Datenquellen-Templates in Form eines Typescript Interfaces umgesetzt wurde.

Neben der Umstellung von Javascript auf Typescript wurden zusätzlich Anpassungen an bestehender Funktionalität eingebaut. Diese werden im folgenden Unterabschnitt beleuchtet. Weiterhin wurden neue Funktionen zum bestehenden Quellcode hinzugefügt, um eine Interaktion mit dem Deployment-Service zu ermöglichen, welche im darauffolgenden Unterabschnitt erklärt werden.

Listing 6.1 Typescript Interface für die Typisierung eines Datenquellen-Templates.

```
interface DatasourceTemplate {
  id: number
  name: string
  type: string
}
```

Anpassungen

Zusätzlich zur Umstellung auf Typescript wurden Teile der Logik zur Anzeige einzelner Komponenten überarbeitet. Dies war notwendig, da das Verhalten bei der Erstellung und Änderung von Instanzen und Templates nicht korrekt funktioniert hat. Bislang wurden Updates der grafischen Oberfläche wie das Expandieren oder Einklappen von Kontext-Menüs über Zustandsveränderungen im Redux-Store ausgelöst. Die bisherige Implementierung führte Updates jedoch nicht automatisch aus, sondern registrierte Veränderungen erst beim nächsten Update des Modellzustands. Dadurch konnten grundlegende Funktionalitäten erst genutzt werden, wenn mehrere Zustandsveränderungen vorgenommen wurden. Dieses Fehlverhalten betraf die Kontext-Menüs für Templates, sowie für Instanzen von Filtern, Knoten und Datenquellen.

Da React als Framework für die Implementierung gewählt wurde, konnten diese simplen Aktualisierungen der anzeigenden Schaltflächen durch sogenannte „Hooks“ ersetzt werden. Hooks sind native Funktionen des React-Frameworks, welche es ermöglichen den Zustand und Lebenszyklus einer React-Applikation zu beeinflussen. Für die Umsetzung der Aktualisierungslogik wurde der „State Hook“[11] gewählt. Mittels dieser Funktion kann der Zustand eines Objekts, beispielsweise einer Datenquellen-Instanz, gespeichert werden. Ändert sich der Zustand des gespeicherten Objekts, wird dies von der React-Anwendung automatisch wahrgenommen und die betroffenen UI-Elemente werden aktualisiert. Mit dieser Lösung war es möglich die Anzeigelogik für sämtliche Kontextmenüs aus dem Redux-Store herauszutrennen und den Quellcode zu vereinfachen.

Erweiterungen

Wie in Abschnitt 6.1 erwähnt, wurde für die Gesamtanwendung eine Architektur gewählt, in welcher die Anzeigelogik von der Businesslogik entkoppelt umgesetzt wird. Aus diesem Grund wurden verschiedene Erweiterungen zum bestehenden Modellierungswerkzeug vorgenommen, welche eine Kommunikation mit dem Deployment-Service ermöglichen. Dieser Abschnitt stellt die einzelnen Erweiterungen vor. Für die Umsetzung wurden vier verschiedene Schnittstellen eingeführt. Diese interagieren mittels Request-Reply mit dem Deployment-Service und beeinflussen so den Zustand der Gesamtanwendung und ermöglichen den Erhalt von Informationen. Die vier Schnittstellen und der Grund für ihre Notwendigkeit werden in den folgenden Paragraphen vorgestellt.

[11] State Hook: https://reactjs.org/docs/hooks-state.html

Listing 6.2 Typescript Interface für die Typisierung eines Modells mit den notwendigen Daten für ein Deployment.

```
interface MinimalModel {
  name: string
  containers: Container[]
  physicalNodes: PhysicalNode[]
  datasources: DatasourceOutput[]
  filters: Filter[]
  connections: Connection[]
}
```

Deployment Um eine modellierte Datenpipeline an den Deployment-Service zu senden, wurde zunächst ein neues Typescript Interface entwickelt, welches in Listing 6.2 dargestellt ist. Das Listing zeigt dabei das Ausgangsformat einer Funktion, welche das interne Datenformat in ein Objekt mit den minimal notwendigen Attributen überführt. Überflüssige Attribute für das Deployment stellen beispielsweise die angelegten Templates dar. Während diese zwar einen Mehrwert für die Modellierung schaffen, bieten diese beim Deployment keinen Nutzen und können somit weggelassen werden. Das resultierende Objekt besteht anschließend aus einem Namen, Containern, Knoten, Datenquellen, Datensenken, Filtern und Verbindungen.

Anschließend wird das Objekt in ein JSON Format überführt und als HTTP-Request an den Deployment-Service versandt. Als HTTP-Methode wurde die POST-Methode gewählt. Mit dem entsprechenden Modell im JSON Format wird der Body des Requests befüllt und anschließend an den Endpunkt gesendet, wo die Daten weiterverarbeitet werden können.

Undeployment Analog zum Deployment kann ein Undeployment mittels HTTP-Request an den Deployment-Service eingeleitet werden. Für das Anstoßen eines Undeployments wurde eine Liste mit deployten Modellen auf der linken Seite der Webanwendung eingeführt. Durch einen Klick der entsprechenden Schaltfläche eines deployten Modells innerhalb der Liste, kann dieses entfernt werden. Beim Aufruf der Funktion wird der Name des deployten Modells als Parameter übergeben. Dieser bildet den Body des HTTP-Requests, welcher als DELETE-Methode implementiert wurde.

Abfrage deployter Modelle Zusätzlich zum Deployment war es notwendig, eine neue Funktion für die Abfrage bereits deployter Modelle hinzuzufügen. Ein Grund dafür ist die beschränkte Möglichkeit des Frontends zur Speicherung des Zustands. Da dieser nicht in einer Datenbank festgehalten wird, sondern nur sitzungsübergreifend, ist die Anwendung darauf angewiesen die Daten durch einen externen Dienst zu erhalten.

Hierfür kann ein Endpunkt des Deployment-Service mittels GET-Methode aufgerufen werden. Im Body der Antwort befindet sich anschließend eine Liste an Namen, welche die deployten Datenpipelines widerspiegelt. Die empfangenen Informationen dienen als Grundlage für die Anzeige innerhalb der Webanwendung.

Abfrage vorhandener Images Für die Erstellung von Datensenken, Filtern und Datenquellen ist es notwendig ein Docker-Image auszuwählen. Dieses Docker-Image spiegelt das Softwarepaket wider, welches innerhalb der Datenpipeline deployt werden soll. Die Auswahl des korrekten Images wird dem Nutzer durch ein Dropdown-Menü erleichtert. Um die zur Anzeige notwendigen Informationen zu erhalten, wurde eine neue Funktion eingeführt. Diese sendet einen GET-Request an den Deployment-Service und erhält eine Liste von Namen als Antwort, welche für die entsprechende Anzeige genutzt wird.

6.3 Deployment

Für die Provisionierung eines Datenpipeline Modells wurde der Deployment-Service entwickelt. Um diese Aufgabe zu erfüllen, wurden Schnittstellen implementiert, über welche das Modellierungswerkzeug mit dem Deployment-Service kommunizieren kann. Dies ist für verschiedene Aspekte notwendig, wie die Abfrage des Zustands deployter Modelle, oder das Deployment und Undeployment neuer oder bestehender Datenpipelines. Als Deploymentstrategie wurde das Container-basierte Deployment mittels Container-Orchestrator ausgewählt. Die Orchestration wurde mittels Kubernetes umgesetzt. Abschnitt 6.3.1 beschreibt wie der Deployment-Service Kubernetes nutzt, um Datenpipelines zu deployen. Anschließend werden in Abschnitt 6.3.2 die internen Funktionalitäten des Deployment-Service erklärt, um zu verstehen, wie das empfangene Modell zu einer deployten Datenpipeline transformiert wird.

6.3.1 Kubernetes

Als Deploymentstrategie wurde die Verwendung von Docker in Kombination mit Kubernetes als Orchestrator gewählt. Einer Openstack Umfrage[12] von 2021 zufolge, sind dies die am weitesten verbreiteten Technologien in der Industrie. Unter den Openstack Nutzern gaben 67% an Docker für die Containerisierung zu nutzen. Mit 68% der Nutzer belegte Kubernetes den ersten Platz der am häufigsten verwendeten Software für das Management von Container-basierten Anwendungen.

Für das Aufsetzen eines Kubernetes Clusters werden typischerweise mehrere Rechner oder virtuelle Maschinen genutzt. Ein Cluster bezeichnet einen Verbund aus verschiedenen Knoten, welche containerisierte Anwendungen betreiben. Da es im Rahmen dieser Arbeit nicht möglich war diese Ressourcen bereitzustellen, wurde ein lokales Cluster aufgesetzt. Für diese Aufgabe wurde Minikube genutzt. Minikube ist ein frei verfügbares Werkzeug, welches es ermöglicht Kubernetes mit einem Cluster bestehend aus einem Knoten zu betreiben. Dies wird ermöglicht, in dem Kubernetes innerhalb eines Docker Containers auf dem lokalen Rechner gestartet wird.

In diesem Abschnitt werden verschiedene Konzepte von Kubernetes erklärt, welche der Deployment-Service nutzt, um eine Anwendung zu provisionieren. Abbildung 6.11 zeigt einen Ausschnitt der verwendeten Kubernetes Konzepte, welche für das Deployment einzelner Verarbeitungsschritte innerhalb der Datenpipeline genutzt werden. Die folgenden Unterabschnitte gehen auf die einzelnen Komponenten ein und erklären diese näher. Zusätzlich werden Namespaces, ein weiteres Kubernetes Konzept vorgestellt.

[12] Openstack Umfrage: https://www.openstack.org/analytics/

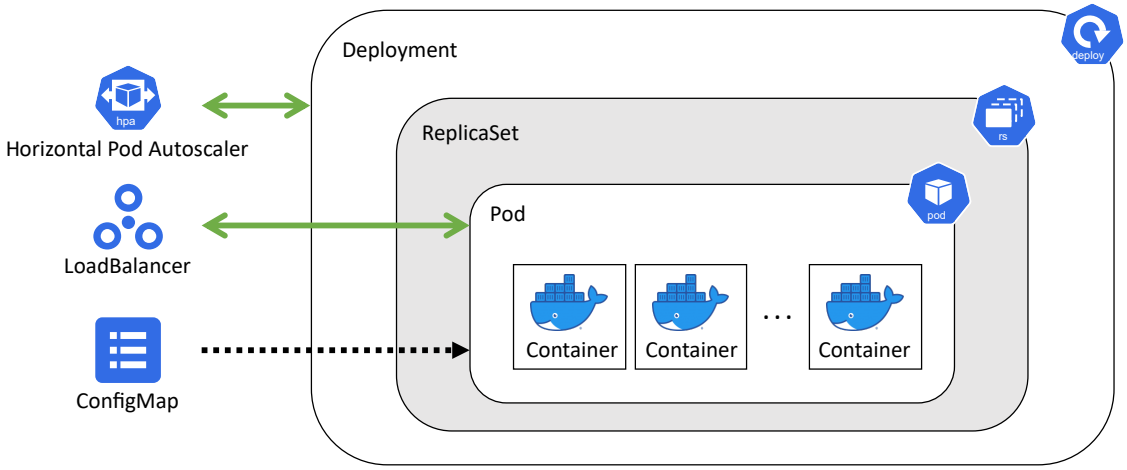


Abbildung 6.11: Übersicht verschiedener Kubernetes Konzepte, welche vom Deployment-Service genutzt werden.

Container

Die Grundlage für ein Deployment stellen Container dar. Um einen Container zu erstellen werden Images genutzt. Ein Image ist eine Datei welche alle notwendigen Informationen für die Ausführung einer Anwendung beinhaltet. Dazu zählen der Quellcode, Bibliotheken, Abhängigkeiten und sonstige Artefakte, die von der Anwendung benötigt werden, um ihre Funktionalität zu gewährleisten. Images werden durch den Nutzer vor einem Deployment angelegt. Ein Image bildet somit das Abbild einer Datenquelle, Datensenke oder eines Filters.

Ein Docker Container ist eine virtualisierte Laufzeitumgebung. Diese ermöglicht es, die im Image spezifizierten Informationen zu verwenden, um die Anwendung innerhalb des Containers zu betreiben. Somit ist es das Ziel des Deployment-Prozesses, Container für die angegebenen Images zu erstellen, wodurch eine lauffähige Anwendung entsteht.

Pod

Ein Pod stellt die kleinste deploybare Einheit in Kubernetes dar, welche erstellt und verwaltet werden kann. Pods beinhalten einen oder mehrere Container und ermöglichen es, Ressourcen wie Speicher und Netzwerk-Ressourcen zwischen Containern zu teilen. Für das Deployment der Datenverarbeitungsschritte besteht eine eins-zu-eins Zuordnung zwischen Pod und Datenverarbeitungsschritt, was das am häufigsten verwendete Prinzip für Kubernetes Deployments darstellt [Kub22]. Der Pod bildet in dieser Konstellation eine Abstraktion des Containers. Diese Abstraktionsebene ist notwendig, da Kubernetes nicht in der Lage ist Container direkt zu beeinflussen.

ReplicaSet

Das ReplicaSet ist ein Konzept, welches es ermöglicht eine Vielzahl identischer Pods zur Ausführungszeit zu garantieren. Durch die Angabe der gewünschten Replikats-Anzahl kann deklarativ festgelegt werden, wann das ReplicaSet neue Pods in Form einer Replika erstellt. Zusätzlich ermöglicht die Verwendung des ReplicaSets die Robustheit der Anwendung, sowie die Verfügbarkeit deutlich zu erhöhen.

Die Vorteile dessen kann man anhand einer deployten Filter-Komponente innerhalb einer RSU erkennen. Wenn die Anwendung, welche die Logik des Filters darstellt, nur als einzelner Pod deployt wird, können bei einem Ausfall des Pods keine Daten der vorbeifahrenden VFs mehr gesammelt werden. Wird hingegen ein ReplicaSet genutzt, ist es möglich die Anzahl der verfügbaren Replikate des Filter-Pods zu erhöhen. In diesem Fall wird die Datenpipeline nicht unterbrochen, wenn ein einzelner Pod ausfällt, da weitere identische Pods existieren, welche die nachfolgenden Anfragen entgegennehmen können.

ReplicaSets werden beim Deployment einer Datenpipeline mittels Prototyp jedoch nur indirekt genutzt. Deployments, welche im folgenden Abschnitt betrachtet werden, ermöglichen es implizit die Funktion eines ReplicaSets zu gewährleisten.

Deployment

Das Deployment ist ein Konzept für die Verwaltung von Pods und ReplicaSets. Mit einer deklarativen Syntax lässt sich so spezifizieren, welcher Zustand für einen Pod oder ein ReplicaSet gelten muss. Zur Steuerung eines ReplicaSets lässt sich innerhalb des Deployments beispielsweise die Anzahl an verfügbaren Replikas konfigurieren. Bei der Ausführung des Deployments erstellt Kubernetes dann automatisch ein ReplicaSet, welches so konfiguriert wird, dass die korrekte Anzahl an Replikas für den entsprechenden Pod gestartet werden.

Des Weiteren wird auch der zugehörige Pod durch Konfigurationen innerhalb des Deployments beeinflusst. Somit stellt ein Deployment eine Art Template für die Provisionierung von Pods dar. Wie bei der impliziten Nutzung eines ReplicaSets, wird der zugehörige Pod ebenfalls über das Deployment konfiguriert. Zu den wichtigsten Informationen für diese Konfiguration zählen das Image, die Port-Konfiguration und der Name der zugehörigen ConfigMap.

Horizontal Pod Autoscaler

Der Horizontal Pod Autoscaler ist ein Kubernetes Konzept zur automatischen Skalierung der Arbeitslast. Horizontale Skalierung bedeutet in diesem Kontext, dass die Reaktion auf eine erhöhte Last darin besteht, mehr Pods einzusetzen. Dies kann durch die Konfiguration von Grenzwerten für CPU- und Speicherauslastung konfiguriert werden. Bei einer Überschreitung der konfigurierten Werte, werden betroffene Pods automatisch durch neue Replikas erweitert.

ConfigMap

ConfigMaps stellen ein Konzept dar, welches für die Übergabe von Umgebungsvariablen genutzt wird. Sie bieten die Möglichkeit Informationen in Form von Schlüssel-Wert-Paaren zu speichern. Pods können diese Daten in Form von Umgebungsvariablen übergeben bekommen, was im Fall des Deployment-Services über das Konzept der Deployments geschieht. Mit einer ConfigMap können also umgebungsspezifische Informationen von den im Pod verwendeten Images entkoppelt werden. Dieses Konzept wird im Prototypen für die korrekte Konfiguration der Messaging-Anbindung genutzt, welche für die Kommunikation innerhalb der Datenpipeline essenziell ist.

LoadBalancer

Für jeden datenverarbeitenden Schritt innerhalb der Datenpipeline wird ein LoadBalancer erstellt. Dieser zählt zum Konzept der Services in Kubernetes, welche für die Kommunikation zu und von Pods genutzt werden. Durch die Nutzung von Deployments in Kubernetes, werden Pods dynamisch erstellt und gelöscht. Da jeder Pod bei der Erstellung eine neue IP-Adresse zugewiesen bekommt, wäre es notwendig eine Art Service-Discovery einzusetzen, um die Anwendungen an wechselnde IP-Adressen anzupassen.

Aus diesem Grund wurden Services in Kubernetes als Abstraktionsebene eingeführt und für die Provisionierung von Datenpipelines verwendet. Sie ermöglichen es die Kommunikation mit Pods deutlich zu vereinfachen. Ankommende Anfragen werden nicht direkt an einen Pod gesendet, sondern zunächst an den LoadBalancer. Dieser weiß durch die konfigurierten Selektoren mit welchen Pods er verbunden ist und kann die Anfragen dementsprechend weiterleiten. Fällt eine Replika aus und wird anschließend neu erstellt, registriert der LoadBalancer dies automatisch und leitet die ankommenden Anfragen an die korrekte Adresse weiter.

Der Unterschied zwischen einem Service und dem verwendeten LoadBalancer ist die Möglichkeit zur Kommunikation nach außen. So ist die Kommunikation mit einem LoadBalancer auch durch externe Komponenten möglich, selbst wenn sich diese nicht innerhalb des Clusters befinden. Weiterhin wird mit diesem Ansatz die Lastverteilung für verfügbare Pods ermöglicht. Dafür werden die verschiedenen Replikas berücksichtigt und ankommende Anfragen werden an die Instanz mit der niedrigsten Auslastung weitergeleitet.

Namespace

Das hier vorgestellte Konzept wird für die Verwaltung des Clusters genutzt und nennt sich Namespace. Mit Namespaces können Ressourcen gruppiert und voneinander isoliert werden. Im Kontext des Deployment-Services werden Namespaces beispielsweise für die Trennung von Datenpipelines genutzt. Da ein Modell immer einen Namen besitzt und nur deployt werden kann, wenn der Name bislang nicht vergeben wurde, lassen sich so die Pods der verschiedenen Anwendungen voneinander trennen. Ebenfalls werden die für das Messaging benötigten Ressourcen von den deployten Datenpipelines durch die Nutzung von Namespaces isoliert.

6.3.2 Deployment-Service

In diesem Abschnitt wird die Funktionalität des Deployment-Service beschrieben. Die Hauptfunktionalität ist die Provisionierung modellierter Pipelines, welche im Folgenden erläutert wird. Implementiert wurde der Service als Typescript Anwendung unter Verwendung des Frameworks Express. Für die Kommunikation mit dem Orchestrator wurde ein Javascript Client für Kubernetes[13] verwendet. Darüber hinaus ist die Kommunikation mit Docker Hub mittels Docker Hub API[14] umgesetzt worden.

Die Funktionalität des Deployment-Service kann in fünf Bestandteile aufgeteilt werden, die in den folgenden Unterabschnitten vertieft werden. Zunächst wird die Vorbereitung der Deploymentumgebung betrachtet. Anschließend werden die vier implementierten Schnittstellen vorgestellt und ihre Funktionalität beleuchtet. Hierzu zählen das Deployment und Undeployment von Datenpipelines, das Bereitstellen verfügbarer Docker Images, sowie die Bereitstellung bereits deployter Modelle.

Vorbereitung der Deploymentumgebung

Vor dem Betrieb des Deployment-Service ist es notwendig zwei vorbereitende Schritte zu treffen, um die Provisionierung einer funktionierenden Datenpipeline gewährleisten zu können. Der erste Schritt bezieht sich auf die Initialisierung der Deploymentumgebung. Hierfür ist es notwendig, ein Kubernetes Cluster auf einem lokalen Rechner zu erstellen. Wie bereits erwähnt, wird für das Aufsetzen eines lokalen Clusters Minikube genutzt. Durch das Starten von Minikube wird ein initialer Knoten als Docker Container deployt, welcher ein Kubernetes Cluster darstellt. Dieser Schritt ist für den Deployment-Service notwendig, da die Kommunikation mit dem Cluster gewährleistet sein muss, um Ressourcen erstellen und verwalten zu können.

Der zweite vorbereitende Schritt ist das Aufsetzen der Umgebung für die Kommunikation. Die Logik zur Vorbereitung wurde mittels verschiedener Funktionen innerhalb des Deployment-Services umgesetzt. Um Ressourcen im Kubernetes Cluster zu gruppieren und somit voneinander zu isolieren, wurde eine Funktion zur Erstellung von Namespaces implementiert. Hierfür wird eine Verbindung mit dem Cluster über den Javascript Client hergestellt. Mithilfe dessen kann der Zustand des Clusters durch das Senden von REST-Anfragen abgefragt und modifiziert werden. Die notwendigen Komponenten für die Kommunikation werden durch die Nutzung eines Namespaces so von den restlichen Ressourcen getrennt.

Ist die Erstellung des Namespace abgeschlossen, wird Kafka innerhalb des Clusters aufgesetzt. Kafka ist eine frei zugängliche Ereignis Streaming-Plattform, die eine Publish-Subscribe-Kommunikation zwischen den Bestandteilen der deployten Datenpipelines ermöglicht. Für das Aufsetzen von Kafka wird Strimzi[15] genutzt. Strimzi ermöglicht ein effizientes Aufsetzen von Kafka innerhalb eines Kubernetes Clusters und stellt hierfür vorgefertigte Konfigurationsdateien bereit. Beim Deployment der bereitgestellten Konfigurationen werden Strimzi-spezifische Ressourcen, Kafka Broker, sowie

[13] Javascript Client für Kubernetes: https://www.npmjs.com/package/@kubernetes/client-node
[14] Docker Hub API: https://www.npmjs.com/package/docker-hub-api
[15] Strimzi: https://strimzi.io/

Zookeeper[16]-Instanzen innerhalb des gewählten Namespace erstellt. Zookeeper stellt eine Software dar, welche für das Betreiben von Kafka notwendig ist. Kafka nutzt diese für das Speichern und Verwalten von Metadaten.

Nachdem diese zwei vorbereitenden Schritte abgeschlossen sind, ist das Kubernetes-Cluster bereit für das Deployment von Datenpipelines. Aus diesem Grund wurde eine Funktion implementiert, welche den Zustand der zugehörigen Ressourcen innerhalb des Kuberentes Clusters überprüft. Dies geschieht über die Abfrage des Zustands der verschiedenen Pods, welche in den vorbereitenden Schritten erstellt werden. Die Funktion wird genutzt, um zu erkennen, wann die Vorbereitung abgeschlossen ist und Deployments durchgeführt werden können. Dadurch ist es außerdem möglich den Deployment-Service zu unterbrechen oder neu zu starten, ohne die initialen Vorbereitungsschritte erneut durchführen zu müssen. Sind die notwendigen Ressourcen einmal deployt, bleiben diese so lange bestehen und aktiv, bis das Kubernetes Cluster oder die entsprechende Ressource gestoppt wird. Dies geschieht unabhängig vom Lebenszyklus des Deployment-Services.

Deployment einer Datenpipeline

Um eine modellierte Datenpipeline zu deployen, wurde ein Endpunkt für das Empfangen von Modellen erstellt. Dieser wird vom Modellierungswerkzeug aufgerufen, wenn der *Deployment*-Knopf gedrückt wird. Über eine POST-Anfrage wird das Modell im JSON-Format versendet und beim Empfangen in eine interne Repräsentation konvertiert. Anschließend wird das Modell dem ModelManager, einer Klasse zur Verarbeitung des empfangenen Modells, übergeben.

Innerhalb des ModelManagers werden verschiedene Funktionen ausgeführt, um aus den empfangenen Daten neue Informationen zu extrahieren. Das Ziel des ModelManagers ist es, das empfangene Datenmodell in eine neue Struktur zu überführen. Dies ist notwendig, da das empfangene Modell keine direkte Möglichkeit besitzt, die Nachbarn eines Knotens ausfindig zu machen. Ein Grund dafür ist, dass jeder Datenquelle, Datensenke und jedem Filter eine ID zugewiesen wird. Die Verbindungen zwischen den Knoten werden als Tupel von zwei IDs festgehalten. Somit wurde eine Funktion entwickelt, welche die Nachbarn eines Knotens ausgibt.

In Abschnitt 6.2 wurde bereits erklärt, dass Datensenken und Filter gleich modelliert werden und sich von den Datenquellen unterscheiden. Im ModelManager wurde daher eine Funktionalität entwickelt, welche Datensenken von Filtern trennt. Dies geschieht durch die Analyse der eingehenden und ausgehenden Verbindungen eines Knotens. Mit dem Wissen, dass eine Datensenke nur eingehende Verbindungen hat und ein Filter sowohl ein-, als auch ausgehende Kanten besitzt, können entsprechende Objekte für jede Komponente des Modells erstellt werden.

Nach der Vorbereitung des empfangenen Modells werden die erweiterten Informationen an eine neue Funktion übergeben, welche das Deployment steuert. Das Deployment findet Knoten für Knoten statt und wird rekursiv abgearbeitet. Der Prozess beginnt bei den Datensenken und arbeitet sich über die Kinder der Datensenken fort, bis die Datenquellen erreicht werden. Für jeden Knoten wird dabei eine neue Deploymentkonfiguration erstellt. Eine Deploymentkonfiguration ist eine

[16]Zookeeper: https://zookeeper.apache.org/

Klasse, welche sich aus verschiedenen Kubernetes Konzepten zusammensetzt. In der Instanz einer solchen Klasse werden die notwendigen Informationen für die Erstellung von folgenden Kubernetes Konzepten festgehalten:

- Namespace
- ConfigMap
- Service
- Deployment
- Horizontal Pod Autoscaler

Durch das vom Modellierungswerkzeug empfangene Modell lässt sich der Modellname für die Erstellung eines Namespaces nutzen. Dieser wird zusätzlich bei der Deploymentkonfiguration eines Knotens angegeben, damit dieser im korrekten Namespace deployt wird. ConfigMaps werden genutzt, um Umgebungsvariablen an den Service zu senden. Hierzu zählt der Name des Knotens, sowie die notwendigen Informationen für die Kommunikation mit seinen Nachbarknoten. Der Service, umgesetzt als LoadBalancer, wird mit Selektoren ausgestattet, die im korrekten Namespace auf den Namen des Knotens registriert werden. Dadurch können Anfragen korrekt an die Replicas zugewiesen werden. Darüber hinaus wird eine Portkonfiguration übergeben, welche mit der des zuvor erstellten LoadBalancers übereinstimmt. Zusätzlich wird der Name der passenden ConfigMap an das Deployment übergeben, sodass die erstellten Pods auf die Umgebungsvariablen zugreifen können. Der letzte essenzielle Punkt ist die Konfiguration des richtigen Images für einen Knoten, sodass der Container innerhalb des Pods die richtige Anwendung bereitstellt. Pro Deploymentkonfiguration werden außerdem ein ReplicaSet, sowie ein oder mehrere Pods erstellt.

Wie zuvor erwähnt, werden diese Informationen für jede Datensenke und anschließend rekursiv für ihre Kinder angewandt. Der Horizontal Pod Autoscaler stellt eine Außnahme dar, da er lediglich für Filter und Datensenken erstellt wird. Für Datenquellen wird dieser nicht benötigt, da eine Skalierung dieser nicht als notwendig angesehen wird. Im Gegensatz zu Datenquellen benötigen Filter und Datensenken die Möglichkeit zur Skalierung, da die Anzahl der eingehenden Nachrichten abhängig von der Zahl der angeschlossenen Datenquellen ist. Der beschriebene Deployment-Prozess wird bis zum letzten Filter in der Hierarchie durchgeführt. Zuletzt wird in einem extra Schritt über alle Datenquellen iteriert, um diese gesondert zu deployen. Der Grund für diese Strategie ist, dass eine Datenquelle mehrere Elternknoten haben kann und es bei einem Top-Down Vorgehen zu Dopplungen der deployten Datenquellen kommen könnte. Da mittels empfangenem Datenmodell jedoch alle Nachbarknoten für eine Instanz hinterlegt sind, stellt dieser Sonderfall keinen großen Mehraufwand dar. Ein weiterer Unterschied beim Deployment von Datenquellen ist die Möglichkeit des Multi-Deployments. Bei einem Multi-Deployment wird eine Datenquelle nur einmalig modelliert, durch die Spezifikation mehrerer Adressen sollen jedoch mehrere Datenquellen gleichzeitig erstellt werden. Für diesen Fall wird über das Adressfeld einer Datenquelle iteriert. So können pro Knoten beliebig viele Deploymentkonfigurationen generiert und Instanzen provisioniert werden.

Undeployment

Der zweite Endpunkt, den der Deployment-Service bereitstellt, bietet die Möglichkeit ein Undeployment auszulösen. Hierfür erhält der Service eine DELETE-Anfrage, welche den Namen des zu löschenden Modells im Body der Nachricht beinhaltet. Durch den Aufbau der Kubernetes Konzepte konnte die Funktionalität für das Löschen mit wenigen Zeilen Quellcode implementiert werden. Für diese Operation ist es ausreichend, den entsprechenden Namespace der deployten Datenpipeline zu löschen. Dies kann über die verwendete Kubernetes Schnittstelle gelöst werden.

Bevor der Namespace aus Kubernetes entfernt werden kann, müssen zunächst alle Ressourcen beseitigt werden, die ihm zugeordnet sind. Durch die deklarative Syntax von Kubernetes wird dies automatisch erledigt. Mit einem Polling-Mechanismus überprüft der Deployment-Service anschließend, wann der Namespace entfernt wurde. Nach der erfolgreichen Entfernung wird eine Antwort an den anfragenden Service mit einem Statuscode 200 geschickt, um zu signalisieren, dass das Undeployment erfolgreich war.

Bereitstellen von Docker Images

Um die Auswahl eines Docker Images im Modellierungswerkzeug zu ermöglichen, wurde ein Endpunkt für das Bereitstellen der notwendigen Informationen im Deployment-Service eingebaut. Hierfür wurde die Docker Hub API genutzt, welche es ermöglicht mittels HTTP-Anfragen eine Liste von Images abzurufen. Da es sich bei den verfügbaren Docker Images um selbst erstellte Images handelt, wurde hierfür ein eigens erstelltes und öffentliches Konto bei Docker Hub genutzt. Der Nutzername ist daher als Konstante im Quellcode hinterlegt, wodurch auf den richtigen Account zugegriffen werden kann. Mittels API und Nutzername ist es möglich eine Liste von erstellten Repositories abzufragen. Um die Namen der verschiedenen Images zu erhalten wird über die Repositories iteriert. In jeder Iteration werden die verfügbaren *Tags* eines Repositories abgefragt. Mit dieser Liste lassen sich die Namen der Images extrahieren. Um die Anzahl überschaubar zu halten, wird nur die aktuellste Version eines Images gespeichert. Anschließend werden die Image-Namen an den aufrufenden Service als Antwort übergeben und können so beispielsweise im Modellierungswerkzeug zur Anzeige genutzt werden.

Bereitstellen deployter Modelle

Um deployte Modelle im Modellierungswerkzeug anzeigen zu können, wurde ein weiterer Endpunkt entwickelt, welcher der Abfrage von aktiven beziehungsweise bereits deployten Modelle dient. Analog zur Funktionalität des Undeployments, werden aktive Modelle ebenfalls anhand des zugehörigen Namespaces identifiziert. Hierfür wurde eine Funktion implementiert, welche über den Kubernetes Client das Cluster nach vorhandenen Namespaces abfragt und mit einer Liste antwortet, welche den Namen der verfügbaren Datenpipelines entspricht. Vor der Rückgabe wurde ein zusätzlicher Filter verbaut. Dieser sorgt dafür, dass nur Namespaces die zu Datenpipelines gehören, zurückgegeben werden. Durch die Konfiguration einer Liste von Namespaces die ausgeschlossen werden sollen, kann so vermieden werden, dass Namespaces mit anderer Logik vom Endnutzer

beeinflusst werden können. Dazu zählt beispielsweise der „default“ Namespace, welcher Kubernetes-spezifische Komponenten beinhaltet. Ebenso soll der Namespace zur Verwaltung von Messaging-spezifischen Komponenten nicht durch den Nutzer beeinflusst werden können, weshalb er ebenfalls ausgeschlossen wurde.

6.4 Ausführung

Dieser Abschnitt behandelt verschiedene Aspekte, die zur Laufzeit des Deployment-Service und deployter Datenpipelines stattfinden. Im ersten Unterabschnitt wird hierfür die Nutzung von Messaging anhand einer deployten Pipeline erklärt. Anschließend wird erläutert wie Monitoring, die Möglichkeit zur Überwachung des Kubernetes Clusters während der Laufzeit, implementiert wurde. Zuletzt wird darauf eingegangen wie das in Abschnitt 6.1.1 vorgestellte Anwendungsszenario anhand von Docker Images umgesetzt wurde.

6.4.1 Messaging

Für die Kommunikation zwischen den einzelnen Services innerhalb einer Datenpipeline wurde Messaging als Architekturmuster ausgewählt. Als Technologie für die Umsetzung wurde Apache Kafka gewählt. Im folgenden Unterabschnitt wird diese vorgestellt und es wird begründet, warum der Einsatz dieser Technologie für das Anwendungsszenario geeignet ist. Anschließend wird im darauffolgenden Unterabschnitt erläutert, wie die einzelnen Komponenten einer Datenpipeline miteinander kommunizieren und wie die Kommunikation beim Deployment gewährleistet wird.

Apache Kafka

Das vorgestellte Anwendungsszenario und viele vergleichbare Alternativen profitieren von einem Publish-Subscribe-basierten Messaging Muster. Publish-Subscribe-Ansätze werden häufig in IoT-Anwendungen zur Kommunikation genutzt. Sie ermöglichen es Produzenten und Konsumenten in verteilten Systemen voneinander zu entkoppeln. Produzenten senden Nachrichten mit dem Publish-Subscribe-Ansatz nicht direkt an spezifische Empfänger. Stattdessen werden Nachrichten durch Topics kategorisiert und an die Messaging Middleware gesendet. Empfänger können Topics abonnieren und erhalten nach einem Abonnement die entsprechend kategorisierten Nachrichten. Mit diesem Ansatz kann demnach eine höhere Skalierbarkeit und dynamischere Netzwerktopologie erreicht werden.

Für die Umsetzung der Publish-Subscribe-Kommunikation wurde Apache Kafka als Technologie gewählt. Kafka ist ein verteiltes Publish-Subscribe Messaging System, welches erstellt wurde, um persistentes Messaging mit hohem Durchsatz zu ermöglichen [Cur16]. Diese Technologie wurde bereits erfolgreich von verschiedenen Anwendungen im Bereich der Software-Defined-Car-Anwendungen genutzt. Du et al. [DCR+18] stellen in Ihrer Arbeit eine Strategie für ein verteiltes Nachrichtenvermittlungssystem vor, welches auf Kafka basiert und für die Übermittlung von VF-Daten entwickelt wurde. Amini et al. [AGP17] nutzen Kafka für eine Big-Data-Analytics

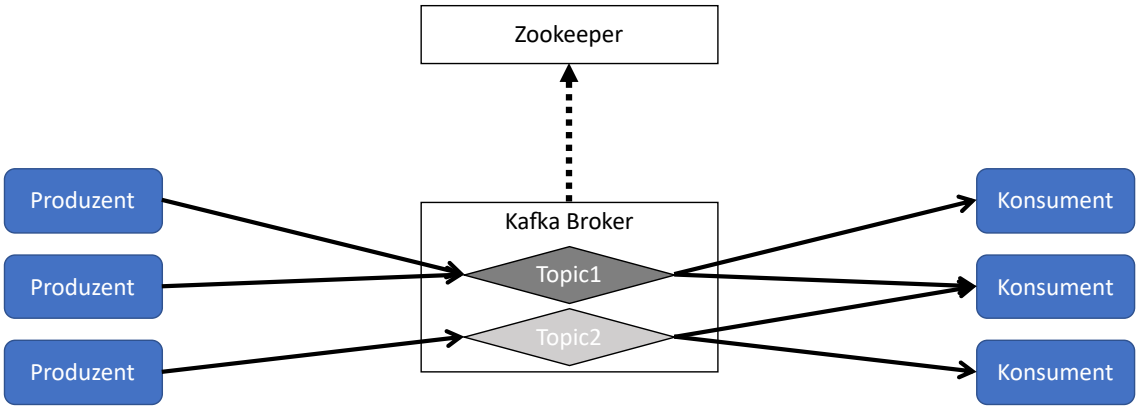


Abbildung 6.12: Darstellung der Kommunikation zwischen Konsumenten und Produzenten mit Hinblick auf notwendige Bestandteile für Kafka.

Architektur für die Verkehrssteuerung in Echtzeit. Dabei betonen die Autoren die Eignung von Kafka als letzten Stand der Technik für die Kommunikation in Datenpipelines und die Stream-Verarbeitung. Weiterhin stellt Curry [Cur16] die Vorteile der Nutzung von Kafka im Bereich von Big Data dar.

Um die Kommunikation mittels Kafka zwischen Services zu ermöglichen werden sogenannte Broker verwendet. Services die mit Kafka kommunizieren können als Produzenten oder Konsumenten eingestuft werden, abhängig davon ob sie Informationen an einen Broker senden, oder Informationen von einem Broker empfangen. Broker nutzen Topics um Nachrichten zu gruppieren. Produzenten senden Nachrichten an ein oder mehrere Topics. Konsumenten können Topics abonnieren und erhalten so die zugehörigen Nachrichten. Der Broker übernimmt die Verwaltung der Topic-Erstellung und Nachrichtenvermittlung. So wird dafür gesorgt, dass Nachrichten den korrekten Topics zugeordnet werden, wenn diese an einen Broker gesendet werden und ein Konsument nur Nachrichten erhält, die zu einem abonnierten Topic gehören.

Abbildung 6.12 zeigt dieses Verhalten anhand eines Beispiels mit drei Produzenten, einem Broker und drei Konsumenten. Anhand der Darstellung wird verdeutlicht, dass Produzenten eine Nachricht an mehrere Topics senden, und Konsumenten Nachrichten verschiedener Topics abonnieren können. Zusätzlich ist eine Zookeeper-Instanz abgebildet. Der Kafka Broker nutzt Zookeeper zur Verwaltung des Zustands. Broker können mit dieser Instanz kommunizieren, um den Zustand des Clusters zu speichern und abzufragen. Dazu zählen Informationen über Broker, Topics und Nutzer.

Kommunikation deployter Datenpipelines

Dieser Abschnitt erklärt, wie die Kommunikation zwischen den einzelnen Bestandteilen einer deployten Datenpipeline gehandhabt wird. Eine Datenpipeline besteht aus verschiedenen Komponenten, welche im Kontext von Kafka Produzenten, Konsumenten oder beides sein können. Datenquellen stellen dabei immer Produzenten dar, wohingegen Datensenken Konsumenten sind. Ein Filter ist sowohl Konsument als auch Produzent, da dieser Daten empfängt, verarbeitet und anschließend an den nächsten Schritt der Datenpipeline sendet.

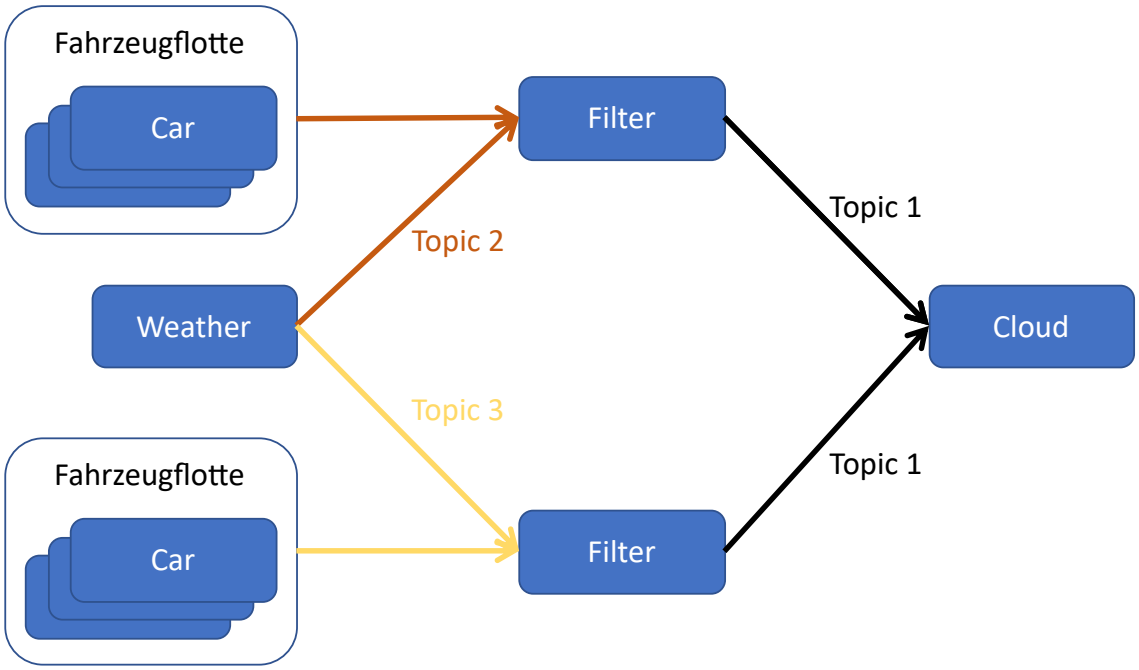


Abbildung 6.13: Topic-Vergabe beim Erstellen einer Datenpipeline anhand des Anwendungsszenarios aus Abschnitt 6.1.1.

Um die einzelnen Schritte der Datenpipeline korrekt miteinander zu verknüpfen, wurden Kafka Topics genutzt. Wie in Abschnitt 6.3.2 bereits beschrieben, wird beim Deployment eine Top-Down Strategie angewandt um Komponenten zu deployen. Während des Deployments werden in dieser Reihenfolge ebenfalls die entsprechenden Kafka Topics erstellt. Abbildung 6.13 zeigt die verschiedenen Kafka Topics die beim Deployment des Anwendungsszenarios auftreten. Zunächst werden die Datensenken deployt, bei jeder Iteration wird dabei ein neues Topic erstellt. Somit stellt jedes Topic die Möglichkeit zur Kommunikation mit einem Knoten dar. Ebenso wird der Verbindungsaufbau für Filter gestaltet. Pro Filter wird also ein neues Topic erstellt.

Beim Deployment eines Filters ist es jedoch notwendig zusätzliche Informationen für eine funktionierende Kommunikation zu übertragen. Da dieser auch einen Produzenten darstellt, im Unterschied zu einer Datensenke, müssen ihm die Topic-Namen übergeben werden, an welche er seine Nachrichten senden soll. Im Falle des Anwendungsszenarios erhält ein Filter somit eine Umgebungsvariable mit zwei Einträgen, den eingehenden und ausgehenden Topics. So kann innerhalb der Filter-Logik konfiguriert werden welche Topics abonniert, und an welche Topics gesendet werden soll. Im Gegensatz zu Filtern und Datensenken benötigen Datenquellen keine eingehenden Topics, da sie keine Daten empfangen. Aus diesem Grund wird bei der Konfiguration einer Datenquelle nur die Information über ausgehende Topics benötigt. Sie stellen jedoch eine Ausnahme dar, da eine Datenquelle Informationen an mehr als eine Komponente senden kann. Daher kann es vorkommen, dass der Komponente eine Liste von Topics übergeben werden muss. Diese Logik wird ebenfalls durch den Deployment-Service gehandhabt, indem bei den Datenquellen auf die Anzahl an ausgehenden Verbindungen geachtet wird.

6.4.2 Monitoring

Eine weitere Funktionalität des Deployment-Service ist die Überwachung des Kubernetes-Clusters. Durch die Kommunikation mit Kubernetes ist es über die verwendete API möglich, Veränderungen des Zustands einzelner Ressourcen zu überwachen und auf diese zu reagieren. In diesem Abschnitt werden die verschiedenen Funktionalitäten erklärt, welche es einem Nutzer ermöglichen die Zustandsänderung innerhalb des Clusters nachzuvollziehen.

Die Ereignisse innerhalb des Kubernetes-Clusters werden durch sogenannte Watches beobachtet, welche von der Kubernetes API zur Verfügung gestellt werden. Watches wurden für die Überwachung von Namespaces und Pods intialisiert und ermöglichen das Aufrufen von Rückruffunktionen, wenn gewisse Ereignisse ausgelöst werden. Zu diesen Ereignissen zählt das Erstellen, Modifizieren oder Löschen einer Ressource. Findet eines dieser Ereignisse statt, wird mit der hinterlegten Rückruffunktion reagiert. Das Monitoring wird hauptsächlich in vier Anwendungsfällen eingesetzt. Diese werden in den folgenden Unterabschnitten beschrieben und es wird erklärt, wie und wofür das Monitoring dabei genutzt wird.

Initiales Setup

Beim initialen Setup wird das Monitoring genutzt, um ein gelungenes Setup sicherstellen zu können. Dabei werden Watches für Pods genutzt, um festzustellen, wann Pods deployt und funktionsfähig sind. Das initiale Aufsetzen beinhaltet die Provisionierung von Komponenten, die zur Kommunikation von Datenpipelines dienen. Dabei handelt es sich um eine fest definierbare Liste an Pods. Durch den Abgleich dieser Liste mit den Rückgabewerten der Pod-Watches kann bestimmt werden, wann die Kommunikationsumgebung voll funktionsfähig ist. Nach einem erfolgreichen Setup wird dem Nutzer auf der Konsole eine Nachricht ausgegeben, die dies signalisiert.

Deployment einer Datenpipeline

Beim Deployment von Datenpipelines besteht aufgrund der hohen Anzahl an Ressourcen die Möglichkeit, dass Fehler auftreten. Daher wurde bei diesem Anwendungsfall das Monitoring genutzt, um dem Nutzer den Fortschritt eines Deployments über die Konsole offenzulegen. Analog zum initialen Setup werden Pod-Watches genutzt, um dem Nutzer neu erstellte Pods zu signalisieren, wenn diese betriebsbereit sind. Hierfür werden Informationen zum Pod-Namen, sowie zum zugehörigen Namespace ausgegeben. Zusätzlich wurden Namespace-Watches eingeführt, um den Gesamtzustand der Datenpipeline überwachen zu können. Durch die Kombination der Zustandsinformationen von Pods und Namespaces kann dem Nutzer angezeigt werden, wann das Deployment erfolgreich beendet wurde. Darüberhinaus wird die gleiche Herangehensweise genutzt, um dem Modellierungswerkzeug mitzuteilen, dass die Anfrage zur Erstellung der Datenpipeline erfolgreich ausgeführt wurde.

Auftreten von Fehlern

Ein essenzieller Teil des Monitorings bezieht sich auf das Überwachen von Fehlern. Im Deployment-Service wurde das Monitoring für zwei Fehlerarten implementiert, das Terminieren von Pods und das Terminieren von Containern. Beim Auftreten beider Fehlerarten wird der Nutzer über die

Konsole benachrichtigt. Dabei wird der Nutzer darauf hingewiesen, ob der gesamte Pod, oder nur ein Container innerhalb eines Pods unerwartet terminiert ist. Durch die deklarativen Anweisungen bei der Konfiguration eines Deployments, werden die entsprechenden Pods oder Container automatisch durch Kubernetes neu gestartet. Dadurch können Fehler häufig ohne Aufwand behoben werden. Ist dies nicht der Fall, hilft die Ausgabe der Pod- und Container-Informationen dem Nutzer schneller den Fehler zu finden.

Undeployment

Analog zum Deployment wird Monitoring auch beim Undeployment einer Datenpipeline genutzt. Hierfür werden sowohl Pod- als auch Namespace-Watches genutzt, um sicherzustellen, dass eine Datenpipeline erfolgreich gelöscht wurde. Für beide Ressourcen wird dem Nutzer eine Nachricht ausgegeben, welche den Namen der Ressource beinhaltet. Dies ermöglicht es nachzuvollziehen, in welchem Zustand sich das Undeployment befindet. Außerdem hilft es dabei Fehler besser zuordnen zu können, indem transparent signalisiert wird, in welchem Prozess sich der Deployment-Service gerade befindet. Das Monitoring gewährleistet in diesem Fall auch das rechtzeitige Senden einer Antwort an das Modellierungswerkzeug. Durch die Nutzung von Namespace-Watches kann garantiert werden, dass der Namespace und alle zugehörigen Ressourcen entfernt wurden, bevor eine Antwort für das erfolgreiche Abschließen der Anfrage gesendet wird.

6.4.3 Bestandteile der Datenpipeline

In diesem Abschnitt werden die entwickelten Services vorgestellt, welche das Anwendungsszenario aus Abschnitt 6.1.1 widerspiegeln. Um dieses umzusetzen, wurde in Abschnitt 6.2 bereits ein Modell des Anwendungsszenarios, welches mit dem Modellierungswerkzeug erstellt wurde, dargestellt. Im Folgenden geht es um die erstellten Docker Images, welche für die Datenquellen, Filter und Datensenken des Modells genutzt wurden. Alle Services wurden dabei mittels Typescript und mithilfe des Express-Frameworks erstellt. Weiterhin benutzt jeder Service die Kafka-node API[17], um Nachrichten mittels Kafka versenden oder empfangen zu können. Zu jedem Service wurde ein Docker Image erstellt und anschließend in einem DockerHub Repository hochgeladen, was das Deployment innerhalb eines Kubernetes Clusters ermöglicht.

Car-Service

Der Car-Service ist ein Service, welcher zur Simulation eines Software-Defined-Cars entwickelt wurde und als Datenquelle der Pipeline dient. Wird der Service gestartet, verbindet sich dieser zunächst über den Kafka-node Client mit der Kafka Schnittstelle. Diese stellt Funktionen bereit, welche es ermöglicht Nachrichten an den Broker zu senden. Über Umgebungsvariablen werden die IP-Adresse und der Port des Kafka Brokers an den Service übermittelt. Diese Informationen sind notwendig, um eine Verbindung mit dem Broker zu ermöglichen.

[17] Kafka-node API: https://www.npmjs.com/package/kafka-node

Listing 6.3 Ausgabe simulierter Fahrzeugdaten innerhalb des Car-Service in JSON-Repräsentation.

```
{
  vin: 'UCS2G4NXAU0CS1SS1',
  ip: '170.227.249.146',
  modelInformation: { manufacturer: 'porsche', model: '911' },
  geoPosition: { lat: 48.834881, long: 9.152181 },
  fuel: 'petrol',
  currentConsumption: { value: 14.9, unit: 'l/100km' },
  numberOfPassengers: 1,
  tirePressure: 2.4,
  mileage: { value: 91981, unit: 'km' },
  power: { value: 173, unit: 'horsepower' },
  currentSpeedLimit: { value: 80, unit: 'km/h' },
  safetyDistance: { value: 71, unit: 'm' }
}
```

Anschließend wird die Klasse CarDataGenerator initialisiert. Diese generiert initial ein JSON-Objekt, welches die simulierten Daten enthält. Listing 6.3 zeigt die Ausgabe der Generierungsfunktion. Das resultierende Objekt enthält diverse Felder. Dazu zählen Informationen zur Identifizierung eines Fahrzeugs wie die Fahrzeugidentifikationsnummer, welche als „vin“ gespeichert wird. Zu identifizierenden Attributen zählen außerdem die IP und Modellinformationen. Weiterhin gibt es Felder, welche Informationen über den aktuellen Zustand des Fahrzeugs beinhalten wie die Geo-Koordinaten und der momentane Kraftstoffverbrauch. Zusätzlich wurden Informationen eingebaut, die zur Ermittlung von Emissionsdaten nicht zwingend notwendig sind, wie der Sicherheitsabstand zu anderen Verkehrsteilnehmern. Für die Generierung dieser Werte wurden Funktionen implementiert, die möglichst realistische Werte erzeugen. So werden beispielsweise nur Modelle des entsprechenden Herstellers generiert, bzw. das Format einer Fahrzeugidentifikationsnummer korrekt eingehalten.

Nach der Generierung dieser Werte wird eine Kafka Nachricht erstellt, welche das generierte Objekt im Body beinhaltet. Außerdem werden ein Nachrichtentyp, die momentane Uhrzeit und der Name des sendenden Knotens im Body hinzugefügt. Über die Umgebungsvariablen können anschließend die hinterlegten Kafka-Topics genutzt werden, um die Nachricht zu senden. Für eine realistischere Simulation wartet der Service für eine definierte Zeit und verändert anschließend die generierten Daten. Identifizierende Merkmale wie die Identifikationsnummer und Modellinformationen bleiben bestehen, variable Informationen wie die Geschwindigkeit, Geo-Koordinaten und Verbrauch werden neu berechnet. Bei den Veränderungen wurde darauf geachtet, realistische Wertänderungen zu simulieren, indem Intervalle für mögliche Werteveränderungen definiert wurden.

Weather-Service

Der Weather-Service ist ein weiterer Service, der eine Datenquelle darstellt und einen Wetterdienst simulieren soll. Verhalten und Aufbau des Service sind sehr ähnlich gestaltet wie beim Car-Service, weshalb dieser Unterabschnitt hauptsächlich auf die Unterschiede der beiden Services eingeht. Der größte Unterschied ist dabei das generierte Datenmodell, welches in Listing 6.4 dargestellt ist. Die generierten Daten bestehen aus vier Feldern, welche die Temperatur, den Zustand des Wetters,

Listing 6.4 Ausgabe simulierter Wetterdaten innerhalb des Weather-Service in JSON-Repräsentation.

```
{
  temperature: { unit: 'celsius', value: 27.6 },
  weather: 'windy',
  wind: { direction: 'E', speed: 10.1 },
  humidity: 65.96
}
```

Informationen zum Wind und zur Luftfeuchtigkeit beinhalten. Analog zum Car-Service verändern sich diese Informationen über die Zeit und senden Kafka Nachrichten an die zutreffenden Topics, welche durch Umgebungsvariablen definiert sind.

Filter-Service

Der Filter-Service stellt einen Filter in der Datenpipeline dar, welcher dafür gedacht ist auf einer RSU deployt zu werden. Er wurde so implementiert, dass er Daten vom Car- und Weather-Service empfangen kann, diese verarbeitet und die verarbeiteten Daten weiterschickt. Um diese Aufgabe zu erfüllen, ist der Filter-Service sowohl ein Konsument von Kafka Nachrichten, als auch ein Produzent. Aus diesem Grund übergibt ihm der Deployment-Service beim Deployment die notwendigen Daten für den Verbindungsaufbau. Hierfür benötigt der Filter-Service die Kafka-Topics für das Empfangen und Versenden von Nachrichten, sowie die IP und den Port des Kafka Brokers.

Zunächst wird der Filter-Service initialisiert, indem zwei Objekte erstellt werden, der Kafka-Consumer und KafkaProducer. Beide verbinden sich mit dem Broker, woraufhin der Service funktionsfähig ist. Für die Weiterleitung von Nachrichten wurde ein Listener verwendet, der durch die Kafka API zur Verfügung gestellt wird. Dieser wird bei ankommenden Nachrichten ausgelöst, woraufhin diese verarbeitet und im Anschluss weitergesendet werden. Ankommende Nachrichten beinhalten im Body immer ein JSON-Objekt, welches vom Weather- oder Car-Service gesendet wird. Für die Unterscheidung der ankommenden Nachrichten wird der Message-Body inspiziert und anhand des hinterlegten Typen werden die Daten entweder an den CarDataHandler oder den MessageDataHandler übergeben. Beide enthalten Logik zum Filtern der übergebenen Daten. Dabei werden Felder, welche für die Berechnung der Emissionsdaten überflüssig sind, entfernt.

Im Fall einer Weather-Service-Nachricht besteht das resultierende Objekt anschließend nur noch aus Temperatur- und Wind-Informationen. Bei Daten die vom Car-Service empfangen werden, entfallen die Felder für Geo-Koordinaten, Reifendruck, Leistung, Sicherheitsabstand und die geltende Geschwindigkeitsbeschränkung, da diese zur weiteren Verarbeitung nicht benötigt werden. Ist die Filterung abgeschlossen, wird eine neue Nachricht an den Topic des Cloud-Services gesendet.

Cloud-Service

Der Cloud-Service ist eine Datensenke und bildet damit das Ende des Datenflusses innerhalb der Pipeline. Aus diesem Grund agiert der Service nur als Konsument von Kafka Nachrichten. Um die zugehörigen Nachrichten zu empfangen, wird ihm der Topic-Name vom Deployment-Service über Umgebungsvariablen übermittelt.

Beim Starten des Services wird zunächst die Verbindung zu den benachbarten Filter-Knoten über ein Abonnement des richtigen Topics hergestellt. Anschließend ist der Cloud-Service bereit die verarbeiteten Daten der verschiedenen Filter zu empfangen. Bei der Ankunft einer Nachricht wird vorab überprüft, von welchem Filter diese stammt. Anschließend wird ein neues Objekt pro Filter initialisiert. Daten von zukünftigen Nachrichten werden einem bereits bestehenden Filter zugeordnet, falls dieser existiert.

Für die Berechnung der Emissionswerte wurden zwei verschiedene Verfahren implementiert. Zum einen hat jeder Filter die Möglichkeit Durchschnittswerte auszugeben. Durch eine Konstante im Quellcode kann hinterlegt werden, wie oft dies geschehen soll. Die zweite Art der Berechnung erfolgt über die Gesamtheit aller verknüpften Filter. Auch hier kann konfiguriert werden, wie oft diese Berechnung stattfinden soll. Ist der Schwellwert erreicht, wird eine Zusammenfassung verschiedener Informationen angezeigt.

Die Ausgabe beinhaltet Informationen über die Gesamtanzahl erhaltener Nachrichten und die Zahl der VFs, deren Nachrichten am Cloud-Service angekommen sind. Zusätzlich wird der Durchschnitt des Gesamtverbrauchs aller bisherigen Werte angezeigt, ebenso wie die durchschnittliche Temperatur und Windgeschwindigkeit.

7 Evaluation

In diesem Kapitel werden die Anforderungen aus Kapitel 4 retrospektiv betrachtet. Die nachfolgenden Abschnitte widmen sich jeweils einer Anforderung und evaluieren das Konzept sowie den Prototyp basierend auf dieser. Abschließend werden die limitierenden Faktoren vorgestellt.

Skalierbarkeit

Die erste Anforderung stellt die Skalierbarkeit dar. Bei der Betrachtung dieses Aspekts müssen verschiedene Gesichtspunkte analysiert werden. Hierzu zählt die Skalierbarkeit einzelner Komponenten. Diese wurde durch die Nutzung des Horizontal Pod Autoscalers umgesetzt. Mit dieser Kubernetes-Ressource werden die deployten Pods automatisch horizontal skaliert, wenn der angegebene Schwellenwert für die CPU-Auslastung überschritten wird [BSM20].

Darüber hinaus wurden LoadBalancer als Services genutzt, um die Kommunikation zu den Replikas eines Pods zu gewährleisten. Mit diesem Konzept ermöglicht Kubernetes eine automatische Lastverteilung [TATS18]. Hierfür überwacht der Orchestrator die angebundenen Pods eines LoadBalancers und verteilt ankommende Anfragen an die Instanz mit der niedrigsten Auslastung.

Durch die Nutzung von Kafka besteht auch die Möglichkeit die Kommunikation innerhalb des deployten Clusters skalierbar zu gestalten. Kafka ist in der Lage tausende von Nachrichten pro Sekunde zu handhaben und wird erfolgreich bei großen Unternehmen wie beispielsweise LinkedIn[1] als skalierbare Lösung für die Anwendung in Echtzeit genutzt [KNR+11]. Im Prototyp besteht die Möglichkeit über eine Konfigurationsdatei die Anzahl an Brokern festzulegen, womit der Durchsatz manuell erhöht werden kann.

Somit kann die erste Anforderung anhand dieser drei Gesichtspunkte als erfüllt betrachtet werden.

Heterogenität

Technologische Heterogenität wurde als zweite Anforderung gewählt. Durch die entworfene Architektur sind keine Einschränkungen in der Auswahl von Programmiersprachen gegeben. Jeder Baustein einer Datenpipeline ist unabhängig von den anderen implementierbar. Zusätzlich kann auch die Technologie zur Kommunikation durch eine Alternative ersetzt werden.

Neben der Architektur bietet auch die ausgewählte Deploymentstrategie eine Möglichkeit zur Erstellung technologisch heterogener Datenverarbeitungsschritte. Durch die Nutzung von Docker Images zur Provisionierung der einzelnen Bausteine wird der Entwickler nicht in der Wahl seiner Implementierungswerkzeuge eingeschränkt. Aus diesen Gründen wurde die Anforderung an technologische Heterogenität ebenfalls gewährleistet.

[1] LinkedIn: https://de.linkedin.com/

Portabilität

Gemeinsam mit der technologischen Heterogenität wurde zusätzlich die Portabilität von Anwendungen als Anforderung gestellt. Wie zuvor beschrieben, bietet die Software-Paketierung mittels Docker viele Vorteile. Durch den geringen Overhead und die eingeschränkte Größe der Images, eignen sich diese gut für die Provisionierung. Zusätzlich sind diese portabel und können von allen gängigen Betriebssystemen genutzt werden, da Docker sowohl auf Windows, macOS, wie auch Linux verfügbar ist. Durch das gewählte Konzept wird somit gewährleistet, dass Datenpipelines auf einer großen Zahl von unterschiedlichen Rechnern und Betriebssystemen betrieben werden können. Demzufolge ist der Prototyp in der Lage auf Geräten von verschiedenen Herstellern genutzt zu werden. Somit wurde die Anforderung der Portabilität ebenfalls erfüllt.

Erweiterbarkeit

Eine weitere Anforderung an das Konzept und den daraus resultierenden Prototypen stellt die Erweiterbarkeit dar. Diese konnte durch die gewählte Architektur gewährleistet werden. Der modulare Aufbau bietet die Möglichkeit bestehende Datenpipelines ohne großen Aufwand zu erweitern.

Die Erweiterbarkeit wurde ebenfalls bei der Auswahl der verwendeten Technologien berücksichtigt. Zukünftige Modifikationen der Software wurden nachfolgenden Entwicklern durch die Wahl von Standards und De-facto-Standards ermöglicht. Nach Angaben von Apache ist Kafka bei tausenden von Firmen in Nutzung, dazu zählen beispielsweise Netflix und LinkedIn [Apa22]. Auch Kubernetes zählt zu den am meisten verbreiteten Technologien zum Management von Container-basierten Applikationen. Zusätzlich wurde durch die Wahl von Typescript eine häufig genutzte Programmiersprache gewählt und durch die Dokumentation von Repository, Quellcode und Schnittstellen dafür gesorgt, die Möglichkeit zur Weiterentwicklung zu gewährleisten. Aus diesen Gründen wird ebenfalls die Anforderung der Erweiterbarkeit abgedeckt.

Robustheit/Verfügbarkeit

Die Anforderung an die Punkte Robustheit und Verfügbarkeit lässt sich in zwei Bereiche unterteilen, *Monitoring* und *Recovery*. Durch die Implementierung des Deployment-Services wurde das Monitoring umgesetzt. Anhand von Watches, bereitgestellt durch die Kubernetes API, kann der Zustand von Ressourcen innerhalb des Clusters überwacht und kontrolliert werden.

Die Recovery wird durch Kubernetes gewährleistet. Aufgrund der deklarativen Syntax zur Konfiguration des deployten Systems, ist Kubernetes der gewünschte Endzustand bekannt. Weicht der aktuelle Zustand von diesem ab, werden Ressourcen automatisch erstellt oder gelöscht, um den konfigurierten Zustand zu erreichen. So werden Container neu gestartet, nachdem diese aufgrund eines Fehlers terminiert sind. Ebenso werden abgestürzte Replicas und Pods automatisch neu gestartet, wenn diese in einen Fehlerzustand übergehen. Auch nach Absturz und erneutem Hochfahren eines gesamten Rechenknotens, ist Kubernetes teilweise dazu in der Lage, Ressourcen automatisch wiederherzustellen. Dadurch können das Monitoring und die Recovery als erfüllt angesehen werden.

Automatisierbarkeit

Die Automatisierbarkeit stellt aus verschiedenen Gründen eine essenzielle Anforderung dar. Angesichts der konzeptionierten Architektur konnte das automatisierbare Deployment von Datenpipelines ermöglicht werden. Durch die Entkopplung von Business- und Anzeigelogik wird auch Nutzern ohne Softwarekenntnisse die Bedienung der Anwendung zugänglich gemacht.

Automatisierbarkeit ist darüber hinaus stark mit den Anforderungen an Skalierbarkeit und Monitoring verknüpft. Mithilfe der Automatisierbarkeit können einzelne Ressourcen automatisch horizontal skaliert werden. Zudem findet eine automatisierte Lastverteilung des eingehenden Datenverkehrs statt. Des Weiteren kann mittels Monitoring für eine automatische Recovery gesorgt werden. Daher wird auch die letzte Anforderung, die Automatisierbarkeit, als erfüllt betrachtet.

Limitationen

Trotz der Umsetzung sämtlicher Anforderungen, beinhaltet die entwickelte Anwendung limitierende Faktoren. Die Ursache hierfür ist der zeitlich begrenzte Rahmen, der dieser Arbeit zugrunde liegt.

Die erste Limitation bezieht sich auf die Art der Datenhaltung. Der entwickelte Prototyp speichert sämtliche Informationen im Hauptspeicher. Diese Limitation ist für den Betrieb des Prototyps zwar ausreichend, skaliert jedoch nicht optimal. So kann es zu Problemen kommen, sobald die modellbezogenen Informationen die limitierte Größe des verfügbaren Speicherplatzes übersteigen.

Die Aspekte der Datengenerierung sowie Verarbeitung stellen ebenfalls mögliche Limitationen dar. Dies hängt wie die zuvor erwähnte Limitation, ebenfalls vom Anwendungsszenario ab, in welchem der Prototyp genutzt wird. Möglicherweise muss eine Anpassung der in dieser Arbeit gewählten Technologien vorgenommen werden, um weitere Datenformate und Kommunikationsmuster zwischen Datenverarbeitungsschritten zu nutzen. Die Datengenerierung innerhalb des Car-Service orientiert sich dabei an einem selbst erstellten Datenformat, da kein Zugang zu reellen Werten eines Fahrzeugs bestand. Abhängig von diesem Faktor ist es möglich, dass weitere Datenaufbereitungs- oder Konvertierungsschritte innerhalb der modellierten Datenpipeline notwendig sind.

Eine weitere Limitation bezieht sich auf die Funktionalität. Updates von Datenpipelines können je nach Anwendungsfall unterschiedlich häufig benötigt werden. Beispielsweise wäre das Update-Verhalten für sicherheitsrelevante Pipelines, wie die Berechnung von Kollisionsvorhersagen, ein bedeutsamerer Bestandteil als bei der Analyse von Emissionswerten. Das momentane Deployment-Verfahren sieht es vor eine Datenpipeline als Gesamtpaket zu aktualisieren. Hierfür ist zunächst das Löschen und anschließend ein neues Deployment der modifizierten Pipeline notwendig. Dieses Vorgehen nimmt mehr Zeit für einen Deployment-Prozess in Anspruch, im Vergleich zu solchen, in welchen lediglich notwendige Komponenten ersetzt werden.

8 Zusammenfassung und Ausblick

Durch die Weiterentwicklung von Personenkraftwagen zu Software-Defined-Cars erhöhte sich die Zahl elektronischer Komponenten auf den Straßen in den letzten Jahren stetig. Gleichzeitig stellt der wachsende Datenaustausch die Forschung und Industrie vor neue Herausforderungen, speziell im Umgang und der Verarbeitung anfallender Informationen. Für diesen Zweck wurde in einer vorangegangenen Arbeit eine Anwendung zur Modellierung von Datenpipelines erstellt. Anhand solcher Pipelines kann der Datenfluss zwischen Datenquellen über Filter bis hin zu Datensenken modelliert werden.

Aufbauend auf der vorangegangenen Arbeit wurde ein Konzept für das Deployment und die Ausführung von Datenpipelines entwickelt, welches aus drei Teilen besteht. Im ersten Teil der Arbeit werden Konzepte beschrieben, welche eine Modellierung von verteilten Datenpipelines ermöglichen. Diese orientieren sich an der vorangegangenen Arbeit und erweitern diese um zusätzliche Funktionalitäten zum Datenaustausch mit anderen Komponenten. Für das Deployment, den zweiten Teil des Konzepts, werden notwendige Schritte vorgestellt, die dabei beachtet werden müssen. Darüber hinaus werden verschiedene Deploymentansätze anhand ausgewählter Charakteristiken miteinander verglichen. Der dritte und letzte Teil des Konzepts beinhaltet relevante Aspekte für die Ausführung von Datenpipelines. Dazu zählen eine wiederverwendbare Architektur zur Entkopplung unterschiedlicher Funktionalitäten sowie das Aufzeigen möglicher Kommunikations- und Monitoring-Ansätze.

Anhand des Konzepts wurde anschließend ein Prototyp erstellt. Dieser basiert auf einem Anwendungsszenario, welches eine Emissionsmessung auf Basis der von einem Fahrzeug versendeten Informationen ermöglicht. Als Lösung wurde eine Anwendung entwickelt, welche aus einem Modellierungswerkzeug, Deployment-Service und Container-Orchestrator besteht. Durch die Nutzung von Kubernetes konnte eine Umgebung in Form eines Clusters für das Deployment von Datenpipelines geschaffen werden. Innerhalb des Clusters wird Kafka zur Kommunikation zwischen den Bestandteilen von Datenpipelines genutzt. Zusätzlich wurden Datenquellen, Filter und Datensenken in Form von Microservices entwickelt. Diese werden gemeinsam als Datenpipeline deployed und spiegeln mit ihrer Funktionalität das Anwendungsszenario wider.

Zusammenfassend lässt sich festhalten, dass der entwickelte Prototyp die zuvor definierten Anforderungen aus Kapitel 4 vollumfänglich erfüllt. Mithilfe des Konzeptes und der Anwendung, welche anhand des beschriebenen Anwendungsszenarios entwickelt wurde, konnten Skalierbarkeit, Heterogenität, Portabilität, Erweiterbarkeit, Robustheit/Verfügbarkeit und Automatisierbarkeit erreicht werden. Diverse Anforderungen konnten in mehr als einem Bereich abgedeckt werden, so skaliert die Anwendung beispielsweise horizontal, wie auch in der Lastverteilung ankommender Anfragen. Gleichermaßen wird die technologische Heterogenität zum einen durch die gewählte Architektur, zum anderen durch die verwendete Deploymentstrategie gewährleistet.

Ausblick

Dieser Abschnitt befasst sich mit dem Ausblick, in dem Möglichkeiten für die zukünftige Forschung vorgestellt werden, welche auf dieser Arbeit aufbauen. Hierfür werden die Limitationen erneut aufgegriffen und darauf basierende Lösungsvorschläge erbracht.

Die erste Möglichkeit zur Weiterentwicklung stellt die Anbindung einer Datenbank dar. Durch die Speicherung modellbezogener Informationen könnte eine Verbesserung der Skalierbarkeit des Deployment-Service erlangt werden. Diese Erweiterung ermöglicht es die Datenhaltung vom Deployment-Service zu entkoppeln, wodurch beide Bestandteile unabhängig voneinander skalieren können. So könnten beispielsweise neue Instanzen des Deployment-Service erzeugt werden, welche Zugriff zur Datenbank bekommen.

Das Anbinden einer Datenbank würde die Möglichkeit bieten, das Konzept von Nutzern einzuführen. Die Architektur der Anwendung ermöglicht es, dass das Modellierungswerkzeug und der Deployment-Service unabhängig voneinander betrieben werden können. Durch die Einführung von Nutzern und Nutzergruppen könnte die Nutzbarkeit der Anwendung verbessert werden. Diese Anpassung würde es erlauben, Nutzern nur noch Teilmengen aller deployten Datenpipelines im Modellierungswerkzeug anzuzeigen, und so die Verwaltung fremder Modelle zu verbieten. Durch die Einführung von Nutzergruppen wäre es weiterhin möglich Modelle verschiedener Nutzer anzuzeigen, die der gleichen Nutzergruppe zugehörig sind. Dies würde die Möglichkeit eröffnen Benutzergruppen für verschiedene Unternehmen oder Institute einzusetzen, sodass die Benutzer einer Gruppe Zugang zu denselben Modellen haben, aber nicht mit den Datenpipelines anderer Benutzergruppen interagieren können.

Als weitere Limitation wurde die Komplexität von Datenpipeline-Updates identifiziert. Auch dieser Punkt kann durch die Anbindung einer Datenbank verbessert werden. Um einzelne Schritte innerhalb bestehender Datenpipelines zu verändern, ohne die gesamte Datenpipeline erneut deployen zu müssen, wäre zusätzlich eine Erweiterung des Deployment-Service notwendig. Ein Ansatz wäre es, das bestehende Modell aus der Datenbank zu laden und mit dem neuen Modell zu vergleichen. Durch die Änderungen innerhalb einzelner Komponenten lässt sich so analysieren, welche Datenpipeline-Bestandteile ersetzt werden müssen.

Literaturverzeichnis

[AAE16] N. Alshuqayran, N. Ali, R. Evans. „A Systematic Mapping Study in Microservice Architecture“. In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, Nov. 2016. DOI: 10.1109/soca.2016.15 (zitiert auf S. 34).

[AGP17] S. Amini, I. Gerostathopoulos, C. Prehofer. „Big data analytics architecture for real-time traffic control“. In: *2017 5th IEEE International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. IEEE, Juni 2017. DOI: 10.1109/mtits.2017.8005605 (zitiert auf S. 24, 31, 75).

[AKCM17] A. Akbar, A. Khan, F. Carrez, K. Moessner. „Predictive Analytics for Complex IoT Data Streams“. In: *IEEE Internet of Things Journal* 4.5 (Okt. 2017), S. 1571–1582. DOI: 10.1109/jiot.2017.2712672 (zitiert auf S. 31).

[AMS+14] J. J. Anaya, P. Merdrignac, O. Shagdar, F. Nashashibi, J. E. Naranjo. „Vehicle to pedestrian communications for protection of vulnerable road users“. In: *2014 IEEE Intelligent Vehicles Symposium Proceedings*. IEEE, Juni 2014. DOI: 10.1109/ivs.2014.6856553 (zitiert auf S. 20).

[Apa22] Apache Kafka. „Kafka - Powered By“. In: (2022). URL: https://kafka.apache.org/powered-by#:~:text=Today%2C%20Kafka%20is%20used%20by,strategies%20with%20event%20streaming%20architecture (zitiert auf S. 84).

[AZ05] P. Avgeriou, U. Zdun. „Architectural patterns revisited – a pattern language“. In: *In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee*. 2005, S. 1–39. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.141.7444 (zitiert auf S. 38).

[Ber14] D. Bernstein. „Containers and Cloud: From LXC to Docker to Kubernetes“. In: *IEEE Cloud Computing* 1.3 (Sep. 2014), S. 81–84. DOI: 10.1109/mcc.2014.51 (zitiert auf S. 25).

[BGF+13] J. Barrachina, P. Garrido, M. Fogue, F. J. Martinez, J.-C. Cano, C. T. Calafate, P. Manzoni. „Road Side Unit Deployment: A Density-Based Approach“. In: *IEEE Intelligent Transportation Systems Magazine* 5.3 (2013), S. 30–39. DOI: 10.1109/mits.2013.2253159 (zitiert auf S. 21, 23).

[BI16] V. A. Butakov, P. Ioannou. „Personalized Driver Assistance for Signalized Intersections Using V2I Communication“. In: *IEEE Transactions on Intelligent Transportation Systems* 17.7 (Juli 2016), S. 1910–1919. DOI: 10.1109/tits.2016.2515023 (zitiert auf S. 22).

[Bon00] A. B. Bondi. „Characteristics of scalability and their impact on performance“. In: *Proceedings of the second international workshop on Software and performance - WOSP '00*. ACM Press, 2000. DOI: 10.1145/350391.350432 (zitiert auf S. 34).

[BSM20] D. Balla, C. Simon, M. Maliosz. „Adaptive scaling of Kubernetes pods“. In: *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Apr. 2020. DOI: 10.1109/noms47738.2020.9110428 (zitiert auf S. 83).

[BSN14] M. Bagheri, M. Siekkinen, J. K. Nurminen. „Cellular-based vehicle to pedestrian (V2P) adaptive communication for collision avoidance“. In: *2014 International Conference on Connected Vehicles and Expo (ICCVE)*. IEEE, Nov. 2014. DOI: 10.1109/iccve.2014.7297588 (zitiert auf S. 20).

[Bun22] Bundesministerium für Digitales und Verkehr. „Neue Förderrichtlinie „Autonomes und vernetztes Fahren in öffentlichen Verkehren“ veröffentlicht“. In: (2022). URL: https://www.bmvi.de/SharedDocs/DE/Artikel/StV/Strassenverkehr/foerili-autonomes-fahren.html (zitiert auf S. 15).

[CFH+98] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, A. L. Wolf. *A characterization framework for software deployment technologies*. Techn. Ber. Colorado State Univ Fort Collins Dept of Computer Science, 1998. URL: https://apps.dtic.mil/sti/citations/ADA452086 (zitiert auf S. 41).

[Cha02] C.-Y. Chan. „A treatise on crash sensing for automotive air bag systems“. In: *IEEE/ASME Transactions on Mechatronics* 7.2 (Juni 2002), S. 220–234. DOI: 10.1109/tmech.2002.1011259 (zitiert auf S. 15).

[CHS+17] S. Chen, J. Hu, Y. Shi, Y. Peng, J. Fang, R. Zhao, L. Zhao. „Vehicle-to-Everything (v2x) Services Supported by LTE-Based Systems and 5G“. In: *IEEE Communications Standards Magazine* 1.2 (2017), S. 70–76. DOI: 10.1109/mcomstd.2017.1700015 (zitiert auf S. 18).

[CKP+11] J.-M. Chung, M. Kim, Y.-S. Park, M. Choi, S. Lee, H. S. Oh. „Time Coordinated V2I Communications and Handover for WAVE Networks“. In: *IEEE Journal on Selected Areas in Communications* 29.3 (März 2011), S. 545–558. DOI: 10.1109/jsac.2011.110305 (zitiert auf S. 22).

[CM16] R. Coppola, M. Morisio. „Connected Car“. In: *ACM Computing Surveys* 49.3 (Dez. 2016), S. 1–36. DOI: 10.1145/2971482 (zitiert auf S. 15).

[CRR+18] M. Chowdhury, M. Rahman, A. Rayamajhi, S. M. Khan, M. Islam, Z. Khan, J. Martin. „Lessons Learned from the Real-World Deployment of a Connected Vehicle Testbed“. In: *Transportation Research Record: Journal of the Transportation Research Board* 2672.22 (Okt. 2018), S. 10–23. DOI: 10.1177/0361198118799034 (zitiert auf S. 15, 30).

[Cur16] E. Curry. „The Big Data Value Chain: Definitions, Concepts, and Theoretical Approaches“. In: *New Horizons for a Data-Driven Economy*. Springer International Publishing, 2016, S. 29–37. DOI: 10.1007/978-3-319-21569-3_3 (zitiert auf S. 23, 75, 76).

[CWG13] C. Cai, Y. Wang, G. Geers. „Vehicle-to-infrastructure communication-based adaptive traffic signal control“. In: *IET Intelligent Transport Systems* 7.3 (Sep. 2013), S. 351–360. DOI: 10.1049/iet-its.2011.0150 (zitiert auf S. 22).

[DCR+18] Y. Du, M. Chowdhury, M. Rahman, K. Dey, A. Apon, A. Luckow, L. B. Ngo. „A Distributed Message Delivery Infrastructure for Connected Vehicle Technology Applications“. In: *IEEE Transactions on Intelligent Transportation Systems* 19.3 (März 2018), S. 787–801. DOI: 10.1109/tits.2017.2701799 (zitiert auf S. 31, 33, 75).

[DF10] K. David, A. Flach. „CAR-2-X and Pedestrian Safety“. In: *IEEE Vehicular Technology Magazine* 5.1 (März 2010), S. 70–76. DOI: 10.1109/mvt.2009.935536 (zitiert auf S. 20, 21).

[DH19] D. Del Gaudio, P. Hirmer. „A lightweight messaging engine for decentralized data processing in the Internet of Things“. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Aug. 2019), S. 39–48. DOI: 10.1007/s00450-019-00410-z (zitiert auf S. 30).

[Dim11] G. Dimitrakopoulos. „Intelligent transportation systems based on internet-connected vehicles: Fundamental research areas and challenges“. In: *2011 11th International Conference on ITS Telecommunications*. IEEE, Aug. 2011. DOI: 10.1109/itst.2011.6060042 (zitiert auf S. 17).

[DS17] R. Dhall, V. K. Solanki. „An IoT Based Predictive Connected Car Maintenance Approach“. In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.3 (2017), S. 16. DOI: 10.9781/ijimai.2017.433 (zitiert auf S. 15, 30).

[Ede16] M. Eder. „Hypervisor- vs. Container-based Virtualization“. en. In: (2016). DOI: 10.2313/NET-2016-07-1_01 (zitiert auf S. 24–26, 44).

[Eic07] S. Eichler. „Performance Evaluation of the IEEE 802.11p WAVE Communication Standard“. In: *2007 IEEE 66th Vehicular Technology Conference*. IEEE, Sep. 2007. DOI: 10.1109/vetecf.2007.461 (zitiert auf S. 20).

[FJM+01] L. Figueiredo, I. Jesus, J. Machado, J. Ferreira, J. M. de Carvalho. „Towards the development of intelligent transportation systems“. In: *ITSC 2001. 2001 IEEE Intelligent Transportation Systems. Proceedings (Cat. No.01TH8585)*. IEEE, 2001. DOI: 10.1109/itsc.2001.948835 (zitiert auf S. 17).

[FZY21] G. Fu, Y. Zhang, G. Yu. „A Fair Comparison of Message Queuing Systems“. In: *IEEE Access* 9 (2021), S. 421–432. DOI: 10.1109/access.2020.3046503 (zitiert auf S. 31).

[GSB12] J. Gozalvez, M. Sepulcre, R. Bauza. „IEEE 802.11p vehicle to infrastructure communications in urban environments“. In: *IEEE Communications Magazine* 50.5 (Mai 2012), S. 176–183. DOI: 10.1109/mcom.2012.6194400 (zitiert auf S. 23).

[GSS+18] F. Giust, V. Sciancalepore, D. Sabella, M. C. Filippou, S. Mangiante, W. Featherstone, D. Munaretto. „Multi-Access Edge Computing: The Driver Behind the Wheel of 5G-Connected Cars“. In: *IEEE Communications Standards Magazine* 2.3 (Sep. 2018), S. 66–73. DOI: 10.1109/mcomstd.2018.1800013 (zitiert auf S. 16).

[GT07] T. Gandhi, M. Trivedi. „Pedestrian Protection Systems: Issues, Survey, and Challenges“. In: *IEEE Transactions on Intelligent Transportation Systems* 8.3 (Sep. 2007), S. 413–430. DOI: 10.1109/tits.2007.903444 (zitiert auf S. 20).

[HBC+07] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, S. Ramesh. „Performance Analysis of FlexRay-based ECU Networks“. In: *2007 44th ACM/IEEE Design Automation Conference*. 2007, S. 284–289. ISBN: 9781595936271 (zitiert auf S. 15).

[HBW+17] S. Hoque, M. S. D. Brito, A. Willner, O. Keil, T. Magedanz. „Towards Container Orchestration in Fog Computing Infrastructures“. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, Juli 2017. DOI: 10.1109/compsac.2017.248 (zitiert auf S. 26).

[Hir18] P. Hirmer. „Anforderungsbasierte Modellierung und Ausführung von Datenflussmodellen“. de. Diss. 2018. DOI: 10.18419/OPUS-9930 (zitiert auf S. 29).

[HKGF09] T. Herpel, B. Kloiber, R. German, S. Fey. „Routing of Safety-Relevant Messages in Automotive ECU Networks“. In: *2009 IEEE 70th Vehicular Technology Conference Fall*. IEEE, Sep. 2009. DOI: 10.1109/vetecf.2009.5378778 (zitiert auf S. 17).

[HL08] H. Hartenstein, K. Laberteaux. „A tutorial survey on vehicular ad hoc networks“. In: *IEEE Communications Magazine* 46.6 (Juni 2008), S. 164–171. DOI: 10.1109/mcom.2008.4539481 (zitiert auf S. 18).

[HM16] P. Hirmer, B. Mitschang. „TOSCA4Mashups: enhanced method for on-demand data mashup provisioning“. In: *Computer Science - Research and Development* 32.3-4 (Okt. 2016), S. 291–300. DOI: 10.1007/s00450-016-0330-7 (zitiert auf S. 29).

[Hol19] H. Holland. „Connected Cars“. In: *Dialogmarketing und Kundenbindung mit Connected Cars: Wie Automobilherstellern mit Daten und Vernetzung die optimale Customer Experience gelingt*. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, S. 51–81. ISBN: 978-3-658-22929-0. DOI: 10.1007/978-3-658-22929-0_3. URL: https://doi.org/10.1007/978-3-658-22929-0_3 (zitiert auf S. 15).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004. ISBN: 9780321200686 (zitiert auf S. 16).

[HWCL14] H. Hu, Y. Wen, T.-S. Chua, X. Li. „Toward Scalable Systems for Big Data Analytics: A Technology Tutorial“. In: *IEEE Access* 2 (2014), S. 652–687. DOI: 10.1109/access.2014.2332453 (zitiert auf S. 24).

[IEE10] IEEE. *IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments*. 2010. URL: https://standards.ieee.org/ieee/802.11p/3953/ (zitiert auf S. 19).

[IEE19] IEEE. *IEEE Standard for Wireless Access in Vehicular Environments (WAVE)–Identifiers*. 2019. URL: https://standards.ieee.org/ieee/1609.12/7446/ (zitiert auf S. 19).

[IEE20] IEEE. *IEEE Standard for Information Technology–Telecommunications and Information Exchange between Systems - Local and Metropolitan Area Networks–Specific Requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*. 2020. URL: https://standards.ieee.org/ieee/802.11/7028/ (zitiert auf S. 19).

[IEE90] IEEE. „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), S. 1–84. DOI: 10.1109/ieeestd.1990.101064 (zitiert auf S. 35).

[JBB+19] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli. „Container Orchestration Engines: A Thorough Functional and Performance Comparison“. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. IEEE, Mai 2019. DOI: 10.1109/icc.2019.8762053 (zitiert auf S. 26).

[JCB+15] S. Jafarnejad, L. Codeca, W. Bronzi, R. Frank, T. Engel. „A Car Hacking Experiment: When Connectivity Meets Vulnerability“. In: *2015 IEEE Globecom Workshops (GC Wkshps)*. IEEE, Dez. 2015. DOI: 10.1109/glocomw.2015.7413993 (zitiert auf S. 15).

[Ken11] J. B. Kenney. „Dedicated Short-Range Communications (DSRC) Standards in the United States“. In: *Proceedings of the IEEE* 99.7 (Juli 2011), S. 1162–1182. DOI: 10.1109/jproc.2011.2132790 (zitiert auf S. 19).

[Ken21] M. Kenzler. „Verteilte Datenpipelines in Software-Defined-Car-Anwendungen“. de. Magisterarb. 2021. DOI: 10.18419/OPUS-11950 (zitiert auf S. 29, 37, 38, 55, 57).

[KK10] A. Kchiche, F. Kamoun. „Centrality-based Access-Points deployment for vehicular networks“. In: *2010 17th International Conference on Telecommunications*. IEEE, 2010. DOI: 10.1109/ictel.2010.5478800 (zitiert auf S. 23).

[KNR+11] J. Kreps, N. Narkhede, J. Rao et al. „Kafka: A distributed messaging system for log processing“. In: 2011 (zitiert auf S. 83).

[KSA09] R. K. Karmani, A. Shali, G. Agha. „Actor frameworks for the JVM platform“. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java - PPPJ '09*. ACM Press, 2009. DOI: 10.1145/1596655.1596658 (zitiert auf S. 49).

[KSV17] C. G. Kominos, N. Seyvet, K. Vandikas. „Bare-metal, virtual machines and containers in OpenStack“. In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, März 2017. DOI: 10.1109/icin.2017.7899247 (zitiert auf S. 43, 44).

[Kub22] Kubernetes. „Pods“. In: (2022). URL: https://kubernetes.io/docs/concepts/workloads/pods/ (zitiert auf S. 68).

[LK10] J. Lee, C. M. Kim. „A Roadside Unit Placement Scheme for Vehicular Telematics Networks“. In: *Advances in Computer Science and Information Technology*. Springer Berlin Heidelberg, 2010, S. 196–202. DOI: 10.1007/978-3-642-13577-4_17 (zitiert auf S. 23).

[LKLA17] Z. Li, M. Kihl, Q. Lu, J. A. Andersson. „Performance Overhead Comparison between Hypervisor and Container Based Virtualization“. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, März 2017. DOI: 10.1109/aina.2017.79 (zitiert auf S. 25, 26, 45).

[LW07] F. Li, Y. Wang. „Routing in vehicular ad hoc networks: A survey“. In: *IEEE Vehicular Technology Magazine* 2.2 (2007), S. 12–22. DOI: 10.1109/mvt.2007.912927 (zitiert auf S. 18).

[LWM17] Y. Lin, P. Wang, M. Ma. „Intelligent Transportation System(ITS): Concept, Challenge and Opportunity“. In: *2017 IEEE 3rd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE, Mai 2017. DOI: 10.1109/bigdatasecurity.2017.50 (zitiert auf S. 35).

[McK14] McKinsey & Company. „What's driving the connected car“. In: (2014). URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/whats-driving-the-connected-car (zitiert auf S. 23).

[Mer+14] D. Merkel et al. „Docker: lightweight linux containers for consistent development and deployment“. In: *Linux Journal* 239.2 (2014), S. 2 (zitiert auf S. 25).

[MM16] P. Masek, T. Magnus. „Container based virtualisation for software deployment in self-driving vehicles“. Magisterarb. 2016. URL: https://hdl.handle.net/20.500.12380/237650 (zitiert auf S. 41).

[MTKM09] A. Molisch, F. Tufvesson, J. Karedal, C. Mecklenbrauker. „A survey on vehicle-to-vehicle propagation channels“. In: *IEEE Wireless Communications* 16.6 (Dez. 2009), S. 12–22. DOI: 10.1109/mwc.2009.5361174 (zitiert auf S. 20).

[MVG+12] V. Milanes, J. Villagra, J. Godoy, J. Simo, J. Perez, E. Onieva. „An Intelligent V2I-Based Traffic Management System“. In: *IEEE Transactions on Intelligent Transportation Systems* 13.1 (März 2012), S. 49–58. DOI: 10.1109/tits.2011.2178839 (zitiert auf S. 22).

[NRSG17] E. Ndashimye, S. K. Ray, N. I. Sarkar, J. A. Gutiérrez. „Vehicle-to-infrastructure communication over multi-tier heterogeneous networks: A survey“. In: *Computer Networks* 112 (Jan. 2017), S. 144–166. DOI: 10.1016/j.comnet.2016.11.008 (zitiert auf S. 21, 22).

[OKF10] S. Oh, J.-H. Kim, G. Fox. „Real-time performance analysis for publish/subscribe systems“. In: *Future Generation Computer Systems* 26.3 (März 2010), S. 318–323. DOI: 10.1016/j.future.2009.09.001 (zitiert auf S. 50).

[Org+15] W. H. Organization et al. *Global status report on road safety 2015: Summary*. 2015. URL: https://www.afro.who.int/publications/global-status-report-road-safety-2015 (zitiert auf S. 20).

[OWMW15] L. B. Othmane, H. Weffers, M. M. Mohamad, M. Wolf. „A Survey of Security and Privacy in Connected Vehicles“. In: *Wireless Sensor and Mobile Ad-Hoc Networks*. Springer New York, 2015, S. 217–247. DOI: 10.1007/978-1-4939-2468-4_10 (zitiert auf S. 17, 18).

[Pah15] C. Pahl. „Containerization and the PaaS Cloud“. In: *IEEE Cloud Computing* 2.3 (Mai 2015), S. 24–31. DOI: 10.1109/mcc.2015.51 (zitiert auf S. 25).

[PGKM20] A. M. Potdar, N. D. G, S. Kengond, M. M. Mulla. „Performance Evaluation of Docker Container and Virtual Machine“. In: *Procedia Computer Science* 171 (2020), S. 1419–1428. DOI: 10.1016/j.procs.2020.04.152 (zitiert auf S. 45).

[PSA+17] O. Popescu, S. Sha-Mohammad, H. Abdel-Wahab, D. C. Popescu, S. El-Tawab. „Automatic Incident Detection in Intelligent Transportation Systems Using Aggregation of Traffic Parameters Collected Through V2I Communications“. In: *IEEE Intelligent Transportation Systems Magazine* 9.2 (2017), S. 64–75. DOI: 10.1109/mits.2017.2666578 (zitiert auf S. 22).

[RBV03] R. V. Renesse, K. P. Birman, W. Vogels. „Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining“. In: *ACM Transactions on Computer Systems* 21.2 (Mai 2003), S. 164–206. DOI: 10.1145/762483.762485 (zitiert auf S. 33).

[RSGJ13] K. Raman, A. Swaminathan, J. Gehrke, T. Joachims. „Beyond myopic inference in big data pipelines“. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, Aug. 2013. DOI: 10.1145/2487575.2487588 (zitiert auf S. 24).

[SAK+13] Y. Simmhan, S. Aman, A. Kumbhare, R. Liu, S. Stevens, Q. Zhou, V. Prasanna. „Cloud-Based Software Platform for Big Data Analytics in Smart Grids“. In: *Computing in Science & Engineering* 15.4 (Juli 2013), S. 38–47. DOI: 10.1109/mcse.2013.39 (zitiert auf S. 24).

[SHM+14] K.-T. Seo, H.-S. Hwang, I.-Y. Moon, O.-Y. Kwon, B.-J. Kim. „Performance comparison analysis of linux container and virtual machine for building cloud“. In: *Advanced Science and Technology Letters* 66.105-111 (2014), S. 2 (zitiert auf S. 26).

[SLY+16] H. Seo, K.-D. Lee, S. Yasukawa, Y. Peng, P. Sartori. „LTE evolution for vehicle-to-everything services“. In: *IEEE Communications Magazine* 54.6 (Juni 2016), S. 22–28. DOI: 10.1109/mcom.2016.7497762 (zitiert auf S. 17, 18).

[SME20] H. Sami, A. Mourad, W. El-Hajj. „Vehicular-OBUs-As-On-Demand-Fogs: Resource and Context Aware Deployment of Containerized Micro-Services“. In: *IEEE/ACM Transactions on Networking* 28.2 (Apr. 2020), S. 778–790. DOI: 10.1109/tnet.2020.2973800 (zitiert auf S. 15, 30).

[SMTS13] Y. Shoukry, P. Martin, P. Tabuada, M. Srivastava. „Non-invasive Spoofing Attacks for Anti-lock Braking Systems“. In: *Cryptographic Hardware and Embedded Systems - CHES 2013*. Springer Berlin Heidelberg, 2013, S. 55–72. DOI: 10.1007/978-3-642-40349-1_4 (zitiert auf S. 15).

[SNA+13] I. Studnia, V. Nicomette, E. Alata, Y. Deswarte, M. Kaaniche, Y. Laarouchi. „Survey on security threats and protection mechanisms in embedded automotive networks“. In: *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, Juni 2013. DOI: 10.1109/dsnw.2013.6615528 (zitiert auf S. 17).

[SNH08] C. Sugimoto, Y. Nakamura, T. Hashimoto. „Prototype of pedestrian-to-vehicle communication system for the prevention of pedestrian accidents using both 3G wireless and WLAN communication“. In: *2008 3rd International Symposium on Wireless Pervasive Computing*. IEEE, Mai 2008. DOI: 10.1109/iswpc.2008.4556313 (zitiert auf S. 20).

[SS13] S. Sagiroglu, D. Sinanc. „Big data: A review“. In: *2013 International Conference on Collaboration Technologies and Systems (CTS)*. IEEE, Mai 2013. DOI: 10.1109/cts.2013.6567202 (zitiert auf S. 23).

[SS19] P. Sewalkar, J. Seitz. „Vehicle-to-Pedestrian Communication for Vulnerable Road Users: Survey, Design Considerations, and Challenges“. In: *Sensors* 19.2 (Jan. 2019), S. 358. DOI: 10.3390/s19020358 (zitiert auf S. 18, 21).

[TATS18] K. Takahashi, K. Aida, T. Tanjo, J. Sun. „A Portable Load Balancer for Kubernetes Cluster“. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM, Jan. 2018. DOI: 10.1145/3149457.3149473 (zitiert auf S. 83).

[TLP+19] E. Truyen, D. V. Landuyt, D. Preuveneers, B. Lagaisse, W. Joosen. „A Comprehensive Feature Comparison Study of Open-Source Container Orchestration Frameworks“. In: *Applied Sciences* 9.5 (März 2019), S. 931. DOI: 10.3390/app9050931 (zitiert auf S. 26).

[UJ16] G. A. Ubiergo, W.-L. Jin. „Mobility and environment improvement of signalized networks through Vehicle-to-Infrastructure (V2I) communications“. In: *Transportation Research Part C: Emerging Technologies* 68 (Juli 2016), S. 70–82. DOI: 10.1016/j.trc.2016.03.010 (zitiert auf S. 22).

[WCL09] C.-x. Wang, X. Cheng, D. Laurenson. „Vehicle-to-vehicle channel modeling and measurements: recent advances and future challenges“. In: *IEEE Communications Magazine* 47.11 (Nov. 2009), S. 96–103. DOI: 10.1109/mcom.2009.5307472 (zitiert auf S. 15).

[WES03] P. Weibring, H. Edner, S. Svanberg. „Versatile mobile lidar system for environmental monitoring“. In: *Applied Optics* 42.18 (Juni 2003), S. 3583. DOI: 10.1364/ao.42.003583 (zitiert auf S. 15).

[WMG17] X. Wang, S. Mao, M. X. Gong. „An Overview of 3GPP Cellular Vehicle-to-Everything Standards“. In: *GetMobile: Mobile Computing and Communications* 21.3 (Nov. 2017), S. 19–25. DOI: 10.1145/3161587.3161593 (zitiert auf S. 18).

[WSM+15] E. Wilhelm, J. Siegel, S. Mayer, L. Sadamori, S. Dsouza, C.-K. Chau, S. Sarma. „Cloudthink: a scalable secure platform for mirroring transportation systems in the cloud“. In: *Transport* 30.3 (Okt. 2015), S. 320–329. DOI: 10.3846/16484142.2015.1079237 (zitiert auf S. 33).

[XYP14] X. Xu, H. Yu, X. Pei. „A Novel Resource Scheduling Approach in Container Based Clouds“. In: *2014 IEEE 17th International Conference on Computational Science and Engineering*. IEEE, Dez. 2014. DOI: 10.1109/cse.2014.77 (zitiert auf S. 25).

[YMF06] S. Yousefi, M. Mousavi, M. Fathy. „Vehicular Ad Hoc Networks (VANETs): Challenges and Perspectives“. In: *2006 6th International Conference on ITS Telecommunications*. IEEE, Juni 2006. DOI: 10.1109/itst.2006.289012 (zitiert auf S. 18).

[YQC+19] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, L. Qianqian. „Message-oriented Middleware: A Review“. In: *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, Aug. 2019. DOI: 10.1109/bigcom.2019.00023 (zitiert auf S. 31).

[YSCH19] S.-R. Yang, Y.-J. Su, Y.-Y. Chang, H.-N. Hung. „Short-Term Traffic Prediction for Edge Computing-Enhanced Autonomous and Connected Cars“. In: *IEEE Transactions on Vehicular Technology* 68.4 (Apr. 2019), S. 3140–3153. DOI: 10.1109/tvt.2019.2899125 (zitiert auf S. 30).

[ZGC19] S. Zeadally, J. Guerrero, J. Contreras. „A tutorial survey on vehicle-to-vehicle communications“. In: *Telecommunication Systems* 73.3 (Dez. 2019), S. 469–489. DOI: 10.1007/s11235-019-00639-8 (zitiert auf S. 19, 20).

[ZYW+19] L. Zhu, F. R. Yu, Y. Wang, B. Ning, T. Tang. „Big Data Analytics in Intelligent Transportation Systems: A Survey“. In: *IEEE Transactions on Intelligent Transportation Systems* 20.1 (Jan. 2019), S. 383–398. DOI: 10.1109/tits.2018.2815678 (zitiert auf S. 23, 24).

Alle URLs wurden zuletzt am 03. 10. 2022 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