Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Konzept und Umsetzung eines Data Lakes für Connected-Car-Umgebungen

Isabella Kutger

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Dr. rer. nat. Pascal Hirmer
Beginn am:	1. Juni 2021
Beendet am:	1. Dezember 2021

Kurzfassung

Die Ausstattung von Fahrzeugen und deren Infrastruktur mit Sensorik und Rechenressourcen ermöglicht eine umfangreiche Datenerfassung im Verkehrswesen. Diese Daten können an andere Fahrzeuge oder zentrale Server weitergeleitet werden. Dabei werden Fahrzeuge, die mit Sensorik ausgestattet sind, als *Connected Cars* bezeichnet. In Connected-Car-Umgebungen müssen Fahrzeuge direkt auf Ereignisse reagieren, weshalb eine Echtzeitverarbeitung essentiell ist. Nicht nur eine Echtzeitverarbeitung der Daten muss beachtet werden, sondern auch unterschiedliche Strukturen und Formate. Diese unterscheiden sich je nach Datenquelle und Hersteller. Deshalb stellen Datenspeicherung und -verarbeitung in Connected-Car-Umgebungen eine große Herausforderung dar. Eine sinnvolle Option hierfür ist ein Data Lake. Ein Data Lake ist ein effektiver Datenspeicher, der heterogene Daten zusammen bringt. Außerdem werden Metadaten genutzt, um auf relevante Daten zuzugreifen. Nur so ist eine Analyse und Auswertung der Daten für komplexe Verkehrsszenarien möglich. Aus diesem Grund wird in dieser Arbeit ein Konzept eines verteilten Data Lakes vorgestellt. Dazu werden zunächst Anforderungen definiert, die ein Data Lake in Connected-Car-Umgebungen erfüllen sollte. Basierend auf diesen Anforderungen wird das Konzept von mehreren verteilten Data Lagoons und einem zentralen Data Lake eingeführt. Eine Data Lagoon ist ein Teil des Data Lake, der z.B. auf den Rechenressourcen von Ampeln oder Laternen gehostet werden kann. Data Lagoons sammeln Daten der Connected Cars. Diese werden an den zentralen Data Lake weitergeleitet. Dabei besitzen sowohl Data Lake als auch Data Lagoon eine Datenhaltung und Datenverarbeitung. Die Architekturen der Komponenten basieren auf Lambda- und Kappa-Architektur. Im Verlauf der Arbeit wird eine prototypische Implementierung des Konzepts angefertigt. Dieser Prototyp erhält Daten unterschiedlicher Strukturen und Arten. Für diese Daten werden verschiedene Verarbeitungsprozesse aufgezeigt. Weiterhin werden Metadaten, die während der Verarbeitung erweitert werden, mitgesendet. Deren Speicherung wird separat behandelt.

Disclaimer: In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application Programming Interface. 19

ASIL Automotive Safety Integrity Level. 30

DSGVO Datenschutz-Grundverordnung. 28

ECU Electronic Control Unit. 21

ETL Extract-Transform-Load. 15

GPS Global Positioning System. 49

IoT Internet of Things. 19

IP Internet Protocol. 64

ITS Intelligent Transport System. 20

JSON Java Script Object Notation. 9

OBU Onboard Unit. 20

RSU Road Side Unit. 20

TDLIoT Topic Description Language for the Internet of Things. 40

UI User Interface. 63

V2I Vehicle To Internet. 22

V2R Vehicle To Road Infrastructure. 22

V2S Vehicle To Sensors. 22

V2V Vehicle To Vehicle. 22

VANET Vehicular Ad-Hoc Network. 22

VRU Vulnerable Road User. 20

1. Einleitung

Mit der stärkeren Vernetzung und neuen technischen Möglichkeiten, ist auch eine Veränderung im Verkehrswesen nicht fern. Fahrzeuge und Umgebungen, wie Ampeln und Straßenlaternen, werden mit immer mehr Sensorik und Rechenressourcen ausgestattet. Dies ermöglicht eine Erfassung von Daten über die Fahrzeuge und deren Umgebung. Diese Daten können an andere Fahrzeuge oder Infrastrukturen, z.B. an eine Cloud, weitergeleitet werden. Dabei werden Fahrzeuge, die mit Sensorik ausgestattet und eine Verbindung zum Internet besitzen, als *Connected Cars* bezeichnet [CM16]. Da Connected Cars direkt auf Ereignisse, wie Hindernisse auf der Fahrbahn, reagieren müssen, ist in Connected-Car-Umgebungen eine Echtzeitverarbeitung essentiell [CM16; PKÅ+20]. Dabei müssen gleichzeitig enorme Datenmengen berücksichtigt und verarbeitet werden. Schon heute produziert ein Connected Car bis zur 25GB Daten pro Stunde [Wol16]. Ein Großteil dieser Daten sind Sensor-, Bild- und Videodaten. Weitere Daten zur Analyse können aus externen Quellen und von anderen Akteuren der Umgebung, wie Fußgänger, Ampeln oder Werkstätten, stammen. Nicht nur diese Daten unterscheiden sich in Struktur und Format, sondern auch die Daten der Connected Cars je nach Hersteller. Dies erschwert eine herstellerübergreifende Kollaboration zwischen Fahrzeugen. Eine solche Kollaboration ist nötig, um unter anderem autonomes Fahren flächendeckend zu ermöglichen. Weiterhin ist es notwendig, heterogene Daten von Fahrzeugen und ihrer Umgebung an einem zentralen Punkt zu speichern. Nur so sind Auswertungen, Analysen und Simulationen für komplexe Szenarien basierend auf diesen Daten durchführbar. Hierfür könnte ein Data Lake genutzt werden.

Data Lakes ermöglichen eine Speicherung von heterogenen Daten. Darüber hinaus kann mit Hilfe eines Metadatenmanagements effektiv auf die gespeicherten Daten zugegriffen werden. Dadurch können nutzende Applikationen und Services für sie passende Daten schnell auffinden [Mat17]. Ein Data Lake bietet eine skalierbare Sammlung und Speicherung der Daten mit geringem Aufwand. Weiterhin können Daten mit unterschiedlichen Formaten und Strukturen am selben Ort gespeichert werden. Die Daten werden erst bei Nutzung in das benötigte Schema transformiert. Außerdem werden verschiedene Arten der Datenverarbeitung unterstützt, weshalb eine Stream-Verarbeitung für Echtzeitanalysen eingesetzt werden kann [Fan15].

Daher ist das Ziel dieser Arbeit die Konzeption und Umsetzung eines Data Lakes, der einen effektiven Datenspeicher für heterogene Datenquellen darstellt. Das vorgestellte Konzept besteht aus mehreren verteilten Data Lagoons und einem Data Lake. Eine Data Lagoon ist ein Teil des Data Lakes, der z.B. auf in Ampeln oder Laternen verbauten Ressourcen gehostet wird. Data Lagoons verfügen über eigene Möglichkeiten zur Verarbeitung und Speicherung von Daten. Somit werden Daten nah am Fahrzeug gespeichert und analysiert, was Übertragungswege erheblich verkürzt. Im Data Lake kommen die Daten aus den Data Lagoons und weiteren externen Quellen zusammen. Dabei besteht die Möglichkeit, Daten ausschließlich in der Data Lagoon zu speichern oder diese nur in vorverarbeiteter Form weiterzuleiten. Dies kann zu einer Erhöhung von Privatheit und Sicherheit beitragen. Besonders Privatheit und Sicherheit spielen in der Connected-Car-Umgebung eine große Rolle, da viele personenbezogene Daten gesammelt werden [LML15]. Um eine Echtzeitverarbeitung sicherzustellen, können sowohl Data Lagoon als auch Data Lake Daten in Streams verarbeiten. Im

Verlauf der Arbeit werden auf Anforderungen, die ein Data Lake in einer Connected-Car-Umgebung erfüllen muss, sowie Aufbau und Umsetzung des Konzepts eingegangen. Außerdem wird ein Prototyp bestehend aus Data Lagoon und Data Lake vorgestellt.

Gliederung

Der folgende Abschnitt zeigt die Unterteilung und Thematik der Arbeit.

Kapitel 2 - Grundlagen gibt einen Überblick über die benötigten Grundlagen zu Data Lakes und Connected Cars.

Kapitel 3 - Verwandte Arbeiten stellt einige Arbeiten, die sich mit Big Data im Verkehrsbereich beschäftigen, vor und grenzt sie von dieser Arbeit ab.

Kapitel 4 - Anforderungen an Data Lakes in Connected-Car-Umgebungen definiert Anforderungen an einen Data Lake in Connected-Car-Umgebungen. Zunächst wird auf Herausforderungen an Big Data in einer Connected-Car-Umgebung eingegangen. Schließlich werden daraus die Anforderungen abgeleitet.

Kapitel 5 - Konzept eines Data Lakes in Connected-Car-Umgebungen beschreibt das erarbeitete Konzept für einen Data Lake in Connected-Car-Umgebungen. Dabei wird die Architektur des gesamten Data Lakes erklärt. Danach wird auf die Bestandteile Data Lagoon, Data Lake und Registry eingegangen.

Kapitel 6 - Evaluierung der Anforderungen evaluiert das erarbeitete Konzept gegen die zuvor definierten Anforderungen. Für jede Anforderung wird genau betrachtet, wie sie im Konzept umgesetzt ist.

Kapitel 7 - Implementierung eines Data Lakes in Connected-Car-Umgebungen stellt die Implementierung des Prototyps vor. Hier wird auf das Gesamtsystem sowie jedes Bestandteil eingegangen. Diese umfassen Data Lagoon, Data Lake, Data Registry und Datengenerator.

Kapitel 8 - Zusammenfassung und Ausblick fasst die Arbeit zusammen und geht auf mögliche Anknüpfungspunkte für zukünftige Arbeiten ein.

2. Grundlagen

In diesem Kapitel wird zunächst auf die wichtigsten Grundlagen für das Konzept und die Implementierung eines Data Lakes in Connected-Car-Umgebungen eingegangen. In Abschnitt 2.1 werden die Grundidee und einige Aspekte für die Konzeption von Data Lakes vorgestellt. Anschließend werden in Abschnitt 2.2 Connected Cars und deren Umgebung beleuchtet.

2.1. Data Lakes

Data Lakes sind, ähnlich wie Data Warehouses, eine Möglichkeit zur Verwaltung von Big Data. Dabei unterscheiden sie sich in einigen Aspekten grundlegend vom traditionellen Ansatz zur Datenverwaltung. Im Folgenden wird auf diese Aspekte eingegangen. Zunächst werden in Abschnitt 2.1.1 die Idee und die Charakteristiken von Data Lakes erklärt. Danach werden in Abschnitt 2.1.2 verschiedene Architekturen für Data Lakes vorgestellt. Daraufhin wird in Abschnitt 2.1.3 auf die Anwendung von Zonenmodellen eingegangen. Abschließend wird in Abschnitt 2.1.4 die Wichtigkeit von Metadatenmanagement in Data-Lake-Umgebungen erläutert.

2.1.1. Idee und Charakteristiken

Der Begriff „Data Lake“ wurde das erste Mal 2010 von Dixon [Dix10] verwendet. Dieser Begriff sollte zur Unterscheidung eines Hadoop-basierten Datenmanagements und dem klassischen Data Warehouse dienen. Dabei sollen die Daten in ihrer Rohform von der Quelle in den Data Lake fließen, statt wie bei einem Data Warehouse in ein vordefiniertes Schema gebracht zu werden. Erst im Data Lake sollen Nutzer die Daten genauer untersuchen und Schlüsse ziehen. Diese Grundidee wurde in verschiedenen Kontexten und Interpretationen aufgegriffen, sodass es kein allgemein akzeptiertes Konzept des Data Lakes gibt [Mat17]. Viel mehr handelt es sich um verschiedene Definitionen und Sichten, die durchaus einige zentrale Charakteristiken teilen [GGH+19; RZ19a].

Laut Fang [Fan15] ist ein Data Lake ein riesiges Daten-Repository, das die Erfassung, Archivierung und Untersuchung von Rohdaten innerhalb eines Systems unterstützt. Statt die Daten in eine dem Zweck zugeschnittene Datenbank zu speichern, werden die Daten in unveränderter Form im Data Lake gespeichert. Das Ziel ist, Wissen aus den gesammelten Daten durch deren Untersuchung und Nutzung in verschiedenen Services und Applikationen zu ziehen [Mat17]. Dafür besitzt der Data Lake einige Eigenschaften. Haupteigenschaft eines Data Lakes ist das Erfassen und Speichern von Rohdaten unterschiedlicher Typen. Im Datenspeicher sollen Daten jedes Typs und jeder Größe unabhängig von Datenrate abgespeichert werden. Daher können innerhalb des Data Lakes z.B. strukturierte Daten und Mediendaten wie Texte, Bilder und Videos hinterlegt werden. Auf den gespeicherten Daten können Vorverarbeitungen und Extract-Transform-Load (ETL)-Prozesse ausgeführt werden, um die Daten zu analysieren. Weiterhin wird das eigentliche

Schema der Daten erst bei Nutzung definiert. Je nach gesuchten Attributen können für die gleichen Datensätze mehrere Schemata definiert werden. Dieser Ansatz wird *Schema-On-Read* [XRZ+13] genannt. Das Ziel ist, Flexibilität zu erreichen und Transformationen durch ETL-Prozesse erst auszuführen, wenn es für den Gebrauch der Daten notwendig wird. Dadurch wird der Aufwand bei der Datenintegration reduziert, da Daten nicht erst in ein vordefiniertes Schema gebracht werden müssen. Um eine Analyse der Daten zu ermöglichen, unterstützt der Data Lake nicht nur jede Datenart, sondern auch jede Verarbeitungsart. So können Batch-Verarbeitungen gleichermaßen wie Stream-Verarbeitungen durchgeführt werden [Fan15; Mat17]. Aufgrund ihrer Herkunft werden Data Lakes oft mit Apache Hadoop[1] und HDFS in Verbindung gebracht. Gerade die ersten Ansätze wurden mit dieser Technologie entwickelt, da eine Hadoop-Umgebung mit einer großen und heterogenen Datenmenge umgehen kann [RZ19a]. Zusätzlich sind Hadoop und die benötigte Umgebung Open-Source-Projekte, wodurch es zu einem kostengünstigen und umsetzbaren Weg für Big-Data-Management wird [Fan15]. Jedoch ist Hadoop nur eine der vielen Technologien, die für die Umsetzung eines Data Lakes genutzt werden können [GGH+19]. So können auch MongoDB, InfluxDB oder andere Datenbanken zum Speichern der Daten oder Frameworks wie Apache Flink oder Spark verwendet werden. Zusammenfassend lässt sich festhalten, dass ein Data Lake eine geeignete Datenmangementlösung für Systeme mit heterogenen Quellen ist. Die Daten dieser Quellen lassen sich in ihrem originalen Format speichern und können nach verschiedenen Bedürfnissen abgefragt werden [RZ19a]. Durch den Schema-On-Read-Ansatz können Daten mit geringem Aufwand gespeichert werden, was in einer Echtzeitanwendung vorteilhaft ist.

2.1.2. Architektur

Die Architektur eines Data Lakes beruht meist auf *Ponds* oder Zonen [GGH+19; RZ19a]. Bei der Pond-Architektur [Inm16] wird der Data Lake in fünf Ponds unterteilt. Dabei sind die Datensätze immer nur in einem Pond zu finden. Bei Datenaufnahme werden die Datensätze zunächst im *Raw Pond* gespeichert. Sobald die Datensätze genutzt werden, werden sie vom Raw Pond zum *Analog, Application* oder *Textual Data Pond* übertragen. Diese Einteilung erfolgt auf Basis der Eigenschaften der Daten. Zum Beispiel werden Textdaten in den Textual Data Pond eingeteilt. Somit verbleiben im Raw Pond nur ungenutzte Datensätze und Daten, die nicht in die restlichen Ponds einteilbar sind, wie z.B. Bild- und Videodaten. Falls Daten nicht mehr benötigt werden, verlassen sie ihren Pond und werden im *Archival Data Pond* archiviert [Inm16]. Diese Architektur hat einige Nachteile, weshalb vermehrt Zonenarchitekturen genutzt werden. Unter anderem ist ein Nachteil, dass Daten für ihren jeweiligen Pond angepasst werden und dadurch nicht mehr in ihrer ursprünglicher Form vorliegen. Bei den Zonenarchitekturen gibt es verschiedene Modelle und Ansätze, auf die in Abschnitt 2.1.3 eingegangen wird.

Eine weitere wichtige Architekturentscheidung ist die Organisation von Batch- und Stream-Verarbeitung [GGH+19]. Batch-Verarbeitung ist eine Möglichkeit große Mengen an Daten zu verarbeiten. Bei diesem Vorgehen werden nur Daten berücksichtigt, die im Speicher liegen. Das bedeutet, neu ankommende Daten werden nach Start der Verarbeitung nicht berücksichtigt. Wie der Begriff Batch-Verarbeitung impliziert, werden die Daten in Stapel unterteilt. Diese Stapel werden auf Knoten verteilt und dort verarbeitet. Die Resultate stehen erst zur Verfügung, wenn alle Stapel fertig verarbeitet wurden. Dadurch kann der Gesamtprozess mitunter einige Zeit

[1] Apache Hadoop: https://hadoop.apache.org/

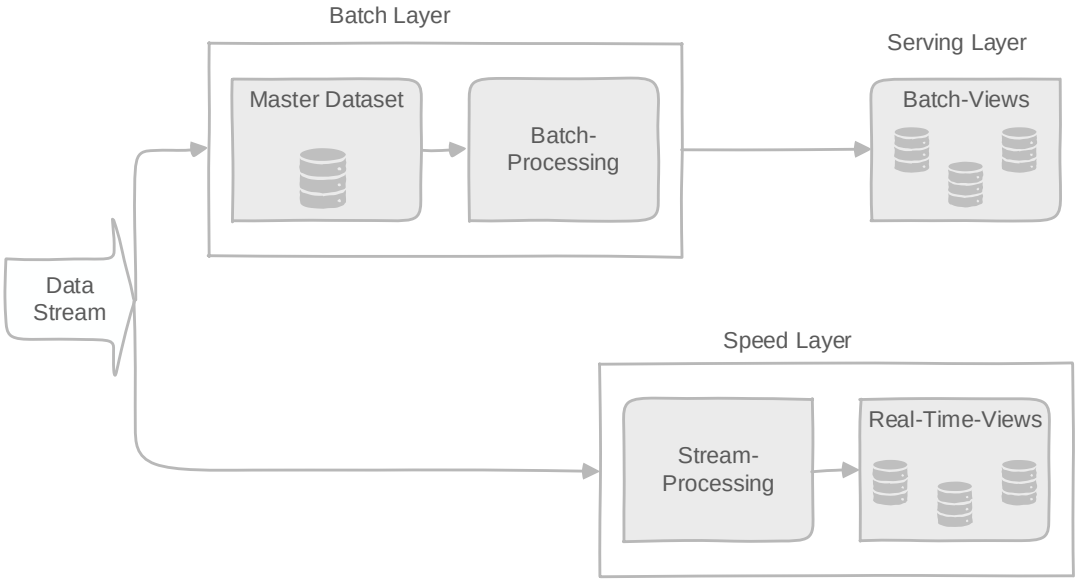

Abbildung 2.1.: Lambda-Architektur nach Warren und Marz [WM15]

in Anspruch nehmen [CY15]. Diese Art der Verarbeitung wird meist auf historischen Daten durchgeführt [GSSM18]. Dagegen wird die Stream-Verarbeitung genutzt, wenn die Daten mit geringer Latenz verarbeitet werden müssen, um ihre Aktualität zu wahren. Auch hier wird auf eine Verteilung und Parallelisierung der Verarbeitung geachtet. Die Daten werden in winzigen Stapeln direkt im Arbeitsspeicher verarbeitet. Somit wird eine geringe Latenz erreicht. Außerdem werden neu ankommende Daten sofort für die Verarbeitung berücksichtigt und Resultate sind direkt verfügbar. Die Stream-Verarbeitung wird vorwiegend für Echtzeitdaten genutzt. Diese kommen in Streams bei der Verarbeitung an [CY15]. In einem Data Lake sollten beide Verarbeitungsarten berücksichtigt werden. Daher wird in der Literatur ein Einsatz der Lambda-Architektur oder Kappa-Architektur vorgeschlagen [CY15; Mat17].

Lambda-Architektur Die Lambda-Architektur [WM15], in Abbildung 2.1 dargestellt, besteht aus zwei Zweigen. Einer dieser Zweige wird für die Batch-Verarbeitung genutzt und der andere für die Stream-Verarbeitung. Weiterhin wird in *Batch Layer, Serving Layer* und *Speed Layer* unterteilt. Ankommende Daten werden auf beide Zweige verteilt. Im Batch Layer werden die Daten an das *Master Dataset* angehängt. Das *Batch-Processing* verarbeitet diese Daten periodisch und stellt die Resultate für den Nutzer in *Batch-Views* zur Verfügung. In den Batch-Views werden nur Daten berücksichtigt, die zum Start der Verarbeitung bereits im Master Dataset gespeichert waren. Dadurch enthalten die Batch-Views nicht die aktuellsten Daten, sondern Analysen auf historischen Daten. Der Nutzer kann diese Views über den Serving Layer abfragen. Der Speed Layer ist für Echtzeitdaten zuständig. Hier werden die ankommenden Daten im *Stream-Processing* in Echtzeit verarbeitet. Die entstehenden Resultate werden in *Real-Time-Views* gespeichert. Diese Views berücksichtigen die aktuellsten Daten und werden regelmäßig überschrieben. Beide Zweige der Lambda-Architektur sind getrennt und überschneiden sich nicht [WM15].

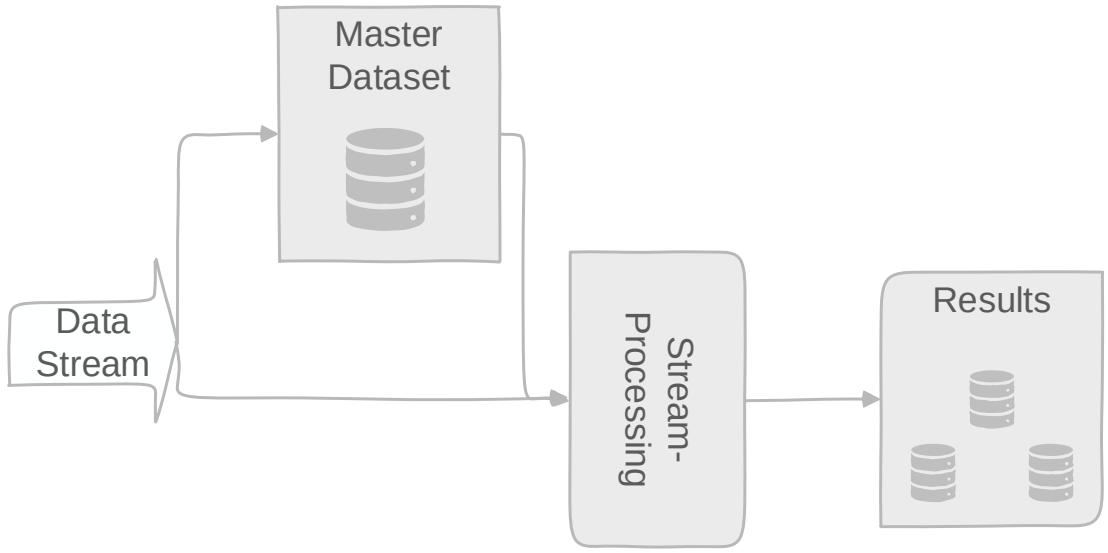

Abbildung 2.2.: Kappa-Architektur nach Kreps [Kre14]

Kappa-Architektur Im Gegensatz zur Lambda-Architektur, fokussiert sich die Kappa-Architektur [Kre14] auf Stream-Verarbeitung. Der Aufbau der Kappa-Architektur ist in Abbildung 2.2 abgebildet. Ankommende Daten werden einerseits im *Master Dataset* gespeichert und andererseits direkt verarbeitet. Das *Stream-Processing* ist so aufgebaut, dass mehrere Verarbeitungs-Jobs parallel ausgeführt werden können. Dies ermöglicht eine gleichzeitige Verarbeitung der historischen Daten des Master Datasets und des ankommenden Streams mit aktuellen Daten. Dadurch besteht weiterhin bei Bedarf die Möglichkeit eine Batch-Verarbeitung auf historischen Daten durchzuführen.

2.1.3. Zonenmodelle

Wie in Abschnitt 2.1.2 erwähnt, ist die Berücksichtigung von Zonen innerhalb der Architektur eine Alternative zur Pond-Architektur. Die Idee dahinter ist, Daten abhängig ihres Verarbeitungsgrads einer bestimmten Zone innerhalb des Data Lakes zuzuordnen. Für diesen Ansatz gibt es verschiedene Modelle, die sich jeweils in Aufteilung und Funktion der Zonen unterscheiden [GGH+19; RZ19a]. Im Folgenden werden die Zonenmodelle von Patel et al. [PWD17] und Giebler et al. [GGH+20] vorgestellt.

Patel et al. [PWD17] unterteilen den Data Lake in vier Zonen: *Transient, Raw, Trusted* und *Refined*. Die Transient Zone wird für flüchtige Daten, wie temporäre Kopien, genutzt. Rohdaten werden in der Raw Zone gespeichert. Hier sollten sensible Daten entsprechend, z.B. durch Verschlüsselung, geschützt werden. Nachdem auf den Daten in der Raw Zone Vorverarbeitungsschritte hinsichtlich der Datenqualität und Validierung ausgeführt wurden, werden diese in die Trusted Zone übertragen. Die Daten in der Trusted Zone können von Services und anderen Interessenten genutzt werden. Ergebnisse von Analysen werden in der Refined Zone gespeichert. Um Privatheit und Sicherheit zu gewährleisten, können für die jeweiligen Zonen Zugriffsbeschränkungen festgelegt werden. Diese können z.B. in der Raw Zone strenger sein als in der Refined Zone.

Giebler et al. [GGH+20] führen noch weitere Zonen ein, wodurch der Verarbeitungsgrad und die Zielgruppen der Daten mehr berücksichtigt werden. Der Data Lake wird in *Landing, Raw, Harmonized, Distelled, Explorative* und *Delivery Zone* eingeteilt. Bei der Datenannahme landen die Daten zunächst in der Landing Zone. Diese Zone wird genutzt, um eine hohe Rate bei der Datenannahme zu erreichen. Dabei werden die Daten als Batches oder Streams von der Landing Zone entgegengenommen und an die Raw Zone weitergeleitet. Wenn die Daten in der Raw Zone gespeichert sind, werden sie aus der Landing Zone entfernt. Die Raw Zone enthält die Daten in Rohform bzw. mit notwendigen Anpassungen. Innerhalb dieser Zone werden die Daten permanent gespeichert. Auf diesen Daten werden keine direkten Analysen durchgeführt. Dazu werden die Daten in die jeweilige Zone kopiert. Die Harmonized Zone enthält Kopien der Daten aus der Raw Zone. Für Analysen und weitere Verarbeitung werden die Daten dieser Zone verwendet. Auf diese Weise wird sichergestellt, dass die Daten weiterhin in der Raw Zone in ihrer Originalform vorliegen. Bei der Übertragung von Raw Zone zur Harmonized Zone werden Transformationen bezüglich Integration und Anforderungen wie Privatheit ausgeführt. Die Distelled Zone bereitet die Daten für Analysen vor, um eine höhere Effizienz zu erreichen. Dabei werden Daten z.B. für spätere Berechnungen aggregiert, aber auch komplexere Prozesse sind möglich. Die Daten werden für verschiedene Anwendungsfälle vorbereitet und können so mehrfach in verschiedener Form vorliegen. Die Explorative Zone dient als Sandbox für Data Scientists. Hier können Data Scientists Daten aus der Harmonized Zone laden, diese flexibel analysieren und erkunden. Zuletzt enthält die Delivery Zone kleine Teile der Daten, die auf bestimmte Nutzen und Anwendungen zugeschnitten sind. Diese Daten werden vor allem im operativen Gebrauch genutzt. Daher werden sie entsprechend in die jeweiligen Formate und Struktur transformiert. Eine Möglichkeit, Sicherheit und Privatheit innerhalb der Zonen zu gewährleisten, sind Zugriffsbeschränkungen und Anwendung von Datenschutzmaßnahmen. Diese können von Zone zu Zone variieren. Außerdem können für sensible Daten geschützte Zonen eingeführt werden.

2.1.4. Metadatenmanagement

Bei Data Lakes besteht die Gefahr, dass sie sich in *Data Swamps* verwandeln. Data Swamps sind Data Lakes, die über kein Metadatenmanagement und keine Kontrolle über Daten, Verarbeitungspipelines und Nutzer verfügen [Mat17]. Dadurch ist eine Verwendung der gespeicherten Daten aufgrund von fehlenden Kontrollen und Informationen über die Daten nicht optimal unterstützt [EGG+20]. Auf Dauer enthält der Data Lake eine große Menge Daten, die nicht genutzt werden können. Dies kann unter anderem die Performanz beeinflussen. Aus diesem Grund ist es wichtig, Data Lakes um ein Metadatenmanagement zu erweitern, da Metadaten die gespeicherten Daten beschreiben und somit für Analysen nutzbar machen [EGG+20]. In der Literatur lassen sich einige Modelle zum Metadatenmanagement in Data Lakes finden. Meist unterteilen diese Modelle Metadaten entweder in Kategorien oder in Inter- und Intra-Metadaten [RZ19b]. Für beide Möglichkeiten werden im Folgenden Modelle vorgestellt. Zum einen wird auf das Modell von Gröger und Hoos [GH19] eingegangen und zum anderen auf das Metadaten-Modell von Ravat und Zhao [RZ19b].

Eine Unterteilung in Kategorien wird von Gröger und Hoos [GH19] als Grundlagen genutzt. Hier wird zwischen *Technischen, Fachlichen* und *Operativen Metadaten* unterschieden. Unter die technischen Metadaten fallen alle Informationen zur technischen Struktur und den Datenhaltungen. Beispiele sind Namen von Attributen und Zugriffsrechte. Inhaltliche Informationen über die Daten werden als fachliche Metadaten kategorisiert. Diese sind z.B. konzeptionelle Datenmodelle und

Begriffsabgrenzungen. Die operativen Metadaten enthalten die technischen Details zur Datenverarbeitung, z.B. Informationen über die Quellsysteme und über angewandte Transformationen. All diese Metadaten sind Informationen über die im Data Lake hinterlegten Daten. Gröger und Hoos [GH19] fassen diese Metadaten unter *Originäre Data-Lake-Metadaten* zusammen. Darüber hinaus definieren sie weitere Kategorien für Metadaten innerhalb des Data Lakes und des dazugehörigen Systems. *Ergebnis-Metadaten* sind Metadaten zu Analyseergebnissen, wie verwendete Parameter bei Data-Mining-Prozessen. Metadaten über Geräte in einem Internet of Things (IoT)-System werden durch die *IoT-Geräte-Metadaten* dargestellt. Zuletzt enthalten die *Data-Lake-API-Metadaten* Informationen über die Application Programming Interfaces (APIs) des Data Lakes. Den Hauptteil dieses Modells bilden die originären Metadaten und die Ergebnis-Metadaten.

Ravat und Zhao [RZ19b] unterscheiden in ihrem Metadaten-Modell vor allem zwischen *Intra-* und *Inter-Metadaten*. Inter-Metadaten umfassen alle Metadaten über die Beziehungen zwischen Datensätzen. Je nach Art der Beziehung werden diese in verschiedene Sub-Kategorien unterteilt. Zum Beispiel werden Metadaten über Attribute, die sich bei Daten überschneiden, als *Content Similarity* klassifiziert. Intra-Metadaten sind Metadaten über den Datensatz. Diese werden im Modell in Folgende Sub-Kategorien unterteilt:

- *Data Characteristics:* Basisinformationen, wie Name oder Größe der Daten
- *Definitional Metadata:* Metadaten über die Semantik und Schema der Daten
- *Navigational Metadata:* Metadaten über Speicherort der Daten
- *Lineage:* Metadaten über die Herkunft und Verarbeitungshistorie der Daten
- *Access Metadata:* Zugriffsinformationen über die Daten
- *Quality Metadata:* Metadaten über die Datenqualität
- *Security Metadata:* Informationen über sensible Daten und deren Zugriffslevel

Metadaten können auf verschiedene Weisen gesammelt werden. Einerseits können Quellen direkt entsprechende Metadaten zur Verfügung stellen und anderseits können Algorithmen zur Analyse der Daten angewendet werden [GH19]. Weiterhin können Metadaten genutzt werden, die entlang der Verarbeitungspipeline von verschiedenen Bestandteilen und Technologien bereitgestellt werden.

2.2. Connected Cars

Mit dem Fortschritt der Technik in Bereichen wie Mobile und Cloud Computing werden immer mehr Fahrzeuge mit Sensorik ausgestattet und mit dem Internet verbunden. Diese Fahrzeuge werden unter dem Begriff *Connected Cars* zusammengefasst. In diesem Abschnitt werden einige Aspekte von Connected-Car-Umgebungen vorgestellt. Zu Beginn werden in Abschnitt 2.2.1 die Eigenschaften und Bestandteile eines Intelligent Transport System (ITS) erklärt. Danach folgt in Abschnitt 2.2.2 eine Definition für Connected Cars. Schließlich werden in Abschnitt 2.2.3 die Kommunikationsarten zwischen Connected Cars und dem ITS erläutert. Abschließend wird in Abschnitt 2.2.4 auf die verschiedenen Daten, die innerhalb des ITS und dem Connected Car anfallen, eingegangen.

2.2.1. Intelligent Transportation System

In einem Intelligent Transport System (ITS) [ML13] werden Verkehrsteilnehmer effektiv integriert. Dadurch soll erreicht werden, dass der Verkehr für Fußgänger oder Radfahrer sicherer ist und Kraftstoff besser eingesetzt wird. Um dies zu erreichen, sind die Teilnehmer und Bestandteile des Systems mit Sensoren ausgestattet und können auf verschiedene Weisen untereinander kommunizieren. Hauptteilnehmer eines ITS sind Fahrzeuge, die mit Onboard Units (OBUs) und Sensorik ausgestattet sind [SLY+16]. Diese Fahrzeuge werden auch Connected Cars genannt. Weitere wichtige Bestandteile eines ITS sind Road Side Units (RSUs), Vulnerable Road Users (VRUs) und ITS-Server. RSUs sind Komponenten innerhalb der Infrastruktur. Einerseits können RSUs eigene Services zur Verfügung stellen. Andererseits können sie durch ihre Sensoren weitere Informationen zur Verkehrssituation im System beitragen [SLY+16]. Zum Beispiel sind RSUs in Ampeln, Laternen oder Straßenschildern verbaut [CM16]. Unter VRUs fallen beispielsweise Fußgänger, Radfahrer und Motorradfahrer. Diese Gruppen haben meist nicht den gleichen technologischen Vorteil wie Connected Cars. Zwar können VRUs z.B. über Smartphones am System teilnehmen, jedoch tragen sie dies nicht durchgehend bei sich. Daher muss auf die Sicherheit im Verkehr von VRUs zusätzlich durch Sensoren in der restlichen Infrastruktur geachtet werden [PKÅ+20]. ITS-Server sind die zentrale Steuerung. Hier werden Informationen über den Verkehr und die verfügbaren Services verwaltet [SLY+16]. Typische Services in einem ITS sind z.B. verschiedene Arten der Verkehrsüberwachung, Navigation und Kraftstoffoptimierung [CM16].

2.2.2. Definition und Autonomielevel von Connected Cars

Coppola und Morisio [CM16] betrachten einige Definitionen und stellen Charakteristiken eines Connected Cars vor. Dabei ist die wichtigste Eigenschaft die Verbindung zum Internet. Diese kann entweder durch eingebaute Hardware oder über das Smartphone der Fahrer bestehen. Weiterhin ist ein Connected Car mit Sensoren ausgestattet, um seine Umgebung wahrzunehmen und darauf zu reagieren. Zusätzlich werden Daten aus eingebauten Electronic Control Units (ECUs) und OBUs genutzt. Dies ermöglicht unter anderem die Bereitstellung von modernen Services, Infotainment und die Kommunikation mit der Infrastruktur des ITS. Darüber hinaus können Connected Cars untereinander kommunizieren.

Da beim autonomen Fahren der Fahrer durch das Fahrzeug beim Fahren unterstützt wird, hängt dieser Bereich eng mit den Connected Cars zusammen. Hierbei wird meist an selbstfahrende Fahrzeuge gedacht. Jedoch gibt es verschiedene Abstufungen von Autonomie [CM16; PKÅ+20]. Die *Society of Automoitve Engineers (SAE)* definiert in Standard SAE J3016 [Int21] sechs Autonomielevel. Diese Level lauten wie folgt:

Level 0 *Keine Autonomie:* Der Fahrer kontrolliert vollständig das Fahrverhalten.

Level 1 *Fahrassistenz:* Fahrassistenzsysteme, wie Spurhalteassistent, werden eingesetzt. Der Fahrer lenkt weiterhin das Fahrzeug.

Level 2 *Teilweise Autonomie:* Das Fahrzeug kann autonom fahren. Jedoch muss der Fahrer das Verhalten überwachen und in Gefahrensituationen eingreifen.

Level 3 *Bedingte Autonomie:* Das Fahrzeug überwacht seine Umgebung und Funktionalität. Der Fahrer wird vom Fahrzeug über nötiges Eingreifen informiert.

Level 4 *Hohe Autonomie:* Das Fahrzeug kann Gefahrensituationen selbstständig bewältigen, wenn der Fahrer nicht auf Anfragen reagiert.

Level 5 *Volle Autonomie:* Der Fahrer muss nicht mehr im Fahrzeug anwesend sein. Das Fahrzeug trifft alle Fahrentscheidung, auch in unerwarteten Situationen.

Heutzutage bewegen sich die meisten zugelassenen Fahrzeuge zwischen Level 0 und 2. Einige Hersteller arbeiten an Prototypen mit höherem Autonomielevel [PKÅ+20].

2.2.3. Kommunikation

Innerhalb eines ITS kommunizieren Connected Cars auf unterschiedliche Weisen. Diese Kommunikationsarten umfassen nicht nur die Kommunikation zwischen den Fahrzeugen, sondern auch die Kommunikation mit anderen Bestandteilen des Systems. Dabei lassen sich die Kommunikationsarten folgendermaßen einteilen [CM16; LCZ+14]:

Vehicle To Sensors (V2S) ist die Kommunikation zwischen den ECUs und den Sensoren innerhalb des Fahrzeugs. Der Austausch kann über kabelgebundene oder kabellose Netzwerke erfolgen.

Vehicle To Vehicle (V2V) bezeichnet die Kommunikation zwischen den Fahrzeugen. Diese Kommunikation wird z.B. zur Unfallvermeidung und Streckenoptimierung genutzt. Fahrzeuge bilden dabei ein *Vehicular Ad-Hoc Network (VANET)*. VANETs sind Ad-Hoc Netzwerke, in denen Connected Cars und RSUs ohne zusätzliche Infrastruktur untereinander Netzwerke bilden [KMC+13].

Vehicle To Road Infrastructure (V2R) ist die Kommunikation zwischen dem Fahrzeug und der Straßeninfrastruktur, wie Ampeln, RSUs und Straßenschildern. Diese Kommunikation ist vor allem für ein effizientes Management des Verkehrs wichtig.

Vehicle To Internet (V2I) ist die Verbindung des Connected Cars zum Internet, was die Haupteigenschaft eines Connected Cars darstellt. Diese Verbindung ist essentiell, um Services und Infotainment nutzen zu können.

Im restlichen Verlauf der Arbeit wird sich hauptsächlich mit der V2I-Kommunikation beschäftigt. Der Data Lake bildet einen Service zur Datenverwaltung im System. Daher wird die V2I-Kommunikation genutzt, um Daten vom Connected Car zum Data Lake zu übertragen.

2.2.4. Daten

In Connected-Car-Umgebungen werden viele Daten gesammelt. Allein ein Connected Car produziert schon heute bis zu 25GB pro Stunde [Wol16]. Hinzu kommen Daten der RSUs, der VRUs und von internen und externen Services. In diesem Abschnitt werden die Daten genauer betrachtet, die in einem ITS anfallen. Ein Connected Car ist mit vielen Sensoren ausgestattet. Von den dabei entstehenden Sensordaten wird jeweils nur ein kleiner Teil für Services, wie Verkehrsvorhersagen, Diagnostik und Infotainment, genutzt. Außerdem sind diese Daten nur für bestimmte Zeiträume aktuell. Vor allem Vorhersagen und Management des Verkehrs sind Services bei denen Sensordaten von mehreren Fahrzeugen, RSUs und Smartphones zusammengeführt werden [LKM+15]. Sensordaten sind Streams von geordneten Events [GZ17a]. Ein Sensordatum besteht meist aus einem Wert und einem Zeitstempel. Durch diesen Zeitstempel können Sensordaten in eine Reihenfolge gebracht

werden. Somit können sie zu den Zeitreihendaten gezählt werden. Darüber hinaus werden in einem Connected Car und dem dazugehörigen ITS noch weitere Datenarten gesammelt. Ein Beispiel sind unstrukturierte Daten in Form von Bildern und Videos von Kamera-basierten Sensoren [LKM+15]. Diese verschiedene Datenarten und die damit einhergehenden Anforderungen müssen bei der Verarbeitung und der Speicherung beachtet werden.

Ein besonderes Augenmerk sollte auf die personenbezogenen Daten gelegt werden. Connected Cars sammeln nicht nur Daten über das Fahrzeug selbst, sondern auch über Fahrer und Passagiere. Typische Beispiele sind Positionsdaten, Fahrverhalten, Präferenzen im Infotainment-Bereich und Anzahl der Fahrzeuginsassen. Diese Beispiele entsprechen nur einem Bruchteil der gesammelten personenbezogenen Daten [LML15]. Durch diese Daten lassen sich Rückschlüsse auf Personen ziehen, daher ist ein Schutz bei der Speicherung und Verarbeitung unerlässlich [NFM16].

3. Verwandte Arbeiten

In diesem Kapitel werden einige Systeme aus der Literatur, die Verkehrsdaten mit Hilfe von Big-Data-Technologien verarbeiten, betrachtet. Zusätzlich wird ein Konzept mit verteilten Data Lakes vorgestellt.

Nkenyereye und Jang [NJ15] erstellen ein Hadoop-basiertes System für die Verarbeitung von Fahrzeugdaten. Die Grundidee ist Diagnostikdaten der Fahrzeuge in ein Datencenter zu laden. Diese Daten sollen mit Hadoop verarbeitet und die Ergebnisse auf einem Webserver gespeichert werden. Zuerst werden die Diagnostikdaten aus einer MySQL-Datenbank ins HDFS der Hadoop-Cluster importiert. HDFS ist das Dateisystem von Hadoop zur Speicherung großer Datenmengen. Danach werden die Daten mit dem MapReduce-Ansatz von Hadoop verarbeitet. Die Ergebnisse der Verarbeitung werden im CSV-Format vom HDFS zum Webserver übertragen. Durch das vorgestellte System wird gezeigt, dass die Anwendung von Big-Data-Technologien auf Connected Cars möglich ist. Jedoch wird eine Batch-Verarbeitung basierend auf Hadoop eingesetzt. Die Verwendung einer Batch-Verarbeitung in Connected-Car-Umgebungen kann wertvolle Antwortzeit kosten und eine Reaktion auf Ereignisse wäre nicht in Echtzeit möglich. Eine Stream-Verarbeitung ist hier schneller und somit besser geeignet [MRS+15]. Weiterhin werden die Diagnostikdaten zunächst in eine MySQL-Datenbank gespeichert, bevor sie von Hadoop importiert werden. Somit müssen die Daten vor Speicherung in ein bestimmtes Schema transformiert werden, was dem Schema-on-Read-Ansatz von Data Lakes widerspricht. Außerdem werden keine heterogenen Datenquellen berücksichtigt, da keine Verarbeitungsmöglichkeit für unstrukturierte Daten, wie Bilddaten, besteht.

Auch Guerreiro et al. [GFS+16] verdeutlichen den Nutzen von Big-Data-Technologien innerhalb eines ITS. Dabei stellen sie eine Architektur für ETL-Prozesse vor. In einem Beispielszenario soll dynamisch die Maut auf Highways abgebildet werden. Dazu sollen Echtzeitdaten und historische Daten aus heterogenen Quellen verwendet werden. Ziel ist, diese Daten effizient zu sammeln, zu transformieren und zu speichern. Die Verarbeitung erfolgt mit Apache Spark. Außerdem wird für die Speicherung MongoDB genutzt. Die finale Architektur wird mit einem typischen ETL-Ansatz ohne Berücksichtigung von Big-Data-Technologien verglichen. Die Autoren kommen zu dem Schluss, dass durch den Einsatz von Big-Data-Technologien ein deutlicher Anstieg in der Performanz zu erkennen ist.

Grulich und Zukunft [GZ17a; GZ17b] entwickeln das System NASCIS.. Bei NASCIS liegt der Fokus vor allem in der skalierbaren und fehlertoleranten Verarbeitung von Stream-Daten aus heterogenen Quellen innerhalb des ITS. Das Konzept beruht auf dem Streaming der Daten aller Connected Cars zu einer zentralen Serverkomponente. In der Serverkomponente werden die Daten aus verschiedenen, auch externen, Quellen verrechnet und an interessierte Parteien gesendet. Da vorrangig Stream-Daten verarbeitet werden sollen, wird eine Variante der Kappa-Architektur verwendet. Diese nutzt für die Verarbeitungskomponente Apache Spark. Die Rohdaten und Analyseergebnisse werden in einer CouchDB und MySQL-Datenbank gespeichert. Auf einem Prototypen werden Tests hinsichtlich der Skalierbarkeit und Fehlertoleranz durchgeführt. Die Entwickler kommen zu dem Ergebnis, dass

beide Aspekte in allen Testfällen zu Genüge erfüllt werden. Somit ist NASCIS ein Beispiel für einen skalierbaren und fehlertoleranten Data Lake, der vorwiegend auf Stream-Daten spezialisiert ist. Jedoch könnte die Anwendung eines zentralen Data Lakes in einem ITS zu Latenzeinbußen führen. Wenn sich z.B. ein Connected Car am Rand des ITS befindet, kann der Übertragungsweg zum Data Lake und zurück weit sein. Diese Verzögerung durch den Übertragungsweg kann mitunter dazu führen, dass in einer Gefahrensituation eine Antwort zu spät ankommt. Daher sollte auf eine Verarbeitung der Daten nah am Connected Car geachtet werden.

Munshi und Mohamed [MM18] wenden erfolgreich eine Lambda-Architektur im Smart-Grids-Kontext an. In einem Smart-Grid fallen viele unterschiedliche Daten an. Diese umfassen unter anderem Video-, Bild- und Sensordaten. Um die steigende Menge an Daten zu analysieren, reichen traditionelle Datenmangementsysteme im Smart-Grids-Kontext nicht mehr aus. Aus diesem Grund konzipieren die Autoren einen Data Lake basierend auf der Lambda-Architektur. Für die Speicherung und Verarbeitung der Daten wird das Hadoop-Ecosystem genutzt. Somit können Daten aus heterogenen Quellen verarbeitet und die Vorteile von Stream- und Batch-Verarbeitung genutzt werden. Die Datenarten in einem Smart-Grid und ITS sind ähnlich, wodurch sich die Lambda-Architektur auch auf eine Connected-Car-Umgebung anwenden lassen würde. Allerdings muss wieder die Länge der Übertragungswege beachtet werden. Weiterhin werden alle Daten im HDFS gespeichert. An dieser Stelle könnte die Performanz durch Nutzung mehrerer Datenbanken, die auf die anfallenden Datenarten spezialisiert sind, erhöht werden.

Theodorou und Diamantopoulos [TD19] führen in ihrer Arbeit den Begriff *Data Lagoon* ein. Die Grundidee ist, Edge Computing auf das Konzept des Data Lakes zu übertragen. Im Edge Computing werden Berechnungen und Speicherung von Daten aus der Cloud näher an die Quellen ausgelagert. Data Lagoons sind leichtgewichtige Repositories und werden auf Edge Nodes gehostet. Ihre Aufgabe besteht aus dem Sammeln der Daten aller Geräte, die mit diesem Edge Node verbunden sind. Auch Teile der Verarbeitung kann auf den Edge Node ausgelagert werden. Dadurch werden einerseits vorhandene Ressourcen besser genutzt und andererseits wird schneller auf auftretende Events reagiert. Weiterhin kann eine Unabhängigkeit zwischen Datensammlung und -verarbeitung geschaffen werden. Im Data Lagoon werden unverarbeitete Daten gelagert. Der Data Lagoon stellt diese über Daten-Topics anderen Komponenten zur Verarbeitung bereit. Zusätzlich haben Theodorou et al. [THQ19] in einer weiteren Arbeit ein Metadaten-Framework für die Data Lagoon erstellt. Mit diesem wird die Interaktion zwischen der Data Lagoon und einem zentralen Data Lake genauer beschrieben. Dabei wird die Data Lagoon vom Data Lake als Service für Daten genutzt. Die Verteilung des Data Lakes im System ist eine Möglichkeit, Übertragungswege zu verringern. Allerdings wurde das System nicht im Kontext eines ITS oder einem ähnlichen Systems entwickelt. Daher wird im restlichen Verlauf der Arbeit die Grundidee eines verteilten Data Lakes von Theodorou und Diamantopoulos [TD19] auf eine Connected-Car-Umgebung übertragen.

4. Anforderungen an Data Lakes in Connected-Car-Umgebungen

Ein Data Lake in einer Connected-Car-Umgebung muss einige Anforderungen erfüllen. Diese Anforderungen kommen einerseits aus dem Bereich des Data Lakes mit Big Data und andererseits aus besonderen Herausforderungen im Kontext der Connected-Car-Umgebung. Gerade hier ist z.B. eine hohe Performanz zur Verarbeitung in Echtzeit und die Beachtung der Privatheit bei der Speicherung und Verarbeitung von persönlichen Daten nötig. Daher wird in Abschnitt 4.1 auf diese und weitere Herausforderungen von Big Data in Connected-Car-Umgebungen eingegangen. Anschließend werden in Abschnitt 4.2 Anforderungen definiert, die ein Data Lake in diesem Bereich erfüllen sollte.

4.1. Herausforderungen an Big Data in Connected-Car-Umgebungen

Fahrzeuge produzieren immer mehr Daten. Sei es durch eingebaute Navigationsgeräte oder durch Software, die die Vielzahl an ECUs innerhalb des Fahrzeugs steuern. Mit der immer weiter zunehmenden Verbundenheit zum Internet und somit der Möglichkeit mehr Services zu nutzen oder zu entwickeln, werden Fahrzeuge immer mehr Daten produzieren. Wollschläger et al. [Wol16] sagt bis zu 25 GB pro Stunde für Mittelklassewagen voraus. Hier kommen Daten aus externen Datenquellen, z.B. Wetterdienste, hinzu. Dadurch müssen extrem große Datenmengen verwaltet werden. Gerade diese Herausforderung qualifiziert Daten, die in einem ITS anfallen, als Big Data, das sich über die 3 Vs: Volume (Menge), Velocity (Geschwindigkeit der Verarbeitung), Variety (Vielfalt) definiert [CY15]. Auch im Bezug auf Connected Cars entstehen einige Herausforderungen für Big Data. Auf einige dieser Punkte, die Big Data in Connected-Car-Umgebungen ausmachen, wird im Folgenden eingegangen.

4.1.1. Echtzeitverarbeitung

Eine Herausforderung für das Big-Data-Management in einer Connected-Car-Umgebung ist die Echtzeitverarbeitung. Viele der gesammelten Sensordaten benötigen eine schnelle Reaktion und sind zum Teil nur für einen sehr kleinen Zeitraum relevant [MRS+15]. Ein Beispiel hierfür ist eine Hinderniserkennung. Wenn ein Hindernis auf der Fahrbahn auftaucht, sollte ein Fahrzeug direkt reagieren und keine wertvolle Zeit für lange Berechnungen aufwenden. Besonders im Verkehrsbereich, kann der Zeitaufwand für Berechnungen und Netzwerkübertragungen über Leben und Tod entscheiden. Die Echtzeitdaten des Connected Cars sind dabei ein Stream von geordneten Events [GZ17a]. Außerdem sind die gesammelten Daten zum Teil kurz nach Auftreten des Events oftmals nur noch für historische Auswertungen relevant [PKÅ+20]. Das bedeutet, wenn die Rückmeldung der Verarbeitung kommt, dass sich vor dem Fahrzeug ein Hindernis befindet, das

Fahrzeug aber schon längst daran vorbei gefahren ist, ist diese Information nicht mehr aktuell. Je nach Autonomitätslevel kann eine verspätete Antwort genauso gefährlich sein wie keine Antwort oder eine kontaminierte Antwort. Zum Beispiel wenn grundlos ein Brems- oder Ausweichmanöver eingeleitet wird und somit andere Verkehrsteilnehmer gefährdet werden [CAYM15; PKÅ+20]. Daher sollte ein Big-Data-Management-System in einer Connected-Car-Umgebung eine Möglichkeit zur Verarbeitung von Echtzeitdaten bieten.

4.1.2. Verschiedene Arten und Nutzer der Datenauswertung

Eine weitere Herausforderung ist das Interesse an den gesammelten Daten von vielen verschiedenen Stakeholdern. Zu diesen Stakeholdern zählen unter anderem Hersteller, Verkehrsüberwachung, verschiedene Services, wie Infotainment und Navigation, und die Fahrer selbst. All diese Interessensgruppen benötigen unterschiedliche Daten in unterschiedlichen Formaten und Aggregation [LKM+15; PKÅ+20]. Das bedeutet, es gibt nicht nur viele Nutzer möglicher Datenauswertungen, sondern zusätzlich unterschiedliche Arten der Datenauswertung. Beispielsweise können Auswertungen auf dem Datenstream selbst, aber auch Analysen auf historischen Werten, ausgeführt werden [CY15; PKÅ+20]. Somit ist die Bereitstellung einer Möglichkeit zur Datenauswertung bezogen auf die Interessen von verschiedenen Stakeholdern wichtig. Weiterhin sollte beachtet werden, welche Interessensgruppe auf welche Daten Zugriff in Hinblick auf Sicherheit und Privatheit haben [PKÅ+20]. Gerade für die Verarbeitung von personenbezogenen Daten gibt es meist Vorgaben der Gesetzgeber. In Deutschland ist dies die Datenschutz-Grundverordnung (DSGVO) [Eur16].

4.1.3. Unterschiedliche Formate und Datenstrukturen

Nicht nur explorative Datenauswertungen müssen in unterschiedliche Formate transformiert werden können, auch Datenquellen können in unterschiedlichen Formaten vorliegen. Zum einen gibt es im Automobilbereich keine einheitlichen Standards für Connected Cars, wodurch jeder Hersteller unterschiedliche bzw. eigene Formate für Daten verwendet. Hier wird vom sogenannten Vendor Lock-In gesprochen. Vendor Lock-In bedeutet, dass ein Kunde an einen Hersteller und dessen Services gebunden ist und nicht Services eines anderen Anbieter nutzen bzw. den Anbieter nicht vollständig wechseln kann. Diese Herstellerbindung wird meist durch eigene Datenformate erreicht. Zum anderen werden in einem ITS nicht nur Daten von Fahrzeugen gesammelt, sondern auch von RSUs, externen Services, Smartphones von anderen Verkehrsteilnehmenden usw. Hier liegen ebenfalls Daten in vielen unterschiedlichen Formaten vor [CM16; LKM+15]. Außerdem unterscheiden sich die gesammelten Daten in ihrer Struktur. So können je nach Definition strukturierte Daten, semi-strukturierte Daten und unstrukturierte Daten auftreten [LKM+15]. Jede dieser Datenstrukturen und Formate besitzt andere Anforderungen an eine Datenhaltung, weshalb dies bei der Bereitstellung eines Big-Data-Managements in einer Connected-Car-Umgebung berücksichtigt werden sollte.

4.1.4. Mehrbenutzerzugriffe

In einem ITS sind viele Akteure gleichzeitig mit dem System bzw. untereinander verbunden. Vor allem die im System befindlichen Connected Cars greifen oft auf gleiche Services zu. Dabei kann vorkommen, dass mehrere Connected Cars gleichzeitig auf bestimmte Daten oder gar Datensätze

zugreifen wollen [PKÅ+20]. Ein Beispiel ist die Suche nach einem freien Parkplatz. Wenn mehrere Connected Cars in der Umgebung zeitgleich nach einem Parkplatz suchen, kann passieren, dass ihnen derselbe freie Parkplatz angezeigt wird. Wollen zwei Connected Cars zur selben Zeit diesen Parkplatz für sich beanspruchen, greifen beide auf den Datensatz, der die Belegung des Parkplatzes enthält, zu und wollen diesen entsprechend ändern.

Lesend sind solche Mehrbenutzerzugriffe in der Regel kein Problem. Sobald jedoch eines der Fahrzeuge die Daten ändern möchte, z.B. um die Belegung des Parkplatzes zu signalisieren, kann dies zu Anomalien führen. Beispiele hierfür sind unter anderem Dirty Read und Lost Update. Diese Anomalien der Datenhaltung können auftreten, sobald einer der Akteure schreibend auf Daten innerhalb der Datenhaltung zugreifen will [BBG+95]. Je nach Daten können diese problematisch werden. Aus diesem Grund sollte die im Big-Data-Management genutzte Datenhaltung gegen diese Anomalien durch Mehrbenutzerzugriffe abgesichert sein.

4.1.5. Unkontrollierbare und unvorhersehbare Teilnehmer

In einem ITS gibt es viele verschiedene Akteure, unter anderem Fußgänger, Fahrradfahrer, RSUs und Fahrzeuge. Unter diesen Akteuren ist viel Bewegung. Fahrzeuge und Personen, die über das Smartphone mit dem System verbunden sind, bewegen sich einerseits innerhalb des Systems, wodurch sich z.B. Übertragungswege oder die nächstgelegene RSU ändern, und andererseits aus dem System hinaus oder hinein [PKÅ+20]. Daher sollten Akteure leicht dem System beitreten können. Gleichzeitig muss die Möglichkeit bestehen, sich dynamisch mit nächstgelegenen RSUs oder anderen Servicequellen zu verbinden. Außerdem sollte berücksichtigt werden, dass z.B. eine RSU die Datenanfrage eines Connected Cars annimmt und eine andere RSU die Antwort an das Fahrzeug zurückgibt. Ähnlich ist es bei der V2V-Kommunikation (vgl. Abschnitt 2.2). Hier kommunizieren Connected Cars untereinander, was nur während dem aneinander Vorbeifahren vorkommen kann [CM16].

Diese Herausforderungen sollten bei der Auswahl der richtigen Management-Umgebung für Big Data in einer Connected-Car-Umgebung beachtet werden. Ein Data Lake eignet sich, da er unter anderem durch seinem Schema-on-Read-Ansatz mit der Heterogenität der Daten umgehen kann und in bestimmten Formen eine Echtzeitverarbeitung der Daten erreicht. Aus den aufgezählten Herausforderungen ergeben sich für einen Data Lake in einer Connected-Car-Umgebung bestimmte Anforderungen, die erfüllt werden sollten. Diese werden im nächsten Abschnitt genauer definiert.

4.2. Anforderungen an einen Data Lake

Um die Big-Data-Herausforderungen in einer Connected-Car-Umgebung zu lösen, könnte ein Data Lake genutzt werden. Wie in Abschnitt 2.1 beschrieben, eignet sich ein Data Lake bei heterogenen Datenquellen und verschiedenen Datenstrukturen. In Connected-Car-Umgebungen werden einerseits Daten aus dem Fahrzeug selbst, vor allem Sensordaten, und andererseits Daten aus externen Quellen, wie Wetterdienste und zusätzliche Verkehrsdaten, gesammelt. Daher werden nicht nur Daten aus verschiedenen Quellen verarbeitet, sondern diese liegen auch in unstrukturierter, semi-strukturierter und strukturierter Form vor [LKM+15]. Ein Data Lake ermöglicht durch sein

unterliegendes Konzept (vgl. Abschnitt 2.1) eine effektive Verarbeitung dieser Daten. Jedoch sollten einige Anforderungen bei der Konzeptionierung eines Data Lakes in diesem Bereich beachtet werden, da sich ein ITS grundlegend von einem traditionellen Informationssystem unterscheidet.

Pelliccione et al. [PKÅ+20] erwähnen folgende Anforderungen:

- Sicherheit
- Privatheit
- Hohe Performanz und niedrige Latenz
- Metadatenmanagement
- Bereitstellung von Schnittstellen
- Unterstützung von Heterogenität

Im diesem Abschnitt werden die aufgelisteten Anforderungen genauer erklärt, um in Kapitel 5 ein Konzept für einen Data Lake in einer Connected-Car-Umgebung ableiten zu können.

4.2.1. Sicherheit

Sicherheit spielt in Connected-Car-Umgebungen eine große Rolle. Gerade durch die Verbundenheit zum Internet ist das Connected Car kein geschlossenes System, was laut Chen et al. [CCFC15] für neue Sicherheitsgefahren, wie z.B. Hackerangriffe von außen oder dem Durchsickern von Informationen, sorgt. Sicherheit lässt sich in Security und Safety unterteilen, welche leicht unterschiedliche Schwerpunkte haben. Security bedeutet, dem System kann von außen keinen Schaden zugefügt werden [LNRT06]. Zum Beispiel ist es einem Hacker nicht möglich auf ein System zuzugreifen und Daten auszulesen. Dagegen bezieht sich Safety darauf, dass das System keinem Nutzenden oder Dritten schadet [LNRT06]. Ein Beispiel wäre, wenn ein Connected Car ein Bremsmanöver einleitet, obwohl hierfür kein Grund besteht und somit ggf. andere Teilnehmer im Straßenverkehr gefährdet werden [CAYM15]. Mit dem Ziel einen Standard für Safety im Connected Cars Bereich zu erreichen, wurde 2011 die ISO 26262 veröffentlicht. Kernpunkte dieses Standards sind die Automotive Safety Integrity Levels (ASILs), die eine Klassifizierung von verschiedenen Saftey-Risiken und ihre Konsequenzen enthalten [ISO18]. Vor allem in Connected-Car-Umgebungen ist Security ein wichtiger Einfluss auf Safety. Wenn es einem Hacker durch unzureichende Security-Maßnahmen gelingt, Zugriff auf das Fahrzeug zu erhalten, kann dieser das Fahrzeug entsprechend manipulieren, sodass Safety nicht mehr gewährleistet wäre [CAYM15; CCFC15]. Zusätzlich tragen unter anderem Security-Maßnamen zu dem Schutz von privaten Daten bei. Ansonsten könnten Unbefugte an Informationen über verschiedene Akteure des Systems gelangen. Diese Daten könnten an Dritte weitergegeben oder genutzt werden, um weitere Attacken zu planen [CAYM15].

Im Weiteren wird sich vor allem auf die Sicherheit, welche die im Data Lake gespeicherten Daten vor unbefugten Zugriffen schützt, bezogen. Sicherheitsgefährdungen können in einem ITS an drei Punkten auftreten: im Fahrzeug selbst, bei der Kommunikation und in der Cloud [BAG15]. Chen et al. [CCFC15] und Carten et al. [CAYM15] gehen auf die möglichen Angriffe auf ein Connected Car genauer ein. Ein Hauptpunkt ist die Sicherheit des CAN-Bus, der die Kommunikation zwischen

ECUs ermöglicht. Gleichzeitig sollten sich mögliche Sicherheitsvorkehrungen nicht negativ auf Latenz oder Safety auswirken [PKÅ+20]. Deshalb sollten z.B. geeignete Zugriffsbeschränkungen und Verschlüsselungsverfahren genutzt werden, um den Data Lake zu schützen.

4.2.2. Privatheit

Connected Cars, sowie das ganze umliegende System, sammeln viele persönliche Daten. Dabei werden nicht nur Daten über den Standort gesammelt, sondern auch über Geschwindigkeiten, Fahrzeiten usw. [LML15]. Dies ermöglicht auf viele Aspekte des Lebens der Nutzer des Connected Cars zu schließen [NFM16]. Beispielsweise könnte nur anhand des Standortes und Nutzungszeiten darauf geschlossen werden, wo genau eine Person wohnt oder arbeitet. Weiterhin wäre eine umfangreiche Analyse des Fahrverhaltens durch die Daten der Sensoren möglich [LML15]. In den falschen Händen können diese Informationen schnell ausgenutzt werden, z.B. für personalisierte Werbung oder durch Angriffe für kriminelle Zwecke. Meist wissen Nutzer nicht welche Daten über sie gesammelt und weitergegeben werden, bzw. ist es nur sehr schwer einsehbar, oder diese Daten werden Service-intern mit Datensätzen von anderen Services verarbeitet. Außerdem möchten viele Personen ungern persönliche Daten an Dritte weitergeben, wenn diese Weitergabe keinen Nutzen für die betroffene Person mit sich bringt [Cot09; NFM16]. Eine Missachtung der Privatheit von personenbezogenen Daten kann dazu führen, dass Nutzer einen Service nicht nutzen und eine Abschaltung des Services verlangen [Cot09]. Daher ist es wichtig, persönliche Daten, die im Fahrzeug oder von anderen Akteuren des ITS gesammelt werden, gut zu schützen. Zum einen sollten auf die Daten nur Organisationen Zugriff haben, bei denen der Nutzer der Verarbeitung zugestimmt hat. Zum anderen müssen geltende Verordnungen, wie die DSGVO, in Bezug auf die Verarbeitung personenbezogener Daten beachtet werden. Des Weiteren sollten private Daten durch Verschleierungsmechanismen und Anonymisierung geschützt werden [Sta19].

Da in einem ITS enorme Mengen personenbezogener Daten über das Internet hochgeladen werden, sollte der Nutzer volle Kontrolle darüber haben, welche Daten verarbeitet werden dürfen. Aus diesem Grund sollte der Data Lake einige Prinzipen bezüglich Privatheit und Sicherheit erfüllen [CM16]. Als erstes sollte kommuniziert werden, welche Daten erhoben werden und für was diese genutzt werden sollen, also Transparenz geschaffen werden. Genauso soll eine Einwilligung des Nutzers eingeholt werden bevor Daten mit Dritten geteilt oder weitergeleitet werden. Hier sollten nur die Daten geteilt werden, die für den Kontext von Nutzen sind. Insgesamt sollte die Menge an personenbezogenen Daten klein gehalten und gelöscht werden, sobald sie nicht mehr benötigt werden. Wie in Abschnitt 4.2.1 bereits erwähnt, sollten personenbezogene Daten vor Hackerangriffen geschützt sein [CM16]. In einem Data Lake kann dies z.B. durch Zonen erreicht werden. So könnten Zonen mit verschiedenen Privatheitsstufen eingerichtet werden. In jeder Zone liegen die Daten unterschiedlich stark anonymisiert und verschleiert vor. Nur Interessengruppen mit der jeweiligen Freigabe für diese Zone können auf die Daten dieser Zone zugreifen. Ein ähnliches Prinzip wurde von Stach et al. [SGW+20] angewendet, um Machine Learning mit Daten je nach spezifischen Privatheitsanforderungen zu versorgen und somit die DSGVO einzuhalten.

4.2.3. Hohe Performanz und niedrige Latenz

Wie in Abschnitt 4.1.1 erwähnt, handelt es sich bei vielen Daten in Connected-Car-Umgebungen um Echtzeitdaten, die oft nur während eines bestimmten Ereignisses relevant sind. Auf diese Ereignisse muss in der Regel sofort reagiert werden, z.B. ein Bremsmanöver bei einem Hindernis auf der Fahrbahn. Verspätet sich in diesem Szenario eine Antwort, kann dies fatale Folgen mit sich bringen. Die Sensoren eines Connected Cars senden unter anderem Zeitreihendaten, die als ein Stream von einzelnen Events betrachtet werden können. Diese werden als Stream-Daten bezeichnet [GZ17a]. Die klassische Hadoop-Umgebung, die oft in Data-Lake-Implementierungen genutzt wird, arbeitet mit einer Batch-Verarbeitung. Hierbei werden die Daten, die in der Datenhaltung gespeichert sind, in einzelnen Stapeln verarbeitet. Neu ankommende Daten werden nicht berücksichtigt, sobald die Verarbeitung eines Stapels begonnen hat [CY15]. Diese Art der Verarbeitung dauert nicht nur länger, sondern kann auch zu Rückstau der Daten führen, was die Performanz wiederum zusätzlich beeinträchtigen und im Straßenverkehr gefährliche Konsequenzen haben kann [GZ17a]. Dagegen arbeitet Stream-Verarbeitung mit kleinen Datenstapeln direkt In-Memory, wodurch neu ankommende Daten mit in die Verarbeitung fließen. Durch die Arbeit In-Memory statt auf der Festplatte wird die Latenz erheblich gesenkt [CY15].

Um Streamdaten effektiver verarbeiten zu können, wurden Lambda- und Kappa-Architektur entwickelt. Die Lambda-Architektur besitzt zwei Zweige, einen für Batch-Verarbeitung und einen für Stream-Verarbeitung. Die Resultate werden jeweils in Views gespeichert. Die Kappa-Architektur nutzt dagegen nur eine Stream-Verarbeitung, aber speichert parallel die Daten persistent. In beiden Fällen werden gleichzeitig die Daten aus den Streams für historische Auswertungszwecke in Batch-Views gespeichert. Außerdem bieten beide Architekturen Möglichkeiten zusätzlich Batchdaten, wie z.B. Wetterdaten, zu verarbeiten und entsprechend zu speichern (vgl. Abschnitt 2.1). Beide Architekturen verarbeiten somit Datenstreams aus dem Connected Car effizient. Deshalb sollten sie als Grundlage für die Verarbeitung in eines Data Lakes in Connected-Car-Umgebungen genutzt werden. Zum Beispiel entwickelten Grulich et al. [GZ17b] ein Data Lake basierend auf der Kappa-Architektur mit Verarbeitung durch Apache Spark. Weiterhin sollten die Datenbanken innerhalb des Data Lakes Mechanismen verwenden, die eine Anfrage effektiv und möglichst in Echtzeit beantworten können.

4.2.4. Metadatenmanagement

Besonders wichtig für Datenanalysen im Bereich von Connected Cars sind Metadaten. Diese werden unter anderem dazu genutzt, die Daten in einen Kontext zu bringen bzw. zu wissen, was sie aussagen sollen. Zum Beispiel nehmen einige Sensoren, wie Rückfahrkameras, Bilder oder Videos auf. Würden diese Bilder und Videos ohne zusätzliche Informationen, z.B. Aufnahmeort oder -datum, abgespeichert werden, wäre es schwierig ein bestimmtes Bild oder Video wiederzufinden. Ähnlich würde es sich mit Anfragen zum Inhalt verhalten. Würde ein Service z.B. alle Videos mit Fußgängern von 2020 anfragen, könnte nur schwer eine Datenmenge zurückgegeben werden, da jedes Video einzeln analysiert werden müsste. Ein Service kennt außerdem nicht im Voraus das Schema der Datenhaltung, da im Data Lake ein Schema-On-Read verwendet wird (vgl. Abschnitt 2.1). Aus diesen Gründen ist es notwendig, Metadaten zu Datensätzen zusätzlich abzuspeichern. Im Fall der Bilder und Videos könnten beispielsweise Zeitpunkt, Inhalt und Ort der Aufnahme als Metadaten genutzt werden. Auch bei anderen Sensordaten wäre eine Bereicherung um Metadaten sinnvoll. Hier

könnten z.B. Ort, Zeitpunkt und Einheit der Messung abgespeichert werden. Eine Bereicherung von Datensätzen mit Metadaten kann vielseitig sein. So könnten einerseits Quellen schon Metadaten direkt mitgeben oder andererseits Metadaten extrahiert werden, durch beispielsweise Deep-Learning-Ansätze [LKM+15] oder Kontextwissen. Aber nicht nur Metadaten über die Datensätze selbst, sondern auch über die Datenbanken, welche innerhalb des Data Lakes genutzt werden, und über Quellsysteme sind von großer Relevanz. Hier werden z.B. Tabellenstrukturen der verschiedenen Datenbanken gespeichert oder Werte über das Quellsystem. Diese Metadaten helfen bei Abfrage von Daten aus dem Data Lake und bei dessen Management [GH19]. Das Metadatenmanagement kann mit verschiedenen Modellen umgesetzt werden, einen Überblick über diese wurde bereits in Abschnitt 2.1 gegeben.

4.2.5. Bereitstellung von Schnittstellen

Um den Data Lake effektiv nutzen zu können, ist eine Bereitstellung von Schnittstellen nötig. Nur so können alle Datenquellen ihre Daten zum Data Lake transportieren und Interessenten mit den Daten aus dem Data Lake arbeiten. Ziel ist, Datenquellen ein einfaches Abspeichern ihrer Daten im Data Lake zu ermöglichen. Daher sollte auf Seiten der Datenquelle keine größeren Änderungen am Quellcode nötig sein, um den Data Lake nutzen zu können. Dies kann beispielsweise durch eine API gelöst werden. Eine API stellt Funktionen des Data Lakes nach außen zur Verfügung. Diese Funktionen kann ein Service durch Aufrufen der jeweiligen Methode nutzen [DJ06]. Nachteil an dieser Schnittstellenart ist allerdings eine mehr oder weniger aufgezwungene Struktur zur Übergabe der Daten. Das bedeutet, Methoden einer API stellen meist Anforderungen, wie genau übergebene Daten aussehen müssen. Wie in Abschnitt 4.1.3 bereits erwähnt, haben Connected Cars keine einheitlichen Datenstukturen und -formate, weshalb diese von Hersteller zu Hersteller variieren können. Aus diesem Grund ist es schwer, von Seiten des Data Lakes eine Struktur und ein Format festzulegen, wie Daten an den Data Lake übergeben werden sollten. Außerdem würde eine Festlegung der Struktur dem Schema-On-Read-Ansatz [Mat17] eines Data Lakes widersprechen. Deshalb wäre die Bereitstellung von Message-Queues und einem Parser auf Seiten des Data Lakes eine bessere Entscheidung. So könnten Connected Cars ihre Daten per Nachricht an den Data Lake senden. Dieser parst daraufhin die erhaltene Nachricht und verarbeitet diese entsprechend weiter [GZ17b; LKM+15]. Um die Strukturen der unterschiedlichen Nachrichten verstehen zu können, können sich die Connected Cars und andere Datenquellen zuvor beim Data Lake über eine Registry registrieren. Kommen Daten einer Datenquelle beim Parser im Data Lake an, kann der Data Lake die Daten der Registry nutzen, um die erhaltene Nachricht entprechend zu mappen und weiter zu verarbeiten. Zur Abfrage des Data Lakes könnte wiederum eine API zur Verfügung gestellt werden, welche die Schnittstellen der im Data Lake verfügbaren Datenhaltungen vereint und entsprechende Metadaten berücksichtigt [GH19; LKM+15].

4.2.6. Unterstützung der Heterogenität

In Abschnitt 4.1.3 und Abschnitt 4.2.5 wurde angesprochen, dass Daten innerhalb eines ITS verschiedene Datenstrukturen und -formate besitzen und je nach Hersteller und Datenquelle variieren. Deshalb sollte ein Data Lake in einer Connected-Car-Umgebung diese Heterogenität unterstützen. Dies kann zum einen durch die Bereitstellung von Schnittstellen und zum anderen mit Unterstützung von verschiedenen Datenbanken erreicht werden [GGH+21]. So müssen einerseits die

Datenquellen ihre Daten nicht in eine bestimmte Struktur transformieren und andererseits werden Daten in für sie passende Datenbanken gespeichert. Eine für die Art passende Datenbank beeinflusst wiederum die Performanz positiv, da passende Datenbanken entsprechende Mechanismen zum effektiven Umgang mit der jeweiligen Datenart bereitstellen [GGH+21].

Nicht nur die Unterscheidung von Datenarten hinsichtlich ihrer Struktur muss beachtet werden, sondern auch, ob es sich bei den Daten um Stream-Daten oder Batch-Daten handelt. Je nachdem brauchen diese unterschiedliche Verarbeitungsprozesse. Stream-Daten sind hauptsächlich Daten, die vom Connected Car selbst kommen, z.B. Sensordaten. Diese werden kontinuierlich oder in minimalen Zeitabständen vom Connected Car gesendet [LKM+15]. In der Verarbeitung von Stream-Daten werden diese analysiert und es wird ggf. auf Ereignisse reagiert. Batch-Daten kommen meist von externen Quellen und werden nur bei Bedarf geschickt oder in größeren Zeitabständen, da diese über einen längeren Zeitraum gültig sind [GZ17b]. Hier handelt es sich unter anderem um Wetterdaten, Belegung von Parkplätzen usw. Deshalb sind Daten, die sich nicht innerhalb kürzer Zeitabstände ständig ändern, Batch-Daten. Meist dienen sie dazu, Stream-Daten einen Kontext zu geben [PKÅ+20]. Zum Beispiel wenn die Temperatur unter vier Grad ist, liegt die Geschwindigkeitsgrenze wegen Glatteis niedriger als bei einer Temperatur über vier Grad. Beide Datenarten benötigen verschiedene Verarbeitungsprozesse, da sie auch unterschiedlich analysiert werden. Daher sollte für einem Data Lake in einer Connected-Car-Umgebung die Lambda- oder Kappa-Architektur als Basis genutzt werden. Beide Architekturen unterstützen Stream-Verarbeitung bzw. Stream- und Batch-Verarbeitung [Kre14; WM15].

4.2.7. Weitere Anforderungen

Im Folgenden wird auf weitere Anforderungen eingegangen, die für einen Data Lake in einer Connected-Car-Umgebung wichtig sind. Hierbei handelt es sich vor allem um Qualitäten, die in vielen IT-Systemen vertreten sein sollten und sich daher nicht ausschließlich auf einen Data Lake beziehen. Trotzdem sollten diese nicht vernachlässigt werden, um einen Data Lake optimal in ein ITS integrieren zu können. Daher sollten die einzelnen Bestandteile des Data Lakes skalierbar sein. Je nach Einsatz, kann in einem Bestandteil mehr Leistung benötigt werden [GZ17b]. Zum Beispiel könnten mehrere Verarbeitungs-Jobs zeitgleich ausgeführt werden, um eine Echtzeitverarbeitung im gesamten System gewährleisten zu können. Genauso sollten die im Data Lake genutzten Datenbanken skalierbar sein, da eine große Menge an Daten verwaltet werden muss. Außerdem sollte der Data Lake sollte modular aufgebaut werden. Dies ermöglicht einen einfachen Austausch von Technologien in den einzelnen Komponenten [GZ17b]. Zum Beispiel könnte so die Technologie zur Datenverarbeitung ausgetauscht werden, ohne Anpassungen an anderen Komponenten vornehmen zu müssen. Ein anderes Beispiel wäre das Update auf neue Versionen. Weiterhin unterstützt die Modalisierung die Skalierbarkeit, da die Komponenten unabhängig voneinander skaliert werden können. Dadurch kann eine Skalierung nur in der Komponente vorgenommen werden, die eine höhere Arbeitslast hat. Somit werden vorhandene Ressourcen effizienter eingesetzt und der Data Lake kann sich an ändernde Kontexte anpassen.

5. Konzept eines Data Lakes in Connected-Car-Umgebungen

Nachdem in Kapitel 4 Anforderungen für einen Data Lake in einer Connected-Car-Umgebung definiert wurden, wird in diesem Kapitel auf ein Konzept zur Umsetzung dieses Data Lakes eingegangen. Die Idee zur Data-Lake-Umgebung basiert auf einer Aufteilung des zentralen Data Lakes in mehrere, verteilte Data Lagoons und einem Data Lake. Dabei liegen die Data Lagoons näher an den Connected Cars, wodurch die Übertragungswege erheblich verkürzt werden. Dieses Kapitel gibt zunächst in Abschnitt 5.1 einen Überblick über die Architektur der Data-Lake-Umgebung. Danach folgen genauere Erklärungen der einzelnen Bestandteile des Systems. In Abschnitt 5.2 wird auf das Konzept der Data Lagoon und in Abschnitt 5.3 des zentralen Data Lakes eingegangen. Abschließend wird die Registry für neue Datenquellen in Abschnitt 5.4 erläutert.

5.1. Überblick

Die Idee der Architektur basiert auf dem von Theodorou et al. [TD19] vorgestellten Konzept. Hier wird der Data Lake dezentralisiert, um eine Verarbeitung der Daten so nah wie möglich an der Datenquelle zu ermöglichen. Dadurch kann eine niedrigere Latenz erreicht und die Datenrate in der Infrastruktur besser balanciert werden. Diese Idee basiert auf dem Grundkonzept von Edge Computing. Im Edge Computing werden Berechnungen und Verarbeitungen von Daten aus der Cloud näher an die Quelle ausgelagert. Dadurch werden die Rechenressourcen von Knoten im System effektiv genutzt und schnellere Antworten sind möglich. Dabei können Knoten auf dem Weg zwischen Quelle und Cloud als sogenannte Edge Nodes fungieren. Vor allem Wege über das Netzwerk können ein Bottleneck sein. Diese können durch die Anwendung von Edge Computing verringert werden [SCZ+16]. Nach diesem Prinzip könnte auch ein Data Lake verteilt werden. Teile des Data Lakes werden dabei auf Edge Nodes ausgelagert. Diese Teile werden von Theodorou et al. [TD19] als *Data Lagoons* bezeichnet und sammeln die Daten von allen Geräten, die von der Edge Node verwaltet werden. In einer Connected-Car-Umgebung treten diese in Form von Fahrzeugen und RSUs auf. Dadurch soll die Datenaufnahme von der Datenanalyse getrennt werden. Durch Data Lagoons findet die Datenverabeitung näher an der Quelle statt, wodurch die Übertragungswege kürzer werden. Zusätzlich wird eine bessere Ressourcenausnutzung durch Nutzung der Rechenleistung der Edge Nodes erreicht. Weiterhin können in der Data Lagoon z.B. Rohdaten abgespeichert und nur vorverarbeitete Daten zum zentralen Data Lake geleitetet werden [TD19]. Für das folgende Konzept wurde die Idee einer Data Lagoon übernommen. Daher wird ein verteilter Data Lake, dessen Teile sich näher an den Fahrzeugen befinden, konzipiert.

Die konzipierte Architektur für die Connected-Car-Umgebung besteht aus verteilten Data Lagoons und einem zentralen Data Lake. Ein Überblick über die Gesamtarchitektur ist in Abbildung 5.1 zu sehen. Data Lagoons können z.B. auf Rechenressource in Ampeln oder Laternen gehostet

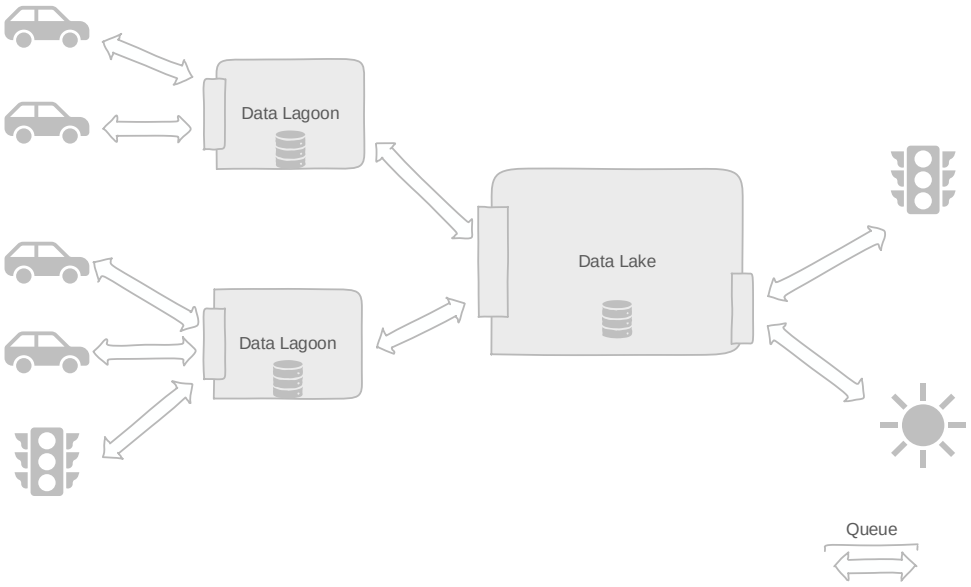

Abbildung 5.1.: Überblick über die Architektur eines Data Lakes in einer Connected-Car-Umgebung

werden. Fahrzeuge und RSUs können Daten über Message-Queues an die Data Lagoons oder den Data Lake direkt senden. Message-Queues sorgen für eine lose Kopplung zwischen den kommunizierenden Partnern. Außerdem können Nachrichten Daten unterschiedlicher Formate und Strukturen transportieren, was zur Heterogenität der Data-Lake-Umgebung beiträgt. Auch Data Lagoon und Data Lake kommunizieren über Message-Queues miteinander. So kann eine Data Lagoon empfangene Daten bei Bedarf an den Data Lake weiterleiten, ohne dass die Quelle die Daten mehrfach senden muss und umgekehrt. Weiterhin findet im Data Lagoon nicht nur eine Verarbeitung der Daten statt, sondern es können auch bestimmte Daten an den Data Lake weitergeleitet werden. Dadurch könnten z.B. auf der Data Lagoon die Rohdaten vorliegen und auf dem Data Lake nur privatisierte Daten. Dabei muss die Data Lagoon nicht alle gespeicherten Daten weiterleiten, sondern Daten können auch ausschließlich in der Data Lagoon gespeichert werden. Da die Data Lagoons im System verteilt sind und näher an den Fahrzeugen liegen, sollten diese bevorzugt für die Verarbeitung der gesendeten Stream-Daten dienen. Der Data Lake sollte einerseits die Stream-Daten von Fahrzeugen und Data Lagoons verarbeiten und andererseits Batch-Daten für Kontextwissen bereithalten. Bei vielen Batch-Daten ist die Nähe zum Data Lake irrelevant, da diese oft von externen Quellen, wie z.B. Wetterdaten, kommen oder systemweit gültig sind, wie Daten über Parkplatzbelegungen. Oft werden zentrale Batch-Daten im Data Lake an mehreren Stellen im System für Berechnungen benötigt. Deshalb können diese Art Daten direkt vom Data Lake verarbeitet werden. Eine Aufteilung, wo welche Daten verarbeitet werden, ist abhängig vom genauen Einsatzszenario des Systems. Dadurch könnten auch Data Lagoons Batch-Daten verarbeiten. Genauere Informationen hierzu folgen in den nächsten Abschnitten zum Aufbau der Data Lagoons und des Data Lake.

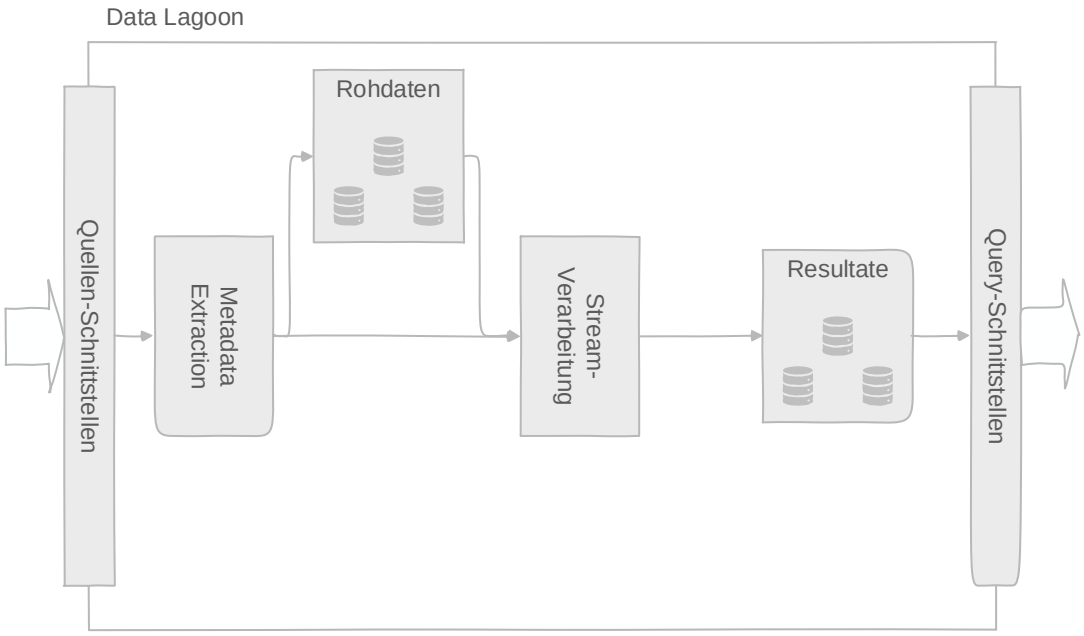

Abbildung 5.2.: Architektur der Data Lagoon

5.2. Data Lagoon

Abbildung 5.2 zeigt die Architektur der Data Lagoons. Der Aufbau einer Data Lagoon basiert auf der Kappa-Architektur. Eine Data Lagoon besitzt zwei Schnittstellen zur Außenwelt, eine für Datenquellen und eine für Nutzer der Daten. Über die *Quellen-Schnittstellen* nimmt die Data Lagoon Daten aus Message-Queues von Fahrzeugen usw. entgegen, um diese zu speichern und zu verarbeiten. Die *Query-Schnittstellen* dienen zur Abfrage der Data Lagoon von zugriffsberechtigten Services, Fahrzeugen, RSUs usw. Über diese Schnittstellen erfolgt zudem bei Bedarf die Weiterleitung der Daten an den Data Lake über entsprechende Message-Queues.

Über die Quellen-Schnittstelle erhaltene Daten werden zunächst durch die Komponente *Metadata Extraction* um Metadaten erweitert. Hier kann Kontextwissen hinzugefügt oder z.B. durch Deep-Learning-Techniken Mediendateien analysiert werden. Diese Metadaten können später genutzt werden, um Datensätze für Analysen zu finden. Metadaten können außerdem direkt von der Datenquelle mitgeliefert werden. Dies können zusätzliche Informationen innerhalb der gesendeten Nachricht oder auch technische Metadaten, die das Messaging-System bereitstellt, sein.

Nach dem Hinzufügen von möglichen Metadaten wird die eigentliche Datenverarbeitung basierend auf der Kappa-Architektur fortgeführt. Dabei werden ankommende Stream-Daten sofort verarbeitet und parallel für spätere Auswertungen persistiert. Die Kappa-Architektur eignet sich besonders, da die Data Lagoon meistens Daten von Conntected Cars, entgegennimmt. Diese Daten sind vor allem Sensordaten in Form von Streams, weshalb eine stream-basierte Verarbeitung am sinnvollsten ist. Hingegen kommen Batch-Daten seltener bei der Data Lagoon direkt an, wodurch eine Batch-Verarbeitung eine geringere Priorität besitzt. Die Kappa-Architektur fokussiert sich in ihrem Konzept mehr auf die Verarbeitung von Stream-Daten und ist daher eine geeignete Wahl für den Kontext des Data Lagoon. Dennoch ist eine Verarbeitung von Batch-Daten weiterhin möglich.

Die Daten werden zum einen als *Rohdaten* gespeichert, um sie für spätere Analysen nutzen zu können, und zum anderen an die *Stream-Verarbeitung* transportiert. Hier werden die Daten entsprechend gewünschter Analysen und Operationen für die Speicherung in den *Resultaten* vorbereitet. Eine Verarbeitung kann auch für die Weiterleitung der Daten zum Data Lake erfolgen. Eine Datenvorverarbeitung könnte z.B. Operationen wie Filtern, Mapping auf Intervalle oder Anwenden von Verschleierungsmechanismen enthalten. Genauso können in diesem Schritt die Daten mit anderen Informationen verrechnet und in den Resultaten abgespeichert werden. Letztendlich können die Daten je nach angewendeten Operationen in unterschiedlichen Verarbeitungsstufen in den Resultaten vorhanden sein. Für die Implementierung der Stream-Verarbeitung sollte ein Framework genutzt werden, das mit Datenstreams arbeiten kann, um die Vorteile dieser Verarbeitung optimal zu nutzen. Beispiele sind Apache Storm[1], Apache Samza[2] und Apache Flink[3].

Die verarbeiteten Daten werden schließlich als *Resultate* in der eigentlichen Datenhaltung der Data Lagoon abgespeichert. Die Datenhaltung besteht aus verschiedenen Datenbanken, die für verschiedene Datenstrukturen und -formate geeignet sind. Durch die Logik dieser Komponente wird entschieden, wo die Daten gespeichert werden. Hierfür werden Informationen zur Datenquelle aus der *Registry* (vgl. Abschnitt 5.4) genutzt. Die Resultate können über die Query-Schnittstelle abgefragt oder an z.B. Services, Data Lake, anderen Systemen oder zurück zum Fahrzeug weitergeleitet werden.

Das Konzept der Data Lagoons kann bei Bedarf um ein Zonenmodell erweitert werden. Dadurch kann eingeschränkt werden, welche Services auf welche Daten Zugriff haben, was zur Erhaltung der Privatheit beitragen kann. Eine Möglichkeit wäre die Anwendung eines ähnlichen Prinzips, das Stach et al. [SGW+20] für AMNESIA nutzen. Hier wird der Datenspeicher, der Machine-Learning-Modelle beliefert, in verschiedene Level unterteilt. Jedes dieser Level hat unterschiedliche Privatheitsanforderungen. Auf neu ankommende Daten werden verschiedene Filter für Privatheit angewendet und die maskierten Daten in den jeweiligen Leveln gespeichert. Ähnliches kann auf das vorliegende Konzept angewendet werden. Die Data Lagoon nimmt neu ankommende Daten entgegen und wendet Filter für Privatheit an. Danach speichert die Data Lagoon die maskierten Daten in der jeweiligen Zone innerhalb der Datenhaltung oder leitet sie an den Data Lake weiter. Services können über die Query-Schnittstellen nur Zonen abfragen, für die sie die entsprechenden Zugriffsrechte besitzen.

Nicht nur für Privatheit kann ein Zonenmodell verwendet werden, sondern auch um Resultate in verschiedenen Verarbeitungsgraden bereitzustellen. Wie in Abschnitt 2.1.3 bereits erklärt, teilen z.B. Giebler et al. [GGH+20] die Zonen in *Landing, Raw, Harmonized, Distilled, Explorative* und *Delivery* ein. Jede dieser Zonen stellt Daten in unterschiedlichen Verarbeitungsgraden für unterschiedliche Zwecke bereit. Eine ähnliche Aufteilung könnte auch für die Data Lagoon genutzt werden. Die Landing Zone übernimmt die Datennahme und es werden keine Verarbeitungsschritte ausgeführt. Die Aufgabe dieser Zone könnten die Quellen-Schnittstellen übernehmen. Die Raw Zone nimmt die Daten aus der Landing Zone entgegen und speichert sie im Rohformat bzw. nur mit kleineren Anpassungen ab. Hier wären dies die Rohdaten nach der Erweiterung mit Metadaten. Die Harmonized Zone kopiert die Rohdaten. Diese Daten werden für die weitere Verarbeitung genutzt und sind für Analysen zugreifbar. So wird sichergestellt, dass die Rohdaten der Raw Zone

[1] Apache Storm: https://storm.apache.org/
[2] Apache Samza: http://samza.apache.org/
[3] Apache Flink: https://flink.apache.org/

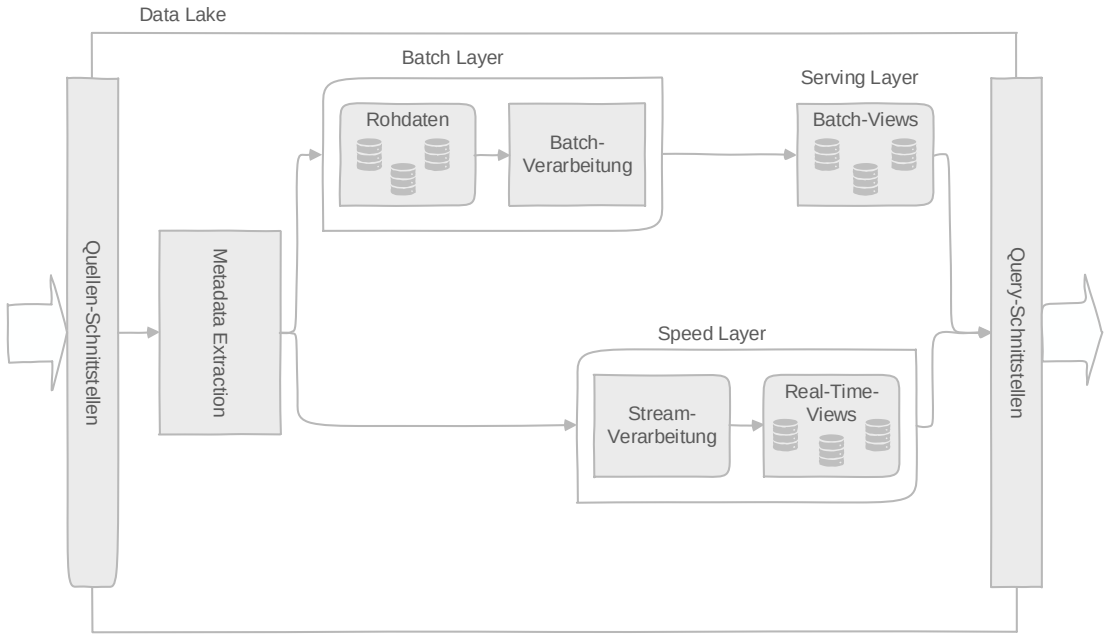

Abbildung 5.3.: Architektur des Data Lakes

in ihrer originalen Form verbleiben. In der Distilled Zone liegen die Daten schließlich nach erster Verarbeitung in aggregierter Form vor. Diese Zone dient für einen effektiveren Zugriff auf Daten, die für Analysen benötigt werden. Da diese Daten bereits aggregiert sind, fällt der Aufwand für Berechnungen weg. In der Data Lagoon wäre diese Zone durch die Datenbanken, die die Resultate speichern, repräsentiert. Eine Explorative Zone, die Data Scientists zum Erkunden der Daten nutzen können, hat in der Data Lagoon keinen großen Nutzen und kann deshalb weggelassen werden. Zuletzt stellt die Delivery Zone verarbeitete Daten für spezifische Zwecke und bestimmte Services zur Verfügung. Auch diese Daten könnten nach Verarbeitung in den Datenbanken der Resultate gespeichert werden. Außerdem sollte jede Zone durch Zugriffsbeschränkungen geschützt werden.

5.3. Data Lake

In Abbildung 5.3 ist die Architektur des Data Lakes abgebildet. Im Gegensatz zum Data Lagoon beruht der Data Lake auf der Lambda-Architektur. Allerdings überschneiden sich Komponenten wie Schnittstellen und *Metadata Extraction*. Die *Quellen-Schnittstellen* empfangen unter anderem Daten von Data Lagoons und anderen Datenquellen über Message-Queues. Im Data Lake fließen die Daten der Data Lagoons zusammen. Diese wurden von den jeweiligen Data Lagoons vorverarbeitet und über Message-Queues gesendet. Die *Query-Schnittstellen* dienen auch hier zur Abfrage und Weiterleitung der Daten. Data Lagoons können diese Schnittstellen nutzen, um zusätzliche Daten für Berechnungen und Analysen abzufragen. Die Metadata Extraction arbeitet im Data Lake nach dem gleichen Prinzip wie in der Data Lagoon.

Da im Data Lake nicht nur Stream-Daten aus den Data Lagoons oder Fahrzeugen, sondern auch Batch-Daten aus externen Quellen gepeichert werden, wird die Lambda-Architektur als Basis genutzt. Die Verarbeitung im Data Lake unterteilt sich in drei Layer. Der *Batch Layer* speichert die Rohdaten

und führt auf diesen eine *Batch-Verarbeitung* durch. Dabei werden auf Daten unterschiedliche Operationen oder Berechnungen ausgeführt. Die erhaltenen Resultate werden in *Batch-Views* im *Serving-Layer* gespeichert. Der Serving Layer dient dazu, die in Batch Views gespeicherten Daten zur Verfügung zu stellen. Stream-Daten werden über den *Speed Layer* verarbeitet. Dieser besitzt eine *Stream-Verarbeitung* und speichert seine Resultate in *Real-Time-Views*. Hier werden unter anderem die Stream-Daten aus den Data Lagoons empfangen. Die Datenhaltung von Batch-Views und Real-Time-Views besteht jeweils aus Datenbanken, die für unterschiedliche Datenformate und -arten geeignet sind. Eine Speicherung erfolgt wie bei der Data Lagoon unter Berücksichtigung der Informationen aus der Registry. Sowohl Batch-Views als auch Real-Time-Views werden über die Query-Schnittstellen gekapselt.

Bei Bedarf kann wieder um ein Zonenmodell erweitert werden. Im Prinzip wäre eine Anwendung eines Zonenmodells recht ähnlich zu der Anwendung in der Data Lagoon. In den Zonen müsste lediglich die Unterscheidung zwischen Batch-Verarbeitung und Stream-Verarbeitung berücksichtigt werden. Zusätzlich könnte im Data Lake die Explorative Zone eingerichtet werden, da dieser ein Sammelpunkt der Daten ist und somit explorative Analysen durch Data Scientists möglich wären.

5.4. Registry

Die Registry ist ein Repository für Data Lagoon und Data Lake. Es wird genutzt, um Informationen über Datenstrukturen und -formate der Datenquellen zu sammeln und bereitzustellen. Bevor eine Datenquelle Nachrichten zu Data Lagoons oder Data Lake schicken kann, muss sie sich bei der Registry registrieren. Dafür stellt die Registry eine REST-API zur Verfügung. Mit dem Endpunkt zur Registrierung werden Informationen über Datenstruktur, -format und Nachrichtenaufbau abgefragt. Diese Informationen können in den Data Lagoons und dem Data Lake genutzt werden, um ankommende Nachrichten entsprechend zu parsen und verarbeiten. Außerdem werden diese Informationen genutzt, um die erhaltenen Daten nach der Verarbeitung in entsprechende Datenbanken zu speichern.

Das Grundprinzip der Registry ist vergleichbar mit der von Franco da Silva et al. [FHB+18] entwickelten *Topic Description Language for the Internet of Things (TDLIoT)*. Hier wird eine Art Repository erstellt, bei dem Beschreibungen für Topics innerhalb eines IoT-Systems hinterlegt werden können. Eine Beschreibung enthält Informationen über die Benutzung des Topics. Ein Service innerhalb des IoT-Systems kann das Repository abfragen und enthält Beschreibungen derer Topics, die zur gewünschten Funktion passen. Der Service kann das passende Topic aussuchen und es anhand der Beschreibung abonnieren. In der Connected-Car-Umgebung würde eine Registry als Informationsquelle von Beschreibungen über Datenquellen dienen. Dabei müsste sich nicht jedes Connected Car mit jeder datenproduzierenden ECU registrieren. Stattdessen sollten die Hersteller diese Beschreibungen in der Registry aktuell halten. Als Voraussetzung müssen sich in den Metadaten die entsprechenden Informationen über Hersteller und Modell, die Data Lake und Data Lagoon zur Abfrage der Registry nutzen können, befinden. Eine prototypische Implementierung einer Registry wird in Abschnitt 7.2.6 vorgestellt.

6. Evaluierung der Anforderungen

In Abschnitt 4.2 wurden Anforderungen definiert, die ein Data Lake in einer Connected-Car-Umgebung erfüllen sollte. Anhand dieser Anforderungen wird in diesem Abschnitt das Konzept mit der Aufteilung in Data Lagoon und Data Lake evaluiert. Dafür wird im Folgenden jede der Anforderungen aufgegriffen und erklärt, wie diese im vorgestellten Konzept umgesetzt werden.

6.1. Sicherheit

Die Erfüllung der Sicherheitsanforderungen ist implementierungsabhängig. Bei der Auswahl von Datenbanken und Werkzeugen für Stream- und Batch-Verarbeitung sollte auf deren Sicherheitsaspekte geachtet werden. Auch bei der Konfiguration der Bestandteile der Data-Lake-Umgebung können einige Sicherheitsmaßnahmen berücksichtigt werden. Bei vielen Datenbanken ist es möglich, spezielle Nutzer mit eigenen Rechten anzulegen. Deshalb sollte diese Funktion genutzt werden, statt den Data Lake und die Data Lagoon über den Root-Nutzer der Datenbank auf die Daten zugreifen zu lassen. Im Normalfall hat ein Root-Nutzer alle Rechte über eine Datenbank und kann alle operative und administrative Funktionen nutzen. Sollte ein Angreifer über den Data Lake oder die Data Lagoon Zugriff auf das System erlangen, könnte er sich frei innerhalb der Datenbanken bewegen. Aus diesem Grund sollten für Data Lake und Data Lagoon eigene Nutzer für Datenbanken innerhalb ihrer Datenhaltung angelegt werden. Diese Nutzer sollten nur die Rechte besitzen, die für den Betrieb der Data-Lake-Umgebung nötig sind.

Weiterhin wird der Zugriff auf Data Lake und Data Lagoons über Schnittstellen gekapselt. Somit fragen Services nicht direkt die jeweilige Datenhaltung ab und es ist sichergestellt, dass nur befugte Services Zugriff haben. Bei Nutzung eines Zonenmodells, ähnlich zu den in Abschnitt 2.1.3 vorgestellten, können Daten in verschiedenen Verarbeitungsstufen je nach Zone angeboten werden. Services hätten nur Zugriff auf Daten in ihrer freigegebenen Zone. So könnte abhängig von der Zone entschieden werden, auf welche Daten und in welcher Form ein Service Zugriff hat. Eine weitere Möglichkeit das System zu schützen ist die Implementierung von Sicherheitsmaßnahmen. Im Rahmen dieser Arbeit wird davon ausgegangen, dass alle Daten, die von den Datenquellen gesendet werden, sicher sind. Sicherheitsmaßnahmen innerhalb der Datenquellen liegen außerhalb des Fokus dieser Arbeit.

6.2. Privatheit

Privatheit der persönlichen Daten innerhalb der Data-Lake-Umgebung wird unter anderem mit Hilfe der vorgestellten Sicherheitsaspekte erreicht. Zugriffsbeschränkungen und Sicherheitsmaßnahmen helfen den Zugriff auf personalisierte Daten zu beschränken und zu kontrollieren. Dadurch können nur berechtigte Services diese Daten abfragen.

Weiterhin sollte in den Verarbeitungskomponenten die Implementierung von Schutzmaßnahmen wie Verschleierungsmechanismen und Anonymisierung berücksichtigt werden. Außerdem ist es möglich, über die Data Lagoons eine Weiterleitung zum Data Lake zu kontrollieren. Hier kann genau definiert werden, welche Daten in welcher Form zum Data Lake transportiert werden. Zum Beispiel können statt der genauen Geschwindigkeit des Fahrzeugs nur Intervalle, in denen die Geschwindigkeit liegt, weitergeleitet werden. Weiterhin könnte für die Stauerkennung nur Bereiche, die ein erhöhtes Fahrzeugaufkommen haben, gesendet werden statt eine genaue Anzahl und Standorte der Fahrzeuge. Wie zuvor beschrieben, kann durch Anwendung eines Zonenmodells zusätzliche Privatheit geschaffen werden.

6.3. Hohe Performanz und niedrige Latenz

Durch die Aufteilung eines zentralen Data Lakes in Data Lagoons und Data Lake werden Übertragungswege für Daten kürzer. Die Data Lagoons liegen näher an den Connected Cars, wodurch weniger Bandbreite für die Übertragung der Daten benötigt wird. Diese Nähe wirkt sich positiv auf auf die Latenz aus.

Connected Cars produzieren vor allem Sensordaten in Form von Stream-Daten. Daher sollte die Verarbeitung innerhalb des Data Lakes Tools und Technologien nutzen, die dies effektiv ermöglichen. Außerdem werden die Lambda- und Kappa-Architekturen als Basis genutzt. Dies bedeutet, beide Architekturen berücksichtigen eine Verarbeitung von Stream-Daten.

6.4. Metadatenmanagement

Die Metadata Extraction ermöglicht eine Bereicherung um Metadaten. Hier können ankommende Daten anhand von Mechanismen zur Metadatenextraktion, wie Deep-Learning, und Kontextwissen erweitert werden. Außerdem können Quellen eigene Metadaten in den Nachrichten mitsenden. All diese Metadaten helfen Daten wiederzufinden oder bei Analysen als relevant zu berücksichtigen. Die Metadaten zu jedem Datensatz werden in der Datenhaltung der Data Lagoons bzw. des Data Lakes abgespeichert.

Jedoch wurde kein Metadatenmanagement basierend auf einem Metadatenmodell konzipiert. Das vorliegende Konzept könnte durchaus um ein Metadatenmodell erweitert werden. Allerdings müsste zusätzliche Arbeit für die Konzeptionierung eines solchen Modells aufgewendet werden. Einige technische Metadaten können schon gesammelt und in Tools wie Apache Atlas[1] verwendet werden. Jedoch wird hierbei Wissen über die konkrete Implementierung benötigt.

[1] Apache Atlas: https://atlas.apache.org

6.5. Bereitstellung von Schnittstellen

Das vorgestellte Konzept bietet Schnittstellen zur Datenannahme und zur Datenabfrage an. Sowohl die Data Lagoons als auch der Data Lake nehmen über die Quellen-Schnittstellen Daten aus Message-Queues entgegen. Über Message-Queues können Connected Cars ihre gesammelten Daten einfach an die nächstgelegene Data Lagoon streamen, ohne deren Struktur und Format anpassen zu müssen. Die Data Lagoon parst die ankommenden Nachrichten anhand der Informationen über die Datenquelle in der Registry. Lediglich eine Registrierung der Quelle mit Informationen über Nachrichteninhalt und -struktur bei der Registry ist nötig, um gesendete Daten in den Data Lagoons und im Data Lake verarbeiten zu können. Diese Registrierung kann von Herstellern verwaltet werden.

Für die Abfrage der Data Lagoons und des Data Lakes werden Query-Schnittstellen zur Verfügung gestellt. Diese Schnittstellen bieten zum einen die Möglichkeit verarbeitete Daten über Message-Queues weiterzuleiten, beispielsweise bei der Kommunikation zwischen Data Lagoons und Data Lake. Zum anderen können APIs definiert werden, die Funktionen zur Abfrage des Data Lakes bzw. der Data Lagoon durch Services umfassen. Dabei sollte auf Zugriffsbeschränkungen hinsichtlich Sicherheit und Privatheit geachtet werden. Eine genaue Definition, wie eine solche Schnittstelle aussehen könnte, liegt nicht im Rahmen dieser Arbeit.

6.6. Unterstützung der Heterogenität

Durch die Möglichkeit Daten über Message-Queues zu senden, sind Datenquellen nicht gezwungen ihre Daten in eine bestimmte Struktur zu transformieren. Dadurch können Connected Cars, RSUs und weitere Datenquellen, wie Services und Smartphones, Daten unabhängig von Struktur und Format zur Data Lagoon bzw. zum Data Lake senden. Daraufhin parsen Data Lagoons und Data Lake die Nachrichten und verarbeiten diese entsprechend. Die erhaltenen Daten werden nach den Verarbeitungsschritten entsprechend in verschiedenen Datenbanken gespeichert. Dabei werden Datenbanken für strukturierte, semi-strukturierte und unstrukturierte Daten zur Verfügung gestellt. Die Verarbeitungskomponenten in Data Lagoon und Data Lake entscheiden unter anderem mit Hilfe der Informationen aus der Registry, wo und wie die ankommenden Daten gespeichert werden. Der genaue Aufbau der Datenhaltung hinsichtlich Datenbanktechnologien und wie ggf. Zonen eingeteilt werden, ist implementierungsabhängig. Dabei sollte der genaue Anwendungsfall in Betracht gezogen und Technologien entsprechend gewählt werden.

Auch eine Verarbeitung von sowohl Stream-Daten als auch Batch-Daten ist möglich. Die Data Lagoon nimmt Daten in Form von Streams entgegen, verarbeitet diese und streamt sie ggf. weiter. Parallel werden die ankommenden Daten gespeichert, um sie für zukünftige Analysen zu nutzen. Der Data Lake besitzt mit der Batch-Verarbeitung und Stream-Verarbeitung zwei Komponenten für die Verarbeitung von Daten. Je nach Art der zu verarbeiteten Daten wird die passende Verarbeitung gewählt.

6.7. Weitere Anforderungen

Der Aufbau von Data Lagoon und Data Lake ist modular. Sowohl Data Lagoon als auch Data Lake besitzen Bestandteile für Verarbeitung, Metadatenextraktion, Datenspeicher und Schnittstellen. Diese einzelnen Bestandteile können unabhängig voneinander implementiert und z.B. über Message-Queues verbunden werden, wodurch eine lose Kopplung zwischen den Komponenten erreicht wird. Wenn eine Technologie innerhalb einer Komponente ausgetauscht werden soll, kann dies ohne direkten Einfluss auf die anderen Komponenten geschehen. So könnte z.B. in der Stream-Verarbeitung Apache Flink durch Apache Samza ersetzt oder Datenbanken ausgetauscht werden, ohne weitere Änderungen an anderen Komponenten vornehmen zu müssen. Durch diesen modularen Aufbau können Technologien und Implementierungen dynamisch an den Kontext angepasst werden. Weiterhin unterstützt ein modularer Aufbau die Skalierbarkeit des Systems.

Die Komponenten der gesamten Data-Lake-Umgebung sind skalierbar. Die Anzahl der Data Lagoons im ITS können leicht angepasst werden, da diese eigenständige Subsysteme sind, die mit dem Data Lake über Schnittstellen kommunizieren. Für den Data Lake ist es irrelevant, von wie vielen Data Lagoons er Daten empfängt. Auch innerhalb von Data Lagoon und Data Lake lassen sich die einzelnen Komponenten skalieren. Die Verarbeitungskomponenten können z.B. je nach Arbeitslast skalieren und mehrere Verarbeitungs-Jobs parallel ausführen. Gleiches gilt für die Datenbanksysteme, die mit der Datenmenge skalieren können. Dabei hängt die Skalierbarkeit vor allem von den genutzten Technologien ab.

6.8. Einschränkungen

In diesem Abschnitt wird auf einige Einschränkungen des Konzepts eingegangen. Zwar werden Metadaten zu einzelnen Datensätzen, wie Inhalt eines Bildes oder Videos, berücksichtigt, aber ein Metadatenmanagement basierend auf einem Metadatenmodell (vgl. Abschnitt 2.1) fehlt. Es können dennoch Werkzeuge für das Metadatenmanagement angewendet werden. Ein Beispiel hierfür ist Apache Atlas, das in Abschnitt 7.2.7 genauer vorgestellt wird.

Eine weitere Einschränkung des Konzepts liegt bei den Query-Schnittstellen. Diese sind, bis auf Message-Queues zwischen den Akteuren, nicht genauer definiert. Dazu sollte noch beschrieben werden, wie genau APIs zum Abfragen von Data Lagoon und Data Lake aussehen könnten.

Eine letzte Einschränkung ist die Synchronisation der Daten zwischen den Data Lagoons. Connected Cars sollten Daten zur nächstgelegenen Data Lagoon streamen können. Diese kann sich im Laufe einer Fahrt mehrfach ändern. Zwar berücksichtigt das vorliegende Konzept eine einfache Verbindung zur Data Lagoon, ein Wechsel der nächstgelegenen Data Lagoon, d.h. eine mögliche Antwort einer anderen Data Lagoon, wird nicht berücksichtigt. Daher müsste das Konzept um eine Synchronisation zwischen Data Lagoons erweitert werden.

Sowohl die Konzeptionierung eines Metadatenmanagements als auch die Definition genauer Schnittstellen zur Abfrage, sind große Teilbereiche einer Data-Lake-Umgebung und müssen daher mit großer Genauigkeit betrachtet werden. Dies war im Umfang dieser Arbeit nicht möglich.

7. Implementierung eines Data Lakes in Connected-Car-Umgebungen

In diesem Kapitel wird eine prototypische Implementierung des Konzepts vorgestellt. Zunächst wird ein Anwendungsfall definiert, den der Prototyp umsetzen soll. Dabei wird auf die Unterteilung zwischen Data Lagoon und Data Lakes sowie der Verarbeitung von mehreren Datenquellen aus verschiedenen Fahrzeugen geachtet. Der Prototyp stellt eine Implementierung einer Data Lagoon, eines Data Lake und einer Data Registry bereit. Um Daten zum Prototypen streamen zu können, wird die Implementierung um einen Datengenerator, der die Datenquellen simuliert, erweitert. Zusätzlich werden einige alternative Technologien getestet. Im Folgenden wird zunächst in Abschnitt 7.1 der Anwendungsfall, der als Grundlage der Implementierung dient, definiert. Daraufhin werden die Details der Implementierung in Abschnitt 7.2 erläutert.

7.1. Anwendungsfall

In diesem Abschnitt wird ein Anwendungsfall für eine prototypische Implementierung des vorgestellten Konzepts definiert. Dieser Prototyp soll ein Proof-of-Concept darstellen und daher einige Anforderungen erfüllen. Zum einen soll eine Data Lagoon, die Daten von Connected Cars annimmt und an einen Data Lake weiterleitet, implementiert werden. Dabei soll eine Verarbeitung der Daten vor der Weiterleitung an den Data Lake berücksichtigt werden. Zum anderen soll ein Data Lake, der Daten von der Data Lagoon und externen Quellen speichert, aufgesetzt werden. Sowohl Data Lagoon als auch Data Lake können eine bereitgestellte Registry nach Informationen über Datenquellen abfragen. Daher soll die Registry die Möglichkeit bieten, Datenquellen mit Informationen, wie Nachrichtenstruktur, Datenformate und -struktur zu registrieren. Der Prototyp soll Daten von mehreren Connected Cars verarbeiten können. Diese Connected Cars und externe Quellen werden durch einen Datengenerator simuliert. Alle Bestandteile des Prototypen sollen mit Message-Queues verbunden werden. Um die Heterogenität des Konzepts zu zeigen, sollen Stream- und Batch-Daten sowie strukturierte, semi-strukturierte und unstrukturierte Daten verarbeitet werden. Dazu sollen geeignete Verarbeitungstools und Datenbanken eingesetzt werden. Außerdem sollen Metadaten berücksichtigt werden.

Dadurch ergibt sich der folgende Anwendungsfall. Der Aufbau ist in Abbildung 7.1 dargestellt. Ein Datengenerator generiert Stream- und Batch-Daten, die innerhalb der Data-Lake-Umgebung verarbeitet werden können. Dabei simuliert der Datengenerator Geschwindigkeitsdaten und Bilddaten einer verbauten Kamera von mehreren Connected Cars als Beispiel für Stream-Daten. Weiterhin werden Wetterdaten eines Wetterdienstes als Beispiel für Batch-Daten genutzt. Damit haben die generierten Daten nicht nur unterschiedliche Arten sondern auch verschiedene Strukturen.

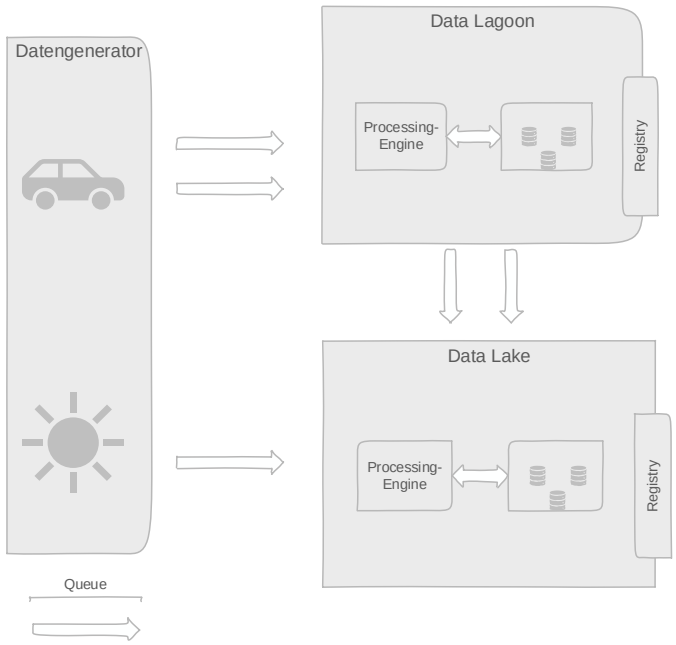

Abbildung 7.1.: Aufbau Anwendungsfall

Die Geschwindigkeiten sind Zeitreihendaten, die Aufnahmen der Kameras fallen unter die semi-strukturierten Daten und die genutzten Wetterdaten sind strukturierte Daten. Außerdem werden für die Datensätze der Geschwindigkeiten und Bilder jeweils Metadaten generiert. Ein Beispiel sind die Bildinhalte zur Personenerkennung.

Die generierten Stream-Daten werden in Nachrichten in Form von JSON zur Data Lagoon transportiert. Die Data Lagoon nimmt die Daten als Streams entgegen und verarbeitet diese. Dazu wird eine Processing-Engine, die für Stream-Verarbeitung ausgelegt ist, genutzt. Diese Processing-Engine wendet verschiedene Methoden zur Datenverarbeitung auf den jeweiligen Streams an. Danach werden die verarbeiteten Daten zur Datenhaltung des Data Lagoons und zum Data Lake weitergeleitet. Dabei kann der Verarbeitungsgrad für beide Ziele unterschiedlich sein. Die Datenhaltung der Data Lagoon besteht aus verschiedenen Datenbanken, die jeweils verschiedene Datenarten und -strukturen verwalten. Anhand der Informationen aus der Registry entscheidet die Logik der Datenhaltung, in welche der Datenbanken die erhaltenen Daten abgelegt werden. Auch die mitgesendeten Metadaten werden in einer Datenbank gespeichert.

Der Datengenerator schickt die Wetterdaten über eine Message-Queue in großen Abständen direkt zum Data Lake. Außerdem kommen die vorverarbeiteten Streams der Data Lagoon mit Geschwindigkeits- und Bilddaten beim Data Lake an. Analog zur Data Lagoon besitzt der Data Lake eine Processing-Engine und eine Datenhaltung bestehend aus verschiedenen Datenbanken. Die Processing-Engine führt Operationen auf den Streams mit Geschwindigkeits- und Bilddaten aus und leitet die resultierenden Daten an die Datenhaltung weiter. Dort werden die Daten zusammen mit ihren Metadaten und mit Hilfe der Informationen aus der Registry in die jeweiligen Datenbanken gespeichert. Weiterhin werden die Wetterdaten durch die Datenhaltung des Data Lakes entgegengenommen und abgespeichert.

Die Registry verwaltet zusätzliche Informationen zu Geschwindigkeits-, Bild- und Wetterdaten. Hierbei handelt es sich vor allem um Informationen zu Nachrichten- und Datenstruktur, beispielsweise welches Feld der Nachricht die Geschwindigkeit enthält oder wie Metadaten gekennzeichnet sind. Anhand dieser Informationen können Data Lagoon und Data Lake Strukturen in den Datenbanken anlegen, ankommende Nachrichten entsprechend parsen und deren Daten in die richtige Datenbankstruktur ablegen. Daher bietet die Registry einen Endpunkt, um diese Informationen über Quellen zu hinterlegen. Dazu nutzt die Registry eine eigene Datenbank mit einer Schnittstelle zum Data Lake und zur Data Lagoon.

Der definierte Anwendungsfall dient vor allem der Demonstration, weshalb nur eine kleine Auswahl an Datenquellen verwendet wird. Jedoch könnte er auf ein Szenario mit mehreren Datenquellen erweitert werden. Die Datenquellen wurden gewählt, da diese unterschiedliche Eigenschaften haben, mit denen die prototypische Implementierung umgehen können sollte. Im Anwendungsfall werden nur strukturierte und semi-strukturierte Daten verarbeitet. Unstrukturierte Daten werden nicht berücksichtigt. Das liegt daran, dass sowohl semi-strukturierte als auch unstrukturierte Daten meist in NoSQL-Datenbanken gespeichert und somit ähnlich behandelt werden. Stattdessen wird ein größerer Fokus auf die Verarbeitung von Zeitreihendaten gelegt. Da eine Metadatenextraktion außerhalb des Umfangs dieser Arbeit liegt, wird im Folgenden davon ausgegangen, dass die Metadaten bereits vom Connected Car mitgeschickt werden.

7.2. Details zur Umsetzung

Im Folgenden wird die Implementierung des Anwendungsfalls erläutert. Dabei wird genauer auf die einzelnen Bestandteile, ihre Umsetzung und verwendete Technologien eingegangen. Außerdem wird die Struktur der Nachrichten, die vom Datengenerator über die Data Lagoon zum Data Lake gesendet werden, aufgezeigt. Zu Beginn wird in Abschnitt 7.2.1 ein Überblick über die Implementierung und die verwendeten Technologien gegeben. In Abschnitt 7.2.2 wird die Simulation der Datenquellen mit dem Datengenerator erläutert. Hier werden die gesendeten Nachrichten und die verwendeten Beispieldaten aufgezeigt. Abschnitt 7.2.3 und Abschnitt 7.2.4 beschäftigen sich mit der Implementierung der Data Lagoon und des Data Lakes. Bei beiden werden Datenannahme, -verarbeitung und -speicherung erklärt. Dabei geht Abschnitt 7.2.5 genauer auf die gesammelten Metadaten ein. Daraufhin wird in Abschnitt 7.2.6 die Data Registry konkretisiert. In Abschnitt 7.2.7 werden alternative Tools und Erweiterungen des Prototyps vorgestellt. Schließlich wird in Abschnitt 7.2.8 auf einige Punkte, die bei der Implementierung beachtet werden sollten, eingegangen.

7.2.1. Überblick

In Abbildung 7.2 wird der Aufbau der Implementierung des Prototyps dargestellt. Hier sind die einzelnen Bestandteile der Data-Lake-Umgebung und ihre Kommunikation untereinander abgebildet. Die verschiedenen Technologien des Prototyps werden auf einer zur Verfügung gestellten Openstack[1]-Umgebung gehostet. Die Technologien werden auf Ubuntu[2]-Instanzen bereitgestellt.

[1] Openstack: https://www.openstack.org/
[2] Linux Ubuntu 16.04: https://ubuntu.com/server

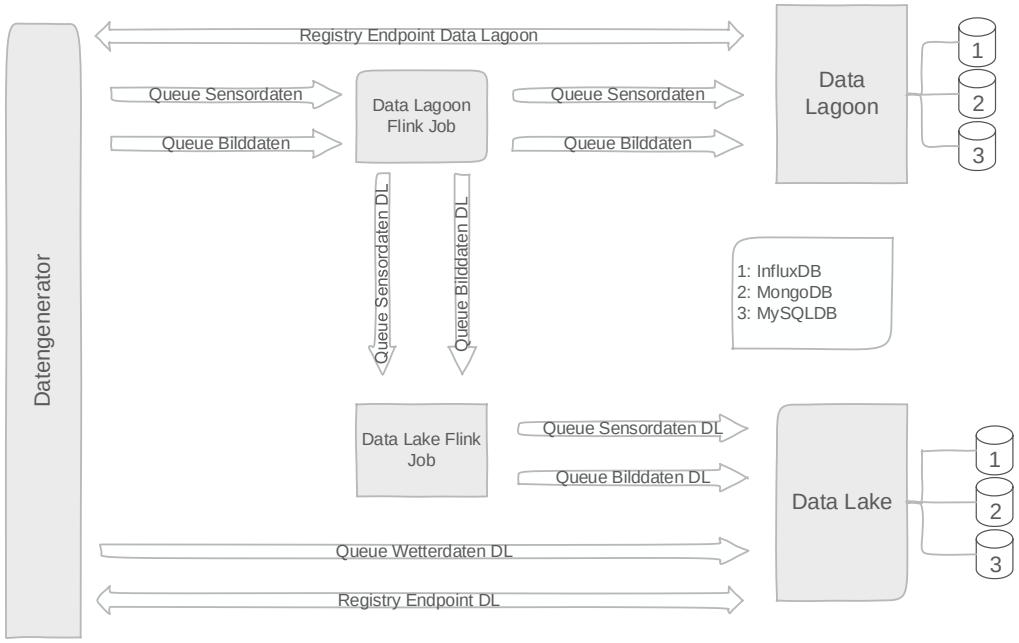

Abbildung 7.2.: Ablaufdiagramm des Prototyps

Die Kommunikation der Bestandteile des Prototyps findet über Apache Kafka[3] statt. In Kafka können Topics erstellt werden, über die eine Applikation Nachrichten senden kann. Die empfangende Applikation kann diese Topics abonnieren und die Nachrichten verarbeiten. Data Lagoon und Data Lake abonnieren Topics, auf welche der Datengenerator Daten sendet. Innerhalb von Data Lagoon und Data Lake erfolgt die Weiterleitung von Daten ebenfalls mit Hilfe von Kafka.

Sowohl Data Lagoon als auch Data Lake nutzen eine Processing-Engine für die Stream-Verarbeitung. Für den Prototypen wurde sich für Apache Flink[4] entschieden. Flink ist ein Framework, mit dem sich Jobs zur Stream-Verarbeitung implementieren und verarbeiten lassen. Dabei stellt Flink Bibliotheken für die Verbindung zu verschiedenen Datenquellen und Datensenken bereit. Im Rahmen des Prototypen werden für Data Lagoon und Data Lake Flink-Jobs implementiert und zu einer Flink-Instanz hochgeladen. Diese Flink-Instanz wird auf einer Ubuntu-Instanz des Openstack gehostet, um die Flink-Jobs auszuführen. Die Flink-Jobs hören auf die jeweiligen Topics, die Stream-Daten senden. Für den Prototypen werden insgesamt zwei Flink-Jobs implementiert, zum einen für die Stream-Verarbeitung der Data Lagoon und zum anderen für die Stream-Verarbeitung des Data Lakes. Flink stellt eine Vielzahl an Möglichkeiten bereit, um Daten in einem Stream zu verarbeiten. In der Implementierung werden kleine Filter und Wertänderungen genutzt, um zu zeigen, dass eine Verarbeitung der Daten prinzipiell möglich ist. Flink stellt viele weitere Methoden zur Verarbeitung bereit und bietet darüber hinaus die Möglichkeit eigene Methoden zu implementieren. Somit ist nach Bedarf auch eine komplexere Verarbeitung der Daten möglich. Genauere Informationen zum Aufbau und Verarbeitungsschritte der Flink-Jobs folgen in Abschnitt 7.2.3 und Abschnitt 7.2.4.

[3] Apache Kafka: https://kafka.apache.org/
[4] Apache Flink: https://flink.apache.org/

Die Datenhaltung besteht bei der Data Lagoon und dem Data Lake jeweils aus drei Datenbanken mit verschiedenen Eigenschaften. Zur Speicherung von Zeitreihendaten, worunter auch die Geschwindigkeitsdaten fallen, wird InfluxDB[5] genutzt. InfluxDB wurde speziell für die Speicherung von Zeitreihendaten entwickelt, wobei der Zeitstempel zur Indexierung dient und somit einen effektiven Datenzugriff ermöglicht. Die Verwaltung von Wetterdaten und Metadaten erfolgt durch die relationale Datenbank MySQL[6]. Bilddaten und Daten der Registry werden in MongoDB[7] gespeichert. Die genaue Struktur der Datenhaltung wird in den Abschnitten über die Bestandteile erklärt.

Durch die Data Registry stellen Data Lagoon und Data Lake eine REST-API bereit, die Endpunkte zur Registrierung beinhalten. Die Data Registry ist Teil der Data Lagoon und des Data Lakes. Um das Ansprechen der Endpunkte zu ermöglichen, werden Data Lagoon und Data Lake auf einer Instanz von Apache Tomcat[8] bereitgestellt. Apache Tomcat ist ein Webserver, der Java-basierte Servlets ausführt.

Die Logik der einzelnen Bestandteile und der Flink-Jobs werden in Java 8[9] mit Hilfe einiger Bibliotheken implementiert. Diese umfassen unter anderem Bibliotheken für die Verbindung zu Kafka-Broker, Datenbank-Instanzen und für die Implementierung von Flink-Jobs. Auf die jeweiligen verwendeten Bibliotheken wird in späteren Abschnitten genauer eingegangen. Für das Build-Management wird Apache Maven[10] verwendet.

7.2.2. Datengenerator

Der Datengenerator ist für die Generierung von Beispieldaten zuständig. Diese Beispieldaten werden an die Data-Lake-Umgebung gesendet und dort verarbeitet. Wie in Abschnitt 7.1 beschrieben, soll im Rahmen des Anwendungsfalls eine Verarbeitung von Geschwindigkeitsdaten, Bilddaten aus Kameras und Wetterdaten möglich sein. Für jede Datenquelle wird ein eigener Kafka-Producer implementiert, der Nachrichten an Topics der Data-Lake-Umgebung sendet. Ein Kafka-Producer ist die Verbindung eines Datenproduzenten zu einem Topic, um Nachrichten veröffentlichen zu können. Um JSON-Objekte über Kafka versenden zu können, müssen diese serialisiert werden. Aus diesem Grund wurde ein Serializer für `JsonNode`[11] implementiert. In den folgenden Abschnitten wird auf die generierten Daten eingegangen und dabei jeweils der Aufbau der gesendeten Nachricht erläutert. Diese simulierten Daten sind stark vereinfacht und dienen lediglich dazu, die Funktionalitäten der Data-Lake-Umgebung zu demonstrieren. In der realen Welt wären die Daten einer solchen Nachricht viel umfangreicher und präziser.

[5] InfluxDB: https://www.influxdata.com/
[6] MySQL: https://www.mysql.com/de/
[7] MongoDB: https://www.mongodb.com/de-de
[8] Apache Tomcat: http://tomcat.apache.org/
[9] Java 8: https://docs.oracle.com/javase/8/docs/
[10] Apache Maven: https://maven.apache.org/
[11] FasterXML Github: https://github.com/FasterXML/jackson

Listing 7.1 Nachricht mit Geschwindigkeitsdaten des Datengenerators

```
{
  "timestamp":"21-10-15 12:00:17",
  "car":"DeLorean DMC-12",
  "sensor_value":"140",
  "metadata":{
    "location":"town",
    "unit":"kmh",
    "manufacturer":"DeLorean"
  }
}
```

Geschwindigkeitsdaten

Die Geschwindigkeitsdaten des Datengenerators sollen die Daten des Geschwindigkeitssensors im Beispielszenario simulieren. Dafür erstellt der Datengenerators ein JSON-Objekt bestehend aus Messwert, Fahrzeug, Zeitstempel und Metadaten. Das JSON-Objekt wird mit Hilfe eines Kafka-Producers an ein Topic gesendet. Dieses Topic wurde von der Data Registry bei Registrierung der Geschwindigkeitsdaten zurückgegeben. Die gesendete JSON-Nachricht ist in Listing 7.1 aufgezeigt. Der Messwert (`sensor_value`) ist ein zufälliger Wert zwischen 0 und 150. Dieser repräsentiert die Geschwindigkeit des Connected Cars zu dem Zeitpunkt des Zeitstempels (`timestamp`). Beim Fahrzeug (`car`) wird zufällig aus einer Liste mit verschiedenen Fahrzeugnamen ausgewählt. Hierbei handelt es sich um das Connected Car, das den Geschwindigkeitswert gesendet hat. Um die Verarbeitung einiger Metadaten zu zeigen, sind einige grundlegende Beispiele gewählt worden. Zuerst wird der Ort (`location`) der Geschwindigkeitsaufname in den Metadaten angegeben. Dieser wird vereinfacht zufällig zwischen Landstraße und Stadt ausgewählt. Zwar könnte ein Global Positioning System (GPS)-Wert verwendet werden, aber für Demonstrationszwecke innerhalb des Beispielszenarios reicht diese Unterteilung aus. Weitere Metadaten sind Einheit der Messung (`unit`) und Hersteller (`manufacturer`). Die Generierung einer Nachricht wird jeweils in einem zufälligen Abstand zwischen einer und 2000 Millisekunden erneut angestoßen.

Bilddaten

Für die Simulation der Bilddaten einer Kamera eines Connected Cars wurde ein Teil des Datensatzes von Maddern et al. [MPLN17] genutzt. Zur Sammlung dieses Datensatzes wurde in einem Zeitraum von sechs Monate eine Strecke in Oxford abgefahren. Hierfür ist ein Connected Car verwendet worden, das unter anderem mit Kameras ausgestattet war. Dabei wurden Daten in unterschiedliche Wetterlagen, bei verschiedenen Straßenarbeiten und mit unterschiedlicher Dichte an Passanten gesammelt. Der Datengenerator nutzt hiervon Kameraaufnahmen eines Beispielausschnitts der Daten, der auf der offiziellen Webseite[12] zur Verfügung gestellt wird.

[12] Bilddaten: https://robotcar-dataset.robots.ox.ac.uk/downloads/

Listing 7.2 Nachricht mit Bilddaten des Datengenerators

```
{
  "timestamp":"03-07-15 12:00:17",
  "car":"Impala67",
  "image_name":"picture-street-0502.png",
  "image":"Base64 string",
  "metadata":{
    "location":"town",
    "format":"png",
    "content":{
      "cars":[
        {
          "id":"car01",
          "coordinates":"front"
        },
        {
          "id":"car02",
          "coordinates":"behind"
        }
      ],
      "pedestrians":[
        {
          "id":"pedestrian01",
          "coordinates":"left"
        }
      ],
      "traffic":[
        {
          "id":"shield01",
          "coordinates":"front"
        },
        {
          "id":"light01",
          "coordinates":"right"
        }
      ]
    },
    "manufacturer":"Chevrolet"
  }
}
```

Die Nachrichten, die der Datengenerator erstellt, bestehen aus einem Bild, dem Fahrzeug, einem Zeitstempel und Metadaten. Analog zu den Geschwindigkeitsdaten wird ein JSON-Objekt erstellt und mit Hilfe eines Kafka-Producers an ein Topic gesendet. Auch dieses Topic wurde nach der Registrierung der Bilddaten von der Data Registry zurückgegeben. Der genaue Aufbau der Nachricht über die Bilddaten ist in Listing 7.2 dargestellt. Das gesendete Bild wird zufällig aus einem Ordner, der die Beispielaufnahmen enthält, ausgewählt. Außerdem wird der Bildname (`image_name`) für die Nachricht übernommen. Um das Bild in einem JSON-Objekt senden zu können, wird es mit Base64 kodiert (`image`). Dies reicht im Rahmen des Prototypen aus. Sollte die Implementierung

Listing 7.3 Nachricht mit Wetterdaten des Datengenerators

```
{
  "location_name":"Stuttgart",
  "timestamp":"1627478568",
  "last_updated":"1627478100",
  "temperature":"22.0",
  "condition":"Partly cloudy",
  "wind":"0.0"
}
```

jedoch in größeren Rahmen genutzt werden, sollten Bilder z.B. in einen S3-Bucket[13] geladen werden. Statt dem codierten Bild wird der Pfad zum Bild im S3-Bucket mitgeschickt. Dadurch würde sich die Nachrichtengröße erheblich verringern, was zusätzlich hilft eine geringere Latenz beizubehalten. Da keine Metadatenextraktion für die Analyse des Inhalts der Aufnahme erfolgt, wird der Bildinhalt als Teil der Nachricht mitgeschickt. Die Bildinhalte könnten z.B. durch Deep-Learning-Techniken analysiert werden. Dies liegt jedoch außerhalb des Umfangs dieser Arbeit, weshalb einige Informationen zum Inhalt eines Bildes (`content`) zufällig generiert werden. Diese Informationen umfassen Anzahl und Koordinaten von Objekten im Verkehr, Fußgängern und anderen Fahrzeugen. Weitere Metadaten sind Hersteller (`manufacturer`) und Ort der Aufnahme (`location`), die analog zu den Geschwindigkeitsdaten generiert werden. Zusätzlich wird das Format (`format`) des Bildes übergeben. Die Nachrichten werden auch hier in einem zufälligen Abstand zwischen einer und 2000 Millisekunden gesendet.

Wetterdaten

Zum Simulieren einer externen Quelle werden Wetterdaten genutzt. Diese Daten werden über die API von WeatherAPI[14] abgefragt. Die Parameter, wie z.B. Ort, können über eine Konfigurationsdatei gesetzt werden. Von den erhaltenen Daten wird nur ein Teil genutzt, um die Nachricht überschaubar zu halten.

Die gesendete JSON-Nachricht besteht aus Ort, Zeitstempel, Zeitstempel des letzten Updates, Temperatur, Beschreibung und Windgeschwindigkeit. Wie zuvor, wird die Nachricht durch einen Kafka-Producer an das Topic, das bei Registierung von der Data Registry zurückgeben wurde, gesendet. Eine Beispielnachricht ist in Listing 7.3 zu sehen. Der Zeitstempel (`timestamp`) repräsentiert die Zeit, zu der die WeatherAPI abgefragt wurde, während `last_updated` den Zeitpunkt des letzten Updates der Daten enthält. Die Temperatur (`temperature`), Beschreibung (`condition`) und Windgeschwindigkeit (`wind`) beschreiben das Wetter zum abgefragten Zeitpunkt am abgefragten Ort (`location_name`). Die Nachrichten dieser Datenquelle werden in einem Abstand von mehreren Minuten gesendet. Außerdem werden die Wetterdaten direkt an den Data Lake gesendet, da es sich um Batch-Daten handelt.

[13] Amazon S3 Bucket: https://docs.aws.amazon.com/de_de/AmazonS3/latest/userguide/UsingBucket.html
[14] WeatherAPI: https://www.weatherapi.com/

Listing 7.4 Resultierende Nachricht der Geschwindigkeitsdaten in der Flink-Verarbeitung der Data Lagoon erweitert um `source`

```
{
  "timestamp":"21-10-15 12:00:17",
  "car":"DeLorean DMC-12",
  "sensor_value":140,
  "metadata":{
    "location":"town",
    "unit":"kmh",
    "source":"carGenerator",
    "manufacturer":"DeLorean"
  }
}
```

7.2.3. Data Lagoon

Um eine Verarbeitung näher an den Connected Cars zu erreichen, wurde in Abschnitt 5.2 das Konzept der Data Lagoon vorgestellt. In diesem Abschnitt wird auf die Implementierung einer solchen Data Lagoon eingegangen. Der Prototyp soll eine Verarbeitung und Speicherung von gesendeten Bild- und Geschwindigkeitsdaten ermöglichen. Außerdem sollen die Daten vorverarbeitet werden können und zum Data Lake gestreamt werden.

Verarbeitung mit Apache Flink

Zur Verarbeitung der ankommenden Stream-Daten wird ein Flink-Job implementiert. Dieser liest die Nachrichten aus den Topics von Geschwindigkeits- und Bilddaten aus, bereitet Daten für die Speicherung in der Data Lagoon vor und leitet Daten an die Stream-Verarbeitung des Data Lakes weiter. Der Flink-Job nutzt die `DataStreamAPI`, wofür die Flink-Streaming Bibliothek importiert wurde. Nachdem die ankommenden Daten des Streams verarbeitet wurden, muss eine Datensenke angegeben werden, zu der Flink den Stream leitet. Flink stellt einige Konnektoren für Message-Queues wie Kafka und etablierte Datenbanken zur Verfügung. Da keine Konnektoren für InfluxDB und MongoDB bereitgestellt werden, sollen die verarbeiteten Streams über Kafka-Topics zur Datenhaltung der Data Lagoon geleitet werden. Somit wird der Flink-Job gleichzeitig unabhängig von den eigentlich verwendeten Datenbanken. Der Aufbau der resultierenden Nachrichten ist bei der Weiterleitung an den Data Lake und die Datenhaltung der Data Lagoon identisch. Eine Beispielnachricht der Geschwindigkeitsdaten ist in Listing 7.4 und der Bilddaten in Listing 7.5 dargestellt.

Bei den Geschwindigkeitsdaten werden alle Daten herausgefiltert, bei denen der Geschwindigkeitswert 0 beträgt. Dazu wird die `filter`-Funktion der DataStreamAPI genutzt. Danach werden die Daten an eine Kafka-Senke weitergegeben. Diese ist mit dem Topic, das die Geschwindigkeitsdaten zur Datenhaltung der Data Lagoon transportiert, verbunden. Auf den Bilddaten wird keine Operation ausgeführt. Sie werden direkt an die Kafka-Senke des Topics, das Bilddaten zur Datenhaltung der

Listing 7.5 Resultierende Nachricht der Bilddaten in der Flink-Verarbeitung der Data Lagoon

```
{
  "timestamp":"03-07-15 12:00:17",
  "car":"Impala67",
  "image_name":"picture-street-0502.png",
  "image":"Base64 string",
  "metadata":{
    "location":"town",
    "format":"png",
    "content":{
      "cars":[
        {
          "id":"car01",
          "coordinates":"front"
        },
        {
          "id":"car02",
          "coordinates":"behind"
        }
      ],
      "pedestrians":[
        {
          "id":"pedestrian01",
          "coordinates":"left"
        }
      ],
      "traffic":[
        {
          "id":"shield01",
          "coordinates":"front"
        },
        {
          "id":"light01",
          "coordinates":"right"
        }
      ]
    },
    "source":"carGenerator",
    "manufacturer":"Chevrolet"
  }
}
```

Data Lagoon sendet, geleitet. Bei beiden Quellen werden die Metadaten innerhalb der Nachricht um einen `source`-Tag, der die originale Herkunft der Daten enthält, erweitert. Somit können bei der Verarbeitung zusätzliche Metadaten hinzugefügt werden.

Parallel werden die ankommenden Daten für die Weiterleitung zum Data Lake vorbereitet. Die Bilddaten werden ohne weitere Operationen weitergeleitet. Die Geschwindigkeitsdaten werden anhand ihres Geschwindigkeitswertes gefiltert. Nur Datensätze mit einer Geschwindigkeit höher 30 werden an den Data Lake gesendet, wobei die `filter`-Funktion der DataStreamAPI angewendet wurde. Dadurch wird gezeigt, dass eine Filterung der Daten, die zum Data Lake gesendet werden, möglich ist. Statt einer Filterung, wäre es auch möglich Daten bestimmter Quellen nicht weiterzuleiten. Somit wären diese ausschließlich in der Data Lagoon gespeichert. Genauso können komplexere Schritte zur Vorverarbeitung der Daten angewendet werden, wie z.B. Verschleierung der Werte oder Verrechnen mit anderen Quellen. Auch hier wird den Metadaten ein `source`-Tag hinzugefügt und an die jeweilige Kafka-Senke übergeben.

Speicherung in der Datenhaltung

Die Logik der Datenhaltung der Data Lagoon ist in einer Applikation gekapselt. Die eigentlichen Datenbanken liegen auf Ubuntu-Instanzen im Openstack, wobei die Verbindung zu diesen über ihre IP-Adressen und entsprechende Bibliotheken erfolgt. Die Applikation, welche die Datenspeicherung verwaltet, importiert die benötigten Treiber für MySQL, MongoDB und InfluxDB über Maven.

Die Applikation implementiert jeweils einen Kafka-Consumer für Daten, die in die InfluxDB, MongoDB und MySQL gespeichert werden sollen. Die Kafka-Consumer lesen die Topics, welche die vorverarbeiteten Daten aus der Flink-Verarbeitung erhalten, aus. Dafür wird bei der Registrierung der Datenquelle ein Thread mit dem Kafka-Consumer, der auf das jeweilige Topic hört, gestartet und der übergebene Name der Quelle gemerkt. Anhand des Namens der Quelle werden bei Erstellung des Kafka-Consumer Informationen zur Quelle aus der Datenbank der Registry abgefragt. Diese Informationen beinhalten unter anderem den Namen der Quelle, welche Felder der Nachricht Zeitstempel, Metadaten und Werte enthalten und die Namen der jeweiligen Datenbankstrukturen, die bei Registrierung angelegt wurden. In Abschnitt 5.4 wird genauer auf den Aufbau und Ablauf der Registry eingegangen. Für das Speichern der erhaltenen Daten nutzt der Consumer Informationen aus der Registry. Der jeweilige Datensatz wird dadurch in die passende Datenbank geschrieben.

Für die Geschwindigkeitsdaten wurde bei Registierung ein Influx-Measurement namens `sensor_measurement` erstellt, da es sich um Zeitreihendaten handelt. Ein *Measurement* enthält alle Datensätze zu einer bestimmten Zeitreihe innerhalb eines Influx-Buckets. *Buckets* entsprechen einer Datenbank einer Influx-Instanz. Beim Aufsetzen der InfluxDB wurde ein Bucket angelegt, in dem die Data Lagoon ihre Daten speichern kann. Aus der Nachricht der Geschwindigkeitsdaten wird jeweils der Zeitstempel, der Fahrzeugname und der Sensorwert herausgezogen. Diese Informationen werden als Zeitpunkt in `sensor_measurement` gespeichert.

Die Bilddaten werden in einer MongoDB-Collection namens `carImages` gespeichert. Auch diese *Collection* wurde bei Registrierung der Bilddatenquelle erstellt. Eine Collection ist eine Substruktur innerhalb einer MongoDB-Datenbank und speichert Daten in Form von *Documents*. Analog zum Bucket der InfluxDB wurde im Voraus eine Datenbank für die Daten der Data Lagoon bei der

MongoDB-Instanz erstellt. Bei den Bilddaten werden Zeitstempel, Fahrzeugname, Bildname, Bild in Base64 codiert und Bildinhalt aus der Nachricht ausgelesen. Diese Informationen werden als Document in `carImages` gespeichert.

Die Metadaten beider Quellen werden innerhalb einer MySQL-Datenbank in Tabellen für diese gespeichert. Die Tabellen sind nach dem Quellennamen mit dem Suffix „Metadata“ benannt. Dabei stammen die Spaltennamen aus Informationen der Registry-Datenbank. Auch diese Tabellen wurden bei der Registrierung der Quellen erstellt. Zusätzliche Informationen zu den Metadaten folgen in Abschnitt 7.2.5, da die Speicherung von Metadaten bei Data Lagoon und Data Lake auf dem gleichen Prinzip basiert. Sowohl die Geschwindigkeitsdaten als auch die Bilddaten lassen sich innerhalb der verschiedenen Datenbanken anhand von Zeitstempeln und Fahrzeugnamen zuordnen.

7.2.4. Data Lake

Die Daten aus der Data Lagoon werden zum Data Lake weitergeleitet. Dieser verarbeitet zum einen Daten der Data Lagoons und zum anderen Daten aus externen Quellen. Im Beispielszenario erhält der Data Lake vorgefilterte Geschwindigkeitsdaten und Bilddaten von der Data Lagoon. Außerdem werden im Abstand von mehreren Minuten Wetterdaten durch den Datengenerator gesendet. Im Folgenden wird der Aufbau und die Implementierung des Data Lakes im Beispielszenario erklärt.

Verarbeitung mit Apache Flink

Für die beim Data Lake ankommenden Daten wurde ein Flink-Job implementiert. Dieser liest die Geschwindigkeitsdaten und Bilddaten aus den Topics aus, welche die vorverarbeiteten Daten der Data Lagoon enthalten. Analog zur Data Lagoon werden die erhaltenen Daten für die Speicherung in der Datenhaltung des Data Lakes vorbereitet, wobei weitere Operationen auf den Daten durchgeführt werden. Zur Implementierung wird die Flink-Streaming-Bibliothek importiert und die DataStreamAPI genutzt. Nach der Verarbeitung werden auch hier die Daten nicht direkt in die Datenbanken gespeichert, sondern an eine Kafka-Senke weitergegeben. Diese Kafka-Senke ist mit der Applikation, welche die Datenhaltung des Data Lake verwaltet, verbunden.

Die Bilddaten werden ohne weitere Filterung und Vorverarbeitung über die Kafka-Senke zum Topic, das mit der Datenhaltung des Data Lakes verbunden ist, weitergeleitet. Bei den Geschwindigkeitsdaten wird je nach Wert auf Intervalle gemappt und der Geschwindigkeitswert dahingehend geändert. Dazu wurde die `map`-Funktion der DataStreamAPI genutzt. Dabei werden die Intervalle `30-50`, `51-70`, `71-100`, `101-130` und `>130` berücksichtigt. Somit werden nur Geschwindigkeitsbereiche im Data Lake abgespeichert statt die genauen Geschwindigkeiten. Dies unterstützt unter anderem die Privatheit. Zusätzlich werden die Metadaten um die durchgeführte Operation (`operation`) und den originalen Geschwindigkeitswert (`original_value`) erweitert, wodurch eine mögliche Erweiterung der Metadaten demonstriert wird. In einem realen Szenario wäre es aus Privatheitsgründen nicht ratsam den originalen Wert der Geschwindigkeit zu speichern. Die resultierende Nachricht der Geschwindigkeitsdaten ist in Listing 7.6 aufgezeigt. Die Nachricht der Bilddaten entspricht der von der Data Lagoon empfangenen Nachricht (vgl. Listing 7.5). Auch hier sind weitere Verarbeitungsschritte unterschiedlicher Komplexität sowohl bei Geschwindigkeitsdaten als auch bei Bilddaten möglich. Im Anwendungsfall wurde sich lediglich für einfache Verarbeitungen entschieden.

Listing 7.6 Resultierende Nachricht der Geschwindigkeitsdaten in der Flink-Verarbeitung des Data Lakes

```
{
  "timestamp":"21-10-15 12:00:17",
  "car":"DeLorean DMC-12",
  "sensor_value":"100-150",
  "metadata":{
    "location":"town",
    "unit":"kmh",
    "source":"carGenerator",
    "operation":"mapping",
    "original_value":"140",
    "manufacturer":"DeLorean"
  }
}
```

Speicherung in der Datenhaltung

Im Grundprinzip verhält sich die Datenhaltung des Data Lakes analog zur Datenhaltung der Data Lagoon. Die Datenhaltung des Data Lakes ist ebenfalls in einer Applikation gekapselt, welche die Speicherung der Daten verwaltet und zur Verbindung zu den jeweiligen Datenbanken deren Treiber importiert. Die Datenbanken des Data Lakes umfassen InfluxDB, MongoDB und MySQL. Die Annahme der Daten erfolgt über Kafka-Consumer, deren Threads bei Registierung der Datenquellen erstellt und gestartet werden. Die Speicherung der Geschwindigkeits- und Bilddaten erfolgt nach dem gleichen Prinzip wie in der Data Lagoon. Dabei liegen die Datenstrukturen in Datenbanken, die zuvor speziell für den Data Lake angelegt wurden. Die Erstellung der Datenstrukturen innerhalb der Datenbanken, wie Bucket und Collection, findet bei Registrierung der jeweiligen Datenquelle statt. Zur Speicherung werden außerdem die Informationen der Registry berücksichtigt.

Zusätzlich zu Geschwindigkeits- und Bilddaten werden im Data Lake Wetterdaten konsumiert. Hierzu wird bei der Registierung der Wetterdaten ein Kafka-Consumer für relationale Daten, die in einer MySQL-Tabelle gespeichert werden, erstellt. Die Wetterdaten werden in Batches direkt vom Datengenerator an das Topic gegeben, das mit dem Data Lake verbunden ist. Bei Ankunft der Daten werden diese mit den Informationen aus der Registry-Datenbank in die angegebene MySQL-Tabelle gespeichert. In diesem Fall heißt diese `Weather`. Aus den Nachrichten werden jeweils Zeitstempel, Ort, Zeitstempel des letzten Updates, Temperatur, Beschreibung und Windgeschwindigkeit herausgezogen. Die Spaltennamen wurden bei der Registrierung übergeben und entsprechen den Feldnamen der Nachricht des Datengenerators. Bei den Wetterdaten wurden keine zusätzlichen Metadaten berücksichtigt.

7.2.5. Metadaten

Wie zuvor erläutert, sind Metadaten im Data-Lake-Kontext wichtig, um Daten wiederzufinden oder Daten zu bestimmten Fragestellungen, z.B. alle Bilder mit freien Parkplätzen, zu finden. Deshalb wird in diesem Abschnitt ausführlicher darauf eingegangen, wie Data Lagoon und Data Lake

Metadaten speichern. Im Konzept wird eine Komponente zur Extraktion von Metadaten vorgestellt. Da eine Implementierung dieser über den Umfang der Arbeit hinaus geht, wurde im Prototyp keine explizite Metadatenextraktion umgesetzt. Daher wird davon ausgegangen, dass die Quellen Metadaten in ihren Nachrichten mitsenden. Zukünftig könnten die Prototypen von Data Lagoon und Data Lake durch ihren modularen Aufbau leicht um eine Komponente zur Extraktion der Metadaten erweitert werden. Somit könnte eine Metadatenextraktion vor die Flink-Komponenten geschaltet werden. Ein Beispiel für eine solche Extraktion wäre der Inhalt der Bilddaten.

In der aktuellen Implementierung werden die Metadaten zum Teil zufällig generiert, in der Datenhaltung aus den Nachrichten ausgelesen und in MySQL-Tabellen zur jeweiligen Quelle gespeichert. Bei der Registrierung durch die Data Registry werden die Spaltennamen als Liste übergeben. Die Felder, welche die Metadaten enthalten, müssen denselben Namen besitzen, wie in der Registry angeben. In der Verarbeitung der Nachricht werden Metadaten, mit Hilfe der angegebenen Felder in die jeweilige Metadatentabelle mit Zeitstempel und Fahrzeugnamen gespeichert. Zu den Geschwindigkeitsdaten wird für jede ankommende Nachricht in der Data Lagoon ein Zeitstempel, der Fahrzeugname, der Ort, die Einheit des Messwertes, der Hersteller und die Datenquelle in der Metadatentabelle gespeichert. Im Data Lake wird zusätzlich pro Datensatz die durchgeführte Operation und der Originalmesswert berücksichtigt. Damit soll demonstriert werden, dass eine Erweiterung der Metadaten entlang der Verarbeitungspipline möglich ist. Die Metadaten zu den Bilddaten umfassen sowohl in der Data Lagoon als auch im Data Lake einen Zeitstempel, den Fahrzeugnamen, den Ort, das Dateiformat des Bildes, die Datenquelle und den Hersteller. Somit ist eine Erweiterung der Metadaten entlang der Verabeitungspipline nicht zwingend. Zu den Wetterdaten werden keine zusätzlichen Metadaten gespeichert.

7.2.6. Data Registry

Die Data Registry wird genutzt, um zusätzliche Informationen über Datenquellen zu erhalten. Datenquellen können sich bei der Data Registry registrieren und dann Daten an die Data-Lake-Umgebung senden, ohne Nachrichten zuvor in eine bestimmte Struktur zu transformieren. Stattdessen werden Informationen zur Nachrichtenstruktur und der enthaltenen Daten bei dieser Registrierung abgefragt. Die Data Registry speichert diese Informationen in einer Datenbank und sowohl Data Lagoon als auch Data Lake können diese Datenbank abfragen, um ankommende Nachrichten zu parsen.

Im Prototyp ist die Data Registry als Teil von Data Lagoon und Data Lake realisiert. Beide Datenhaltungen besitzen eine zusätzliche MongoDB für die Daten der Data Registry. Die Applikationen, welche die Datenhaltungen verwalten, stellen jeweils eine REST-API zur Verfügung, die einen Endpunkt zur Registrierung enthält. Der Endpunkt nimmt die Informationen zur Datenquelle entgegen und speichert diese in der Registry-Datenbank ab. Mit dem Namen der Quelle können Data Lagoon und Data Lake jeweils ihre Registry-Datenbank abfragen und die Informationen zum Parsen der ankommenden Nachrichten nutzen. Listing 7.7 zeigt den Endpunkt der Data Registry. Der Aufbau von diesem ist bei Data Lagoon und Data Lake identisch. Für die Registrierung der Datenquelle wird der dargestellte POST-Request bereitgestellt. Dabei wird im Body ein JSON-Objekt übergeben, das Informationen zur Datenquelle enthält. Diese Informationen werden vom Data-Registry-Teil der Data Lagoon bzw. des Data Lakes gespeichert und anhand des `sourceName` identifiziert. Die Antwort der Nachricht besteht aus dem Topic, das schließlich zum Senden der Daten von der Datenquelle genutzt werden kann. Bevor die Data Registry die Informationen über die Datenquellen speichert,

Listing 7.7 Endpunkt für die Registrierung von Datenquellen

```
POST /registry HTTP/1.1
Conent-Type: application/json
  {
    "dataType": "semi-structured",
    "sourceName": "carImage",
    ....
  }

HTTP/1.1 201 CREATED
topic: carImage-data
```

legt sie innerhalb der Datenbanken von Data Lagoon bzw. Data Lake die entsprechend benötigten Datenbankstrukturen zum Speichern zukünftiger Daten der Quelle an. Daraufhin wird ein Topic für die jeweilige Datenquelle im Kafka-Broker kreiert und schließlich anhand des Quellennamens und dem erstellten Topic ein Kafka-Consumer erstellt. Die Data Registry startet einen Thread, in dem der Kafka-Consumer auf das Topic hört und die ankommenden Daten entsprechend der in Abschnitt 7.2.3 und Abschnitt 7.2.4 vorgestellten Vorgehensweise verarbeitet. Dies läuft in Data Lagoon und Data Lake identisch ab. Die Data Lagoon besitzt lediglich zusätzlich die Möglichkeit die Informationen der Data Registry direkt zum Data Lake weiterzuleiten.

Um Datenbankstrukturen anzulegen und ankommende Nachrichten parsen zu können, benötigt die Data Registry einige Informationen über die Datenquelle. Diese werden im Body des Endpunktes als JSON-Objekt mitgegeben. Wie dieses genau aussieht ist im Anhang A in Form von JSON-Schemata zu finden. Anhang A.1 enthält das Schema der Data Registry in der Data Lagoon und Anhang A.2 das Schema des Data Lakes. Im Anhang werden außerdem alle Informationen aufgelistet, die in diesen JSON-Objekten erhalten sein müssen. Im Folgenden werden diese benötigten Informationen an den Nachrichten zur Registrierung der Datenquellen des Prototypen genauer erklärt.

Für die Registrierung der Geschwindigkeitsdaten wird angegeben, dass es sich um Zeitreihendaten handelt. Außerdem werden die Metadaten definiert, die in der Data Lagoon hinterlegt werden sollen. Da sich die gespeicherten Metadaten bei Data Lagoon und Data Lake unterscheiden, muss eine Registrierung bei beiden durchgeführt werden. Deshalb wird bei der Nachricht zur Data Lagoon `saveToDataLake` auf `false` gesetzt. Die Nachricht zur Data Lagoon wird in Listing 7.8 und zum Data Lake in Listing 7.9 aufgezeigt. Weiterhin wird bei beiden Nachrichten angegeben, in welchen Feldern sich der Sensorwert (`valueField`) und der Zeitstempel (`timestampField`) befinden und wie das Measurement in der InfluxDB heißen soll (`measurementName`). Durch die Angabe des Datentyps als Zeitreihendaten wissen sowohl die Data Registry von Data Lagoon als auch von Data Lake, dass die Datenbankstrukturen in der InfluxDB angelegt werden sollen. Außerdem können Data Lagoon und Data Lake diese Informationen nutzen, um ankommende Geschwindigkeitsdaten in die InfluxDB zu speichern. Dazu dient ebenfalls die Angabe der Felder für Sensorwert und Zeitstempel sowie welches Feld den Fahrzeugnamen (`dataLabelField`) enthält. Für die Metadaten werden jeweils Tabellen innerhalb der MySQL-Datenbank angelegt, deren Spaltenname der mitgegebenen Liste von Metadaten entsprechen. Diese Bezeichnungen sollten bei zukünftigen Nachrichten dieser Liste entsprechen, um eine Zuordnung der Metadaten zu ermöglichen. Nachdem alle zuvor beschriebenen

Listing 7.8 Beispiel einer JSON-Nachricht zur Registrierung der Geschwindigkeitsdaten bei der Data Lagoon

```
{
  "dataType": "timeSeries",
  "sourceName": "sensorMeasurement",
  "metadata": [
    "location",
    "unit",
    "manufacturer",
    "source"
  ],
  "saveToDataLake": false,
  "dataLabelField": "car",
  "timeSeries": {
    "valueField": "sensor_value",
    "timestampField": "timestamp",
    "measurementName": "sensor_measurement"
  }
}
```

Listing 7.9 Beispiel einer JSON-Nachricht zur Registrierung der Geschwindigkeitsdaten beim Data Lake

```
{
  "dataType": "timeSeries",
  "sourceName": "sensorMeasurement",
  "metadata": [
    "location",
    "unit",
    "manufacturer",
    "source",
    "operation",
    "original_value"
  ],
  "dataLabelField": "car",
  "timeSeries": {
    "valueField": "sensor_value",
    "timestampField": "timestamp",
    "measurementName": "sensor_measurement"
  }
}
```

Listing 7.10 Beispiel einer JSON-Nachricht zur Registrierung der Bilddaten bei der Data Lagoon

```
{
  "dataType": "semi-structured",
  "sourceName": "carImage",
  "metadata": [
    "location",
    "format",
    "source",
    "manufacturer"
  ],
  "saveToDataLake": true,
  "dataLabelField": "car",
  "otherStructure": {
    "collectionName": "images",
    "idField": "timestamp"
  }
}
```

Schritte zur Erstellung der Datenstrukturen und Kafka-Consumer abgeschlossen sind, werden die erhaltenen Informationen in die MongoDB der Data Lagoon und des Data Lakes gespeichert. So können diese zukünftig zum Parsen der Nachrichten genutzt werden.

Die Bilddaten werden nur bei der Data Lagoon registriert, da die zu berücksichtigenden Informationen bei Data Lake und Data Lagoon identisch sind. Aus diesem Grund wird `saveToDataLake` auf `true` gesetzt, wodurch die Data Registry der Data Lagoon die Informationen automatisch zum Data Lake weiterleitet. Dies erspart dem Nutzer bei identischen Strukturen eine extra Registrierung beim Data Lake. Die Nachricht hierzu ist in Listing 7.10 dargestellt. Hier wird der Datentyp als semi-strukturierte Daten angeben, was intern eine Erstellung einer Collection bei der MongoDB veranlasst. Bei semi-strukturierten Daten muss außerdem ein Feld, das innerhalb der Collection als ID genutzt werden kann (`idField`), sowie ein Name für die Collection (`collectionName`) angegeben werden. Der Rest ist äquivalent zu den Geschwindigkeitsdaten.

Die Wetterdaten werden schließlich als relationale Daten beim Data Lake registriert. Die Nachricht hierfür ist in Listing 7.11 dargestellt. Für die Erstellung der benötigten Tabelle zur Speicherung von Wetterdaten innerhalb der MySQL-Datenbank des Data Lakes werden die Informationen aus `tables` genutzt. Hier wird angegeben, welche Tabellen für die Daten der Datenquelle genutzt werden sollen. Die Felder, die diese Daten in künftigen Nachrichten enthalten, besitzen die gleiche Bezeichnung wie in der `columns`-Liste angegeben. So kann der Data Lake ankommende Nachrichten der Quellen parsen und die Daten in die jeweilige Tabellenspalte speichern. Der restliche Ablauf der Registrierung ist äquivalent zu den Vorherigen.

7.2.7. Alternative Tools

Zusätzlich zu den im Prototyp verwendeten Tools, wurden noch zwei weitere Tools betrachtet. Zum einen wurde Apache Samza als eine alternative Processing-Engine für Data Lagoon und Data Lake getestet. Für die Implementierung des Prototyps fiel die Entscheidung auf Flink, da hier eine

Listing 7.11 Beispiel einer JSON-Nachricht zur Registrierung der Wetterdaten beim Data Lake

```
{
  "dataType": "relational",
  "sourceName": "weatherAPI",
  "dataLabelField": "location",
  "tables": [
  {
    "tableName": "weather",
    "columns": [
        "timestamp",
        "location",
        "last_updated",
        "temperature",
        "condition_text",
        "wind"
      ]
    }
  ]
}
```

bessere Unterstützung für verschiedene Datenbanken und eine größere Operationsauswahl geboten werden. Samza ist ein relativ neues Projekt unter Apache. Daher ist vor allem interessant, in wie weit eine Verarbeitung mit diesem Tool umsetzbar ist. Dafür wurde ein Teil der Verarbeitungsschritte der Stream-Daten des Data Lakes in Samza umgesetzt. Zum anderen wurde ein mögliches Metadatenmanagement mit Apache Atlas evaluiert. Eine Metadatenmanagement mit Atlas ist nicht nötig für die Kernfunktionalität, weshalb es nicht direkt Teil des Prototyps ist. Jedoch ist eine Betrachtung von möglichen Tools für ein zukünftiges Metadatenmanagement wichtig, da bei wachsender Datenmenge ein solches unausweichlich wird. Daher wurde die Verarbeitungspipline der Geschwindigkeitsdaten vom Datengenerator zur Datenhaltung des Data Lakes in Atlas modelliert. In den folgenden Abschnitten wird auf beide Tools eingegangen.

Streamverabeitung mit Apache Samza

Apache Samza ist wie Apache Flink ein Framework für die Bearbeitung von Datenstreams. Jobs in Samza können entweder als YARN-Jobs oder mit eingebetteter Bibliothek implementiert werden. Apache Hadoop YARN ist ein Ressourcen-Management- und Job-Scheduling-Werkzeug für Hadoop-Umgebungen. Da im Beispielszenario kein Hadoop genutzt wurde, fiel bei der Implementierung die Wahl auf die eingebettete Variante entschieden. Dabei werden über Maven die benötigten Samza-Bibliotheken importiert und entsprechend implementiert.

Zur Demonstration von Samza wurde ein Teil der Verarbeitung der Geschwindigkeitsdaten umgesetzt. Dieser Teil der Verarbeitung reicht aus, um zu zeigen, dass die Umsetzung der Stream-Verarbeitung innerhalb des Prototyps auch mit Samza möglich wäre. Samza erhält Daten in Form eines *InputStream*. Auf diesem werden Operationen ausgeführt und als *OutputStream* zur Datensenke geleitet. Im implementierten Beispiel enthält der InputStream das Kafka-Topic mit den Geschwindigkeitsdaten des Datengenerators. In der Verarbeitung werden zunächst die Geschwindigkeiten, die über dem

Wert 30 liegen, herausgefiltert. Dazu wird die `filter`-Funktion auf den Stream angewendet. Danach werden analog zum Flink-Job des Data Lakes die Geschwindigkeitsdaten auf Intervalle gemappt (vgl. Abschnitt 7.2.4). Schließlich werden die Metadaten um die letzte durchgeführte Operation und den Originalwert erweitert. Der verarbeitete Stream wird als OutputStream an eine Datensenke übergeben, die mit dem Kafka-Topic der Datenhaltung des Data Lakes verbunden ist. Somit ist eine Implementierung der Stream-Verarbeitung mit Apache Samza möglich und die Umsetzung des Anwendungsfalls unabhängig von dem verwendeten Stream-Processing-Framework realisierbar.

Samza bietet weniger direkt einsatzfähige Unterstützung für verschiedene Datensenken als Flink, wobei Samza das deutlich jüngere und kleinere Projekt ist. Die Basisoperationen, die auf Streams ausgeführt werden können, stellt Samza ebenfalls bereit. Flink hat einige zusätzliche Operationen. Jedoch können diese mit etwas Implementierungsarbeit auch mit Samza umgesetzt werden, da Samza die Möglichkeit bietet eigene Funktionen zu realisieren.

Metadata Governance mit Apache Atlas

Apache Atlas unterstützt die Integration von Metadaten aus vielen Quellen von Haus aus. Darunter fallen z.B. Hive, Kafka und Storm. Für diese werden zusätzlich *Hooks* bereitgestellt, die in Echtzeit über Änderungen in der Quelle benachrichtigen können. Außerdem können Metadaten über die API und Message-Queues integriert werden. Weiterhin stellt Atlas ein User Interface (UI) zur Verfügung, das zur Exploration und Administration der Metadaten genutzt werden kann. Bei der Nutzung von Atlas sollte jedoch bedacht werden, dass es nicht geeignet ist, um Metadaten zu einzelnen Datensätzen, wie z.B. Inhalt eines Bildes, zu verwalten. Atlas kann zur Verwaltung von Metadaten über die gesamte Datenhaltung genutzt werden, d.h. es werden Metadaten über die Komponenten gesammelt. Dies sind z.B. Metadaten über ein Kafka-Topic oder eine SQL-Tabelle selbst. Dabei können die Änderungen von Metadaten der Strukturen innerhalb des Systems entlang der Verarbeitungpipline dargestellt werden.

Zu Demonstrationszwecken wurde eine Atlas-Instanz[15] mit Docker[16] aufgesetzt und die Verarbeitungspipline der Geschwindigkeitsdaten vom Datengenerator zu Metadatentabelle und Influx-Bucket abgebildet. Die benötigten Definitionen von *Entities* und *Types* wurden über die API gesendet. Atlas arbeitet mit Types zu denen Entities und *Processes* erstellt werden können. Von Haus aus stellt Atlas eine Reihe an vordefinierten Types für z.B. Kafka und relationale Datenbanksysteme bereit. Weitere Types können über die API definiert werden. Für den Prototypen wurden zusätzlich Types für den Datengenerator, Flink-Verarbeitung, Influx-Bucket und Influx-Measurement erstellt. Für die anderen Bestandteile, wie Kafka-Topics, stellt Atlas Types bereit. Danach können die Entities und Processes erstellt werden. Zunächst wird eine Datengenerator-Entity benötigt, die dem Datengenerator des Prototypen entspricht. Diese Entity ist mit einer Entity eines Kafka-Topics zum Senden der Geschwindigkeitsdaten verbunden. Da zwischen zwei Entities immer ein Process liegen muss, wird mit dem Flink-Verarbeitungstype ein Process, der die Verarbeitung der gesendeten Geschwindigkeitsdaten widerspiegelt, erstellt. Von diesem Process ausgehend wird eine weitere Entity des Kafka-Topics erstellt. Dieses repräsentiert das Topic zum Senden der verarbeiteten Geschwindigkeitsdaten zum Data Lake. Die Logik des Data Lakes, welche die Daten aus dem Kafka-Topic entgegen nimmt und

[15] verwendetes Docker-Image: https://hub.docker.com/r/sburn/apache-atlas
[16] Docker: https://www.docker.com/

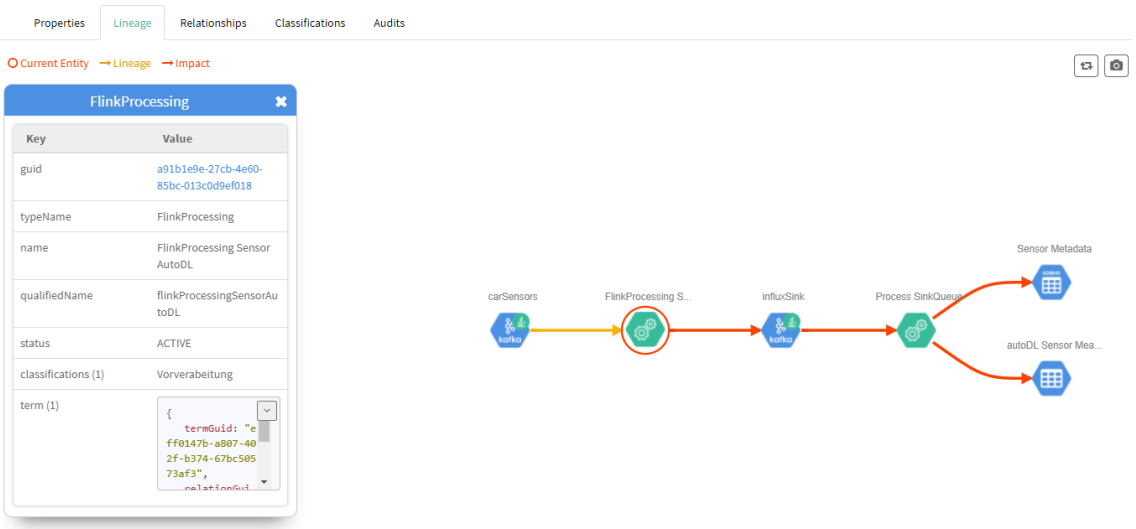

Abbildung 7.3.: Umsetzung der Verarbeitungspipline der Geschwindigkeitsdaten in Apache Atlas

in die jeweiligen Datenbanken speichert, wird wiederum als Process modelliert. Von diesem Process ausgehend wird einerseits eine Entity des Influx-Bucket-Types für die Geschwindigkeitswerte und andererseits eine Entity der SQL-Tabelle für die Metadaten erstellt. Der Influx-Bucket besitzt dabei ein Influx-Measurement zum Speichern dieser Geschwindigkeitsdaten. Die SQL-Tabelle der Metadaten ist eine Sub-Entity einer SQL-Datenbank und SQL-Instanz. Für beide wurde mit vordefinierten Types eine Entity erstellt. In der UI ergibt sich der in Abbildung 7.3 dargestellte Ablauf.

Atlas zeigt wie sich die Daten im Laufe der Verarbeitung an den einzelnen Stationen ändern. Dabei werden die Metadaten über die Bestandteile des Systems gespeichert. Metadaten pro Datensatz werden nicht berücksichtigt. Diese müssen weiterhin zusätzlich verwaltet werden. Das vorliegende Beispiel wurde vor allem mit Hilfe der API-Dokumentation und Blogeinträgen implementiert.

7.2.8. Anmerkungen

Abschließend werden in diesem Abschnitt einige Anmerkungen zum Prototypen und dessen Implementierung gemacht. Besonders soll dabei auf einige Punkte, die für zukünftige Arbeiten am Prototyp hilfreich sein können, eingegangen werden. Der erste Punkt betrifft die Flink-Jobs, die im Prototypen angewendet wurden. Für die Implementierung der Flink-Jobs ist Wissen über die Daten der Quellen nötig, um sie richtig verarbeiten zu können. Daher müssen diese jeweils manuell auf die Datenquelle und die gewünschten Verarbeitungsschritte zugeschnitten werden. Das bedeutet, die Datenvorverarbeitung kann nicht wie die Speicherung der Daten in die Datenhaltung automatisiert werden, da Vorverarbeitungen unzählige Möglichkeiten und Abfolgen beinhalten.

Die weiteren Punkte betreffen das Aufsetzen des Prototyps und Anwenden der Technologien. Zunächst sollte bei Nutzung von Openstack sichergestellt werden, dass alle benötigten Ports in der Freigabeliste eingetragen sind. Nur so kann mit den Technologien, die auf den Instanzen des

Openstacks aufgesetzt wurden, von außen kommuniziert werden. Falls hier trotzdem Probleme entstehen sollten, könnten noch Freigaben der Instanzen durch den Administrator des Openstacks nötig sein.

Generell müssen in allen Konfigurationsdateien der verwendeten Technologien die jeweiligen Einstellungen für Verbindungen von `localhost` zur entsprechenden Internet Protocol (IP)-Adresse der Openstack-Instanz geändert werden. Im Prototyp sind dies die Konfigurationsdateien von Kafka, Zookeeper, Flink, MongoDB, InfluxDB und MySQL. Wenn die Technologien eine Verbindung zu z.B. einem Kafka-Broker über Konfigurationsdateien definieren, muss auch hier die entsprechende IP-Adresse eingetragen werden.

Außerdem sollte bei der Installation durch den Ubuntu-Paketmanager aufgepasst werden. Es kann vorkommen, dass sich die Version und damit auch die Konfiguration zwischen Ubuntu-Paketmanager und der offiziellen Webseite unterscheiden. Dabei können zwischen diesen Versionen sogar Konsolenbefehle unterschiedlich sein. Ein Beispiel wäre Apache Zookeeper. Nach der Installation über den Ubuntu-Paketmanager müssen erst die Konfigurationsdateien gesucht werden und darüber hinaus werden nicht alle Konsolenbefehle unterstützt. Deshalb ist es ratsam die aktuelle Version von der Webseite herunterzuladen und manuell zu installieren. Ein ähnliches Problem tritt bei der Installation von MongoDB auf. Auch bei der Dokumentation und Nutzung von Bibliotheken sollte auf die Versionen geachtet werden. Gerade bei MongoDB und InfluxDB haben sich inzwischen die Bibliotheken und inneren Strukturen grundlegender verändert, sodass bei den Informationsquellen genau darauf geschaut werden sollte, welche Version diese behandeln.

Bei der Konfiguration von Flink müssen nicht nur die IP-Adressen angepasst, sondern sollten auch die Heapgrößen deutlich erhöht werden. Weiterhin sollte die Anzahl der parallel ausgeführten Jobs geändert werden. Der Default ist ein Job. Da im Prototyp beide Jobs auf derselben Flink-Instanz ausgeführt werden, sollte dieser Wert mindestens auf zwei erhöht werden.

8. Zusammenfassung und Ausblick

In Connected-Car-Umgebungen stellen Datenspeicherung und -verarbeitung eine große Herausforderung dar. Nicht nur ist eine Echtzeitverarbeitung der Daten essentiell, sondern es existieren auch viele unterschiedliche Quellen. Zum einen werden Daten aus Connected Cars, RSUs und verschiedenen Services verarbeitet. Zum anderen unterscheiden sich Schnittstellen und Sensorik von Connected Cars je nach Hersteller. Um diese heterogenen Daten für Analysen und Auswertungen zusammenzubringen, ist eine effektive Datenspeicherung nötig. Aus diesem Grund wurde in dieser Arbeit das Konzept eines verteilten Data Lakes erstellt, evaluiert und prototypisch implementiert.

Zunächst wurden einige wichtige Aspekte von Data Lakes, Connected Cars und Intelligent Transport Systems (ITSs) erläutert. Daraufhin wurden verschiedene Systeme betrachtet, die sich mit der Verwendung von Big-Data-Technologien im Verkehrsbereich und Umsetzungen von Data Lakes beschäftigen. Auf Basis der dabei erlangten Erkenntnisse wurde die Idee eines verteilten Data Lakes weiterverfolgt. Mit Hilfe einer Literaturrecherche wurden Anforderungen aufgestellt, die ein Data Lake in einer Connected-Car-Umgebung erfüllen sollte. Eine große Rolle spielen Sicherheit und Privatheit, da in einem ITS viele personenbezogene Daten gesammelt werden. Daher muss besonders darauf geachtet werden, diese entsprechend zu schützen. Weiterhin sind die Unterstützung von heterogenen Datenquellen und eine niedrige Latenz zum Ermöglichen von Echtzeitverarbeitung wichtig. Außerdem sollen Schnittstellen und ein Metadatenmanagement bereitgestellt werden.

Der konzipierte Data Lake verteilt sich auf mehrere Data Lagoons und einem zentralen Data Lake. Data Lagoons sind Teile des Data Lakes, die z.B. auf RSUs gehostet werden können, um die Verarbeitung und Speicherung der Daten näher an die Quelle zu bringen. Dadurch werden Übertragungswege verringert, was schnellere Reaktionen auf Events ermöglicht. Die Data Lagoon leitet die gesammelten Daten an den Data Lake weiter. Dabei können Daten auch ausschließlich in der Data Lagoon gespeichert oder nur in vorverarbeiteter Form weitergeleitet werden. Dies kann sich positiv auf Sicherheit und Privatheit auswirken. Die Kommunikation der Komponenten erfolgt über Message-Queues, wodurch Daten unterschiedlicher Formate und Strukturen gesendet werden können. Data Lagoons basieren auf der Kappa-Architektur, da bei ihnen durch die Nähe zum Connected Car der Fokus auf Stream-Daten liegt. Somit besitzen sie eine Stream-Verarbeitung. Beim Data Lake liegt die Lambda-Architektur zu Grunde. Hier werden zum einen Stream-Daten von den Data Lagoons und zum anderen Daten aus externen Quellen verarbeitet. Daher sind eine Stream-Verarbeitung und Batch-Verarbeitung sinnvoll. Sowohl Data Lagoon als auch Data Lake stellen Schnittstellen zum Sammeln und Abfragen der Daten bereit. Die Datenhaltung beider Komponenten besteht jeweils aus Datenbanken, die für unterschiedliche Datenformate und -arten geeignet sind. Um Informationen über Datenquellen zu verwalten, wird eine Registry eingeführt. Diese Informationen werden von den Data Lagoons und dem Data Lake zum Parsen und Verarbeiten von ankommenden Nachrichten genutzt. Schließlich wird das Konzept gegen die zuvor aufgestellten Anforderungen evaluiert.

Um das Konzept zu testen, wurde eine prototypische Implementierung angefertigt. Dazu wurde zunächst ein Anwendungsfall definiert, den der Prototyp umsetzten soll. Der Prototyp besteht aus Implementierungen einer Data Lagoon, eines Data Lakes und einer Data Registry. Zur Simulation verschiedener Datenquellen wird ein Datengenerator genutzt. Die einzelnen Komponenten kommunizieren über Apache Kafka. Sowohl bei der Data Lagoon als auch beim Data Lake setzt sich die Datenhaltung aus einer InfluxDB, einer MongoDB und einer MySQL-Datenbank zusammen. Außerdem wird für die Verarbeitung Apache Flink verwendet. Bei der Data Lagoon werden ankommende Daten gefiltert und zum Data Lake weitergeleitet. Gleichzeitig werden die Daten in der Datenhaltung der Data Lagoon gespeichert. Der Data Lake erhält Daten von der Data Lagoon und einem externen Wetterdienst. Auf diesen Daten werden Operationen durchgeführt und deren Ergebnisse in der Datenhaltung gespeichert. Durch einen REST-Endpunkt können sich Quellen mit Informationen über Nachrichtenstruktur und gesendeten Daten registrieren. Dieser Endpunkt stellt die Registry dar. Die Informationen werden genutzt, um erhaltene Nachrichten zu parsen und die Daten entsprechend verarbeiten zu können. Weiterhin können Quellen Metadaten mitsenden. In den Flink-Jobs von Data Lagoon und Data Lake werden diese Metadaten zum Teil erweitert. Auch diese Metadaten werden bei der Speicherung berücksichtigt. Abschließend wurden zusätzlich Apache Samza und Apache Atlas als alternative bzw. zusätzliche Tools vorgestellt.

Im aktuellen Konzept werden zwar Metadaten zu den einzelnen Datensätzen berücksichtigt, aber ein Metadatenmanagement basierend auf einem Metadatenmodell wurde nicht konzipiert. Daher sollte zukünftig eine Anwendung eines Metadatenmodells genauer betrachtet werden. Ein ausgereiftes Metadatenmanagement hilft bei Analysen relevante Daten zu finden. Weiterhin sind die Schnittstellen zur Abfrage des Data Lakes und der Data Lagoon nicht genau definiert. Hier sollte beschrieben werden, wie mögliche APIs von Data Lagoon und Data Lake aussehen könnten. Ebenso wurden in der Implementierung des Konzepts die Komponenten zur Metadatenextraktion nicht berücksichtigt. Die Metadaten zu z.B. Bildinhalten werden aktuell direkt vom Datengenerator mitgesendet. Daher sollten mögliche Ansätze zur Analyse der Daten hinsichtlich Metadaten in zukünftigen Arbeiten evaluiert werden.

Literaturverzeichnis

[BAG15] T. Bécsi, S. Aradi, P. Gáspár. „Security issues and vulnerabilities in connected car systems“. In: *2015 International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS)*. 2015, S. 477–482. DOI: 10.1109/MTITS.2015.7223297 (zitiert auf S. 30).

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil. „A Critique of ANSI SQL Isolation Levels“. In: *SIGMOD Rec.* 24.2 (Mai 1995), S. 1–10. ISSN: 0163-5808. DOI: 10.1145/568271.223785. URL: https://doi.org/10.1145/568271.223785 (zitiert auf S. 29).

[CAYM15] P. Carsten, T. R. Andel, M. Yampolskiy, J. T. McDonald. „In-Vehicle Networks: Attacks, Vulnerabilities, and Proposed Solutions“. In: *Proceedings of the 10th Annual Cyber and Information Security Research Conference*. CISR '15. Oak Ridge, TN, USA: Association for Computing Machinery, 2015. ISBN: 9781450333450. DOI: 10.1145/2746266.2746267. URL: https://doi.org/10.1145/2746266.2746267 (zitiert auf S. 28, 30).

[CCFC15] H. Chen, X. Chen, L. Fan, C. Chen. „Classified security protection evaluation for vehicle information system“. In: *2015 International Conference on Cyber Security of Smart Cities, Industrial Control System and Communications (SSIC)*. 2015, S. 1–6. DOI: 10.1109/SSIC.2015.7245673 (zitiert auf S. 30).

[CM16] R. Coppola, M. Morisio. „Connected Car: Technologies, Issues, Future Trends“. In: *ACM Comput. Surv.* 49.3 (Okt. 2016). ISSN: 0360-0300. DOI: 10.1145/2971482. URL: https://doi.org/10.1145/2971482 (zitiert auf S. 13, 21, 22, 28, 29, 31).

[Cot09] C. D. Cottrill. „Approaches to privacy preservation in intelligent transportation systems and vehicle–infrastructure integration initiative“. In: *Transportation research record* 2129.1 (2009), S. 9–15 (zitiert auf S. 31).

[CY15] R. Casado, M. Younas. „Emerging trends and technologies in big data processing“. In: *Concurrency and Computation: Practice and Experience* 27.8 (2015), S. 2078–2091 (zitiert auf S. 17, 27, 28, 32).

[Dix10] J. Dixon. *Pentaho, Hadoop, and Data Lakes*. 2010. URL: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/ (zitiert auf S. 15).

[DJ06] D. Dig, R. Johnson. „How Do APIs Evolve? A Story of Refactoring: Research Articles“. In: *J. Softw. Maint. Evol.* 18.2 (2006), S. 83–107. ISSN: 1532-060X (zitiert auf S. 33).

[EGG+20] R. Eichler, C. Giebler, C. Gröger, H. Schwarz, B. Mitschang. „HANDLE - A Generic Metadata Model for Data Lakes“. In: *Big Data Analytics and Knowledge Discovery*. Hrsg. von M. Song, I.-Y. Song, G. Kotsis, A. M. Tjoa, I. Khalil. Cham: Springer International Publishing, 2020, S. 73–88. ISBN: 978-3-030-59065-9 (zitiert auf S. 19).

[Eur16] Europäischen Parlament und Rat. *Verordnung (EU) 2016/679 des Europäischen Parlaments und des Rates vom 27. April 2016 zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung)*. 2016. URL: https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32016R0679&from=DE (zitiert auf S. 28).

[Fan15] H. Fang. „Managing data lakes in big data era: What's a data lake and why has it became popular in data management ecosystem". In: *2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER)*. 2015, S. 820–824. DOI: 10.1109/CYBER.2015.7288049 (zitiert auf S. 13, 15, 16).

[FHB+18] A. C. Franco da Silva, P. Hirmer, U. Breitenbücher, O. Kopp, B. Mitschang. „TDLIoT: A Topic Description Language for the Internet of Things". In: *ICWE 2018: Web Engineering*. Hrsg. von T. Mikkonen, R. Klamma, J. Hernández. Bd. 10845. Lecture Notes in Computer Science (LNCS). Springer Berlin Heidelberg, Mai 2018, S. 333–348. DOI: 10.1007/978-3-319-91662-0_27. URL: http://www2.informatik.uni-stuttgart.de/cgibin/%20NCSTRL/NCSTRL_view.pl?id=INPROC-2018-18&engl= (zitiert auf S. 40).

[GFS+16] G. Guerreiro, P. Figueiras, R. Silva, R. Costa, R. Jardim-Goncalves. „An architecture for big data processing on intelligent transportation systems. An application scenario on highway traffic flows". In: *2016 IEEE 8th International Conference on Intelligent Systems (IS)*. 2016, S. 65–72. DOI: 10.1109/IS.2016.7737393 (zitiert auf S. 25).

[GGH+19] C. Giebler, C. Gröger, E. Hoos, H. Schwarz, B. Mitschang. „Leveraging the Data Lake - Current State and Challenges". In: *Proceedings of the 21st International Conference on Big Data Analytics and Knowledge Discovery (DaWaK 2019)*. 2019. DOI: 10.1007/978-3-030-27520-4_13 (zitiert auf S. 15, 16, 18).

[GGH+20] C. Giebler, C. Gröger, E. Hoos, H. Schwarz, B. Mitschang. „A Zone Reference Model for Enterprise-Grade Data Lake Management". In: *Proceedings of the 24th IEEE Enterprise Computing Conference (EDOC 2020)*. 2020. DOI: https://doi.org/10.1109/EDOC49727.2020.00017 (zitiert auf S. 18, 19, 38).

[GGH+21] C. Giebler, C. Gröger, E. Hoos, R. Eichler, H. Schwarz, B. Mitschang. „ The Data Lake Architecture Framework: A Foundation for Building a Comprehensive Data Lake Architecture". In: *Proceedings der 19. Fachtagung für Datenbanksysteme für Business, Technologie und Web (BTW 2021)*. 2021 (zitiert auf S. 33, 34).

[GH19] C. Gröger, E. Hoos. „Ganzheitliches Metadatenmanagement im Data Lake: Anforderungen, IT-Werkzeuge und Herausforderungen in der Praxis". In: *BTW 2019*. Hrsg. von T. Grust, F. Naumann, A. Böhm, W. Lehner, T. Härder, E. Rahm, A. Heuer, M. Klettke, H. Meyer. Gesellschaft für Informatik, Bonn, 2019, S. 435–452. DOI: 10.18420/btw2019-26 (zitiert auf S. 19, 20, 33).

[GSSM18] C. Giebler, C. Stach, H. Schwarz, B. Mitschang. „BRAID — A Hybrid Processing Architecture for Big Data". In: *Proceedings of the 7th International Conference on Data Science, Technology and Applications*. Hrsg. von J. Bernardino, C. Quix. Bd. 1. DATA 18. Porto: SciTePress, Juni 2018, S. 294–301. ISBN: 978-989-758-318-6. DOI: 10.5220/0006861802940301 (zitiert auf S. 17).

[GZ17a] P. M. Grulich, O. Zukunft. „Bringing Big Data into the Car: Does it Scale?“ In: *2017 International Conference on Big Data Innovations and Applications (Innovate-Data)*. 2017, S. 9–16. DOI: 10.1109/Innovate-Data.2017.14 (zitiert auf S. 22, 25, 27, 32).

[GZ17b] P. M. Grulich, O. Zukunft. „Smart Stream-Based Car Information Systems that Scale: An Experimental Evaluation“. In: *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. 2017, S. 1030–1037. DOI: 10.1109/iThings-GreenCom-CPSCom-SmartData.2017.181 (zitiert auf S. 25, 32–34).

[Inm16] B. Inmon. *Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump*. Technics publications, 2016 (zitiert auf S. 16).

[Int21] S. International. *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. 2021. DOI: https://doi.org/10.4271/J3016_202104. URL: https://www.sae.org/standards/content/j3016_202104/ (zitiert auf S. 21).

[ISO18] ISO. *Road vehicles – Functional safety*. Standard. Geneva, CH: International Organization for Standardization, 2018 (zitiert auf S. 30).

[KMC+13] V. Kumar, S. Mishra, N. Chand et al. „Applications of VANETs: present & future“. In: *Communications and Network* 5.01 (2013), S. 12 (zitiert auf S. 22).

[Kre14] J. Kreps. „Questioning the lambda architecture“. In: *Online article, July* 205 (2014) (zitiert auf S. 18, 34).

[LCZ+14] N. Lu, N. Cheng, N. Zhang, X. Shen, J. W. Mark. „Connected Vehicles: Solutions and Challenges“. In: *IEEE Internet of Things Journal* 1.4 (2014), S. 289–299. DOI: 10.1109/JIOT.2014.2327587 (zitiert auf S. 22).

[LKM+15] A. Luckow, K. Kennedy, F. Manhardt, E. Djerekarov, B. Vorster, A. Apon. „Automotive big data: Applications, workloads and infrastructures“. In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, S. 1201–1210. DOI: 10.1109/BigData.2015.7363874 (zitiert auf S. 22, 23, 28, 29, 33, 34).

[LML15] P. Lawson, B. McPhail, E. Lawton. „The connected car: who is in the driver's seat“. In: *British Columbia Freedom of Information and Privacy Association* 20 (2015) (zitiert auf S. 13, 23, 31).

[LNRT06] M. B. Line, O. Nordland, L. Røstad, I. A. Tøndel. „Safety vs security?“ In: *PSAM Conference, New Orleans, USA*. sn. 2006 (zitiert auf S. 30).

[Mat17] C. Mathis. „Data lakes“. In: *Datenbank-Spektrum* 17.3 (2017), S. 289–293 (zitiert auf S. 13, 15–17, 19, 33).

[ML13] J. Machan, C. Laugier. „Intelligent vehicles as an integral part of intelligent transport systems“. In: *ERCIM News, ISSN* (2013), S. 0926–4981 (zitiert auf S. 21).

[MM18] A. A. Munshi, Y. A.-R. I. Mohamed. „Data Lake Lambda Architecture for Smart Grids Big Data Analytics“. In: *IEEE Access* 6 (2018), S. 40463–40471. DOI: 10.1109/ACCESS.2018.2858256 (zitiert auf S. 26).

[MPLN17] W. Maddern, G. Pascoe, C. Linegar, P. Newman. „1 Year, 1000km: The Oxford RobotCar Dataset“. In: *The International Journal of Robotics Research (IJRR)* 36.1 (2017), S. 3–15. DOI: 10.1177/0278364916679498. eprint: http://ijr.sagepub.com/content/early/2016/11/28/0278364916679498.full.pdf+html. URL: http://dx.doi.org/10.1177/0278364916679498 (zitiert auf S. 50).

[MRS+15] A. I. Maarala, M. Rautiainen, M. Salmi, S. Pirttikangas, J. Riekki. „Low latency analytics for streaming traffic data with Apache Spark“. In: *2015 IEEE International Conference on Big Data (Big Data)*. 2015, S. 2855–2858. DOI: 10.1109/BigData.2015.7364101 (zitiert auf S. 25, 27).

[NFM16] T. Nawrath, D. Fischer, B. Markscheffel. „Privacy-sensitive data in connected cars“. In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)*. 2016, S. 392–393. DOI: 10.1109/ICITST.2016.7856736 (zitiert auf S. 23, 31).

[NJ15] L. Nkenyereye, J.-W. Jang. „A Study of Big Data Solution Using Hadoop to Process Connected Vehicle's Diagnostics Data“. In: Bd. 339. Jan. 2015, S. 697–704. ISBN: 978-3-662-46577-6. DOI: 10.1007/978-3-662-46578-3_82 (zitiert auf S. 25).

[PKÅ+20] P. Pelliccione, E. Knauss, S. M. Ågren, R. Heldal, C. Bergenhem, A. Vinel, O. Brunnegård. „Beyond connected cars: A systems of systems perspective“. In: *Science of Computer Programming* 191 (2020), S. 102414. ISSN: 0167-6423. DOI: https://doi.org/10.1016/j.scico.2020.102414. URL: https://www.sciencedirect.com/science/article/pii/S0167642320300253 (zitiert auf S. 13, 21, 22, 27–31, 34).

[PWD17] P. Patel, G. Wood, A. Diaz. „Data lake governance best practices“. In: *The DZone Guide to Big Data-Data Science and Advanced Analytics* 4 (2017), S. 6–7 (zitiert auf S. 18).

[RZ19a] F. Ravat, Y. Zhao. „Data Lakes: Trends and Perspectives“. In: *Database and Expert Systems Applications*. Hrsg. von S. Hartmann, J. Küng, S. Chakravarthy, G. Anderst-Kotsis, A. M. Tjoa, I. Khalil. Cham: Springer International Publishing, 2019, S. 304–313. ISBN: 978-3-030-27615-7 (zitiert auf S. 15, 16, 18).

[RZ19b] F. Ravat, Y. Zhao. „Metadata Management for Data Lakes“. In: *New Trends in Databases and Information Systems*. Hrsg. von T. Welzer, J. Eder, V. Podgorelec, R. Wrembel, M. Ivanović, J. Gamper, M. Morzy, T. Tzouramanis, J. Darmont, A. Kamišalić Latifić. Cham: Springer International Publishing, 2019, S. 37–44. ISBN: 978-3-030-30278-8 (zitiert auf S. 19, 20).

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu. „Edge Computing: Vision and Challenges“. In: *IEEE Internet of Things Journal* 3.5 (2016), S. 637–646. DOI: 10.1109/JIOT.2016.2579198 (zitiert auf S. 35).

[SGW+20] C. Stach, C. Giebler, M. Wagner, C. Weber, B. Mitschang. „AMNESIA: A Technical Solution towards GDPR-compliant Machine Learning.“ In: *ICISSP*. 2020, S. 21–32 (zitiert auf S. 31, 38).

[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 (2016), S. 22–28. DOI: 10.1109/MCOM.2016.7497762 (zitiert auf S. 21).

[Sta19] C. Stach. „Konzepte zum Schutz privater Muster in Zeitreihendaten“. In: *INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik – Informatik für Gesellschaft*. Hrsg. von K. David, K. Geihs, M. Lange, G. Stumme. Bonn: Gesellschaft für Informatik e.V., 2019, S. 353–366. DOI: 10.18420/inf2019_54 (zitiert auf S. 31).

[TD19] V. Theodorou, N. Diamantopoulos. „GLT: Edge Gateway ELT for Data-Driven Intelligence Placement“. In: *2019 IEEE/ACM Joint 4th International Workshop on Rapid Continuous Software Engineering and 1st International Workshop on Data-Driven Decisions, Experimentation and Evolution (RCoSE/DDrEE)*. 2019, S. 24–27. DOI: 10.1109/RCoSE/DDrEE.2019.00013 (zitiert auf S. 26, 35).

[THQ19] V. Theodorou, R. Hai, C. Quix. „A Metadata Framework for Data Lagoons“. In: *New Trends in Databases and Information Systems*. Hrsg. von T. Welzer, J. Eder, V. Podgorelec, R. Wrembel, M. Ivanović, J. Gamper, M. Morzy, T. Tzouramanis, J. Darmont, A. Kamišalić Latifić. Cham: Springer International Publishing, 2019, S. 452–462. ISBN: 978-3-030-30278-8 (zitiert auf S. 26).

[WM15] J. Warren, N. Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon und Schuster, 2015 (zitiert auf S. 17, 34).

[Wol16] D. Wollschläger. „Preconditions, requirements & prospects of the connected car“. In: *Auto Tech Review* 5.1 (2016), S. 30–35 (zitiert auf S. 13, 22, 27).

[XRZ+13] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, I. Stoica. „Shark: SQL and Rich Analytics at Scale“. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’13. New York, New York, USA: Association for Computing Machinery, 2013, S. 13–24. ISBN: 9781450320375. DOI: 10.1145/2463676.2465288. URL: https://doi.org/10.1145/2463676.2465288 (zitiert auf S. 16).

Alle URLs wurden zuletzt am 28. 11. 2021 geprüft.

A. Schemata Registry

A.1. JSON-Schema des Registry-Endpunktes der Data Lagoon

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/registry.schema.json",
  "title": "RegistryDataLagoon",
  "description": "Message for registration of a source at autoDataLagoon",
  "type": "object",
  "properties": {
    "dataType": {
      "description": "identifier for type of data a source wants to send",
      "type": "string"
    },
    "sourceName": {
      "description": "name of the source",
      "type": "string"
    },
    "metadata": {
      "description": "metadata that is send additional to source data",
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "saveToDataLake": {
      "description": "flag that indicates if data of source should also be saved to data lake"
,
      "type": "boolean"
    },
    "dataLabelField": {
      "description": "name of field that contains the label for a data set, e.g. ID of car",
      "type": "string"
    },
    "tables": {
      "description": "tables that should be created in case of relational data",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "tableName": {
            "description": "name of the table to be created",
            "type": "string"
          },
```

```
          "columns": {
            "description": "columns of the table",
            "type": "array",
            "items": {
              "type": "string"
            },
            "minItems": 1
          }
        },
        "required": [
          "tableName",
          "columns"
        ]
      },
      "minItems": 1
    },
    "timeSeries": {
      "description": "further information needed in case of time series data",
      "type": "object",
      "properties": {
        "valueField": {
          "description": "name of field that contains the value of the time series data",
          "type": "string"
        },
        "timestampField": {
          "description": "name of field that contains the timestamp of the time series data",
          "type": "string"
        },
        "measurementName": {
          "description": "name that should be used to identify the time series data",
          "type": "string"
        }
      }
    },
    "otherStructure": {
      "description": "further information needed in case of unstructured or semi-structured
data",
      "type": "object",
      "properties": {
        "collectionName": {
          "description": "name that should be used to identify the collection",
          "type": "string"
        },
        "idField": {
          "description": "name of the field that should be used as id for an entry in
collection",
          "type": "string"
        }
      }
    }
  },
  "required": [
    "dataType",
```

```
    "sourceName",
    "saveToDataLake",
    "dataLabelField"
  ]
}
```

A.2. JSON-Schema des Registry-Endpunktes der Data Lake

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/registry.schema.json",
  "title": "RegistryDataLagoon",
  "description": "Message for registration of a source at autoDataLake",
  "type": "object",
  "properties": {
    "dataType": {
      "description": "identifier for type of data a source wants to send",
      "type": "string"
    },
    "sourceName": {
      "description": "name of the source",
      "type": "string"
    },
    "metadata": {
      "description": "metadata that is send additional to source data",
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "dataLabelField": {
      "description": "name of field that contains the label for a data set, e.g. ID of car",
      "type": "string"
    },
    "tables": {
      "description": "tables that should be created in case of relational data",
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "tableName": {
            "description": "name of the table to be created",
            "type": "string"
          },
          "columns": {
            "description": "columns of the table",
            "type": "array",
            "items": {
              "type": "string"
            },
            "minItems": 1
          }
```

```
        },
        "required": [
          "tableName",
          "columns"
        ]
      },
      "minItems": 1
    },
    "timeSeries": {
      "description": "further information needed in case of time series data",
      "type": "object",
      "properties": {
        "valueField": {
          "description": "name of field that contains the value of the time series data",
          "type": "string"
        },
        "timestampField": {
          "description": "name of field that contains the timestamp of the time series data",
          "type": "string"
        },
        "measurementName": {
          "description": "name that should be used to identify the time series data",
          "type": "string"
        }
      }
    },
    "otherStructure": {
      "description": "further information needed in case of unstructured or semi-structured
data",
      "type": "object",
      "properties": {
        "collectionName": {
          "description": "name that should be used to identify the collection",
          "type": "string"
        },
        "idField": {
          "description": "name of the field that should be used as id for an entry in
collection",
          "type": "string"
        }
      }
    }
  },
  "required": [
    "dataType",
    "sourceName",
    "saveToDataLake",
    "dataLabelField"
  ]
}
```

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