Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Modellierung von Daten-Pipelines in Lakehouses

Kerstin Fill

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Jan Schneider, M.Sc., Dr. rer. nat. Pascal Hirmer
Beginn am:	17. Oktober 2022
Beendet am:	17. April 2023

Kurzfassung

Das Aufkommen von Lakehouses hat neue Möglichkeiten für die Speicherung, Verarbeitung und Analyse großer Datenmengen eröffnet. Ein Lakehouse ist eine Datenplattform, die auf einem kostengünstigen Objektspeicher basiert und ACID-Eigenschaften bei Datenzugriffen unterstützt. Dieser Aufbau befähigt Lakehouses, Anwendungsfälle zu bedienen, die typischerweise von Data Lakes, Data Warehouses oder einer Kombination dieser Plattformen abgedeckt werden. Die sogenannte Delta-Architektur schlägt ein fortschrittliches Architekturmuster für die Verarbeitung von Daten innerhalb eines Lakehouses vor. Ein zentrales Konzept dabei bilden Daten-Pipelines, die sich über mehrere Tabellen eines Lakehouses hinweg aufspannen und eine Reihe von Verarbeitungsschritten ausführen. Ein einzelner Verarbeitungsschritt konsumiert und verarbeitet dabei Daten einer Tabelle als Batch oder Stream, um sie in einer anderen Tabelle erneut zu persistieren. Verglichen mit den bisher gängigen Architekturen Lambda und Kappa für Data Lakes ist die Delta-Architektur weniger komplex, da die Koordination und Synchronisation mehrerer Systeme entfällt. Jedoch ist der Entwurf, die Ausführung und Wartung von Daten-Pipelines auch bei der Delta-Architektur weiterhin erforderlich. Im Rahmen dieser Arbeit wird ein plattformunabhängiges Metamodell für Daten-Pipelines in Lakehouses entworfen. Es basiert auf Anforderungen, die aus typischen Anwendungsszenarien für Lakehouses abgeleitet werden. Da Lakehouses neuartige Datenplattformen darstellen, nimmt das Metamodell Bezug auf die in der Forschung bereits diskutierten Zonenmodelle von Data Lakes. Darauf aufbauend wird ein Konzept zur Entwicklung eines Modellierungswerkzeugs vorgestellt, das an die Model Driven Architecture (MDA) angelehnt ist. Das Modellierungswerkzeug soll den Nutzern die Möglichkeit bieten, Daten-Pipelines mit einem deklarativen Ansatz zu entwerfen und diese bei Bedarf um plattformspezifische Informationen zu erweitern. Darüber hinaus soll es die Ausführung solcher Daten-Pipelines innerhalb eines Lakehouses mithilfe eines Prozessierungsframeworks unterstützen. Auf der Grundlage der vorgeschlagenen Konzepte wird das Modellierungswerkzeug prototypisch implementiert. Abschließend legt eine qualitative Evaluation dar, inwieweit das Metamodell die zuvor aufgestellten Anforderungen erfüllt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Abkürzungsverzeichnis

API Application Programming Interface. 69

ASGI Asynchronous Server Gateway Interface. 96

CIM Computation Independent Model. 62

CSS Cascading Style Sheets. 68

DAG Directed Acyclic Graph. 31

DLT Delta Live Tables. 32

DSGVO Datenschutz-Grundverordnung. 40

ETL Extraction-Transformation-Load. 17

HTML Hypertext Markup Language. 68

IoT Internet of Things. 56

IVML Integrated Variability Modeling Language. 30

MDA Model Driven Architecture. 3

MOF Meta Object Facility. 60

OLAP Online Analytical Processing. 17

OMG Object Managment Group. 60

PCF Process Control Framework. 31

PIM Platform Independent Model. 62

PSM Platform Specific Model. 62

SQL Structured Query Language. 72

UI User Interface. 70

UML Unified Modeling Language. 29

1. Einleitung

Ein Blick auf die letzten Jahrzehnte zeigt, dass Daten in vielen Bereichen immer wichtiger geworden sind. Dieser Trend ist ungebrochen. Im Zeitalter der Digitalisierung unterliegen wirtschaftliche und geschäftliche Aktivitäten, aber auch die öffentliche Verwaltung und die wissenschaftliche Forschung einem Wandel [LSB+17]. Insbesondere Unternehmen sind angespornt, sich dieser digitalen Transformation zu unterziehen, um ihre Wettbewerbsfähigkeit zu sichern und zu erweitern. Eine schnelle, datengetriebene Entscheidungsfindung auf zahlreichen Ebenen kann als Mittel zur Prozessoptimierung und Kostensenkung genutzt werden. Dazu sammeln Unternehmen entlang ihrer Wertschöpfungskette eine Vielzahl von Daten. Ihr großer Umfang, ihre Heterogenität und oftmals hohe Erfassungsgeschwindigkeit erlegen den Unternehmen dabei eine Reihe von Herausforderungen auf. Diese ergeben sich vor allem in der Verwaltung und Speicherung der Daten [LSB+17]. Bevor es möglich ist, ihren potentiellen Wert zu erschließen, müssen diese Daten anforderungsgerecht abgelegt und vorverarbeitet werden [RSGJ13].

Um diesen Herausforderungen zu begegnen, wurden verschiedene Arten von Datenplattformen entwickelt und in der Praxis eingesetzt. Ein vergleichsweise neues Konzept auf diesem Gebiet ist das sogenannte *Lakehouse*. Dieses baut auf einem kostengünstigen, direkt zugänglichen Objektspeicher auf und bietet darüber hinaus analytische Verwaltungs- und Leistungsfunktionalitäten wie Datenzugriffe mit ACID-Eigenschaften, Caching und Abfrageoptimierung [AGXZ21]. Wie bereits durch den Namen angedeutet, strebt die Lakehouse-Vision danach, die entscheidenden Vorteile der bisher genutzten Data Warehouses und Data Lakes zu vereinen [AGXZ21]. Ähnlich wie bei einem Data Lake werden die Daten in offenen Dateiformaten gespeichert, was den Zugriff über viele verschiedene Systeme ermöglicht. Durch die Verwaltung technischer Metadaten können Data-Warehouse-Funktionen für das zugrundeliegende Speichersystem umgesetzt werden. Eine solche integrierte analytische Datenplattform ist die Grundlage für die Umsetzung eines neuartigen Architekturmusters, das unter dem Namen *Delta-Architektur* bekannt geworden ist. Es eröffnet Unternehmen die Aussicht, ihre oftmals komplexen Datenarchitekturen zu vereinfachen. Eine wesentliche Neuerung im Vergleich zu den bisher gängigen zusammengesetzten Architekturen Lambda und Kappa für Data Lakes ist in der Vereinheitlichung von Batch- und Stream-Speicherung und -Verarbeitung zu sehen. Im Rahmen der Delta-Architektur können Rohdaten in ein Lakehouse eingespeist und dort logisch in Tabellen organisiert werden, bevor sie durch nachgelagerte Verarbeitungsschritte bereinigt und aufbereitet werden. Jeder Verarbeitungsschritt greift dabei auf eine Tabelle zu, führt seine spezifischen Operationen durch und persistiert seine Ergebnisse in einer neuen Tabelle [PNM22]. Ob die Einspeisung der Daten als Batch oder Stream ausgeführt wird und die Verarbeitungsschritte im Batch- oder Streaming-Modus arbeiten, ist hinsichtlich Korrektheit, Vollständigkeit und Konsistenz der Daten nicht relevant. Das Lesen und Schreiben von Daten einer Tabelle erfolgt über Datenzugriffe mit ACID-Eigenschaften, was die Integrität der abgelegten Daten sicherstellt. Dank diesen Eigenschaften kann die Delta-Architektur den vielfältigen Anforderungen

verschiedener Nutzer gerecht werden. So können beispielsweise Streaming-Analysen ebenso unterstützt werden wie die Auswertung historischer Zeitreihen, um neue Modelle zu trainieren oder standardisierte Berichte zu erstellen [KM21].

Daten-Pipelines sind ein wichtiger Bestandteil der Delta-Architektur und von zentraler Bedeutung für die effiziente Datenverarbeitung innerhalb eines Lakehouses. In einer Daten-Pipeline werden zusammengehörige Verarbeitungsschritte gruppiert. Ihre Ersteller können sich im Idealfall auf die funktionale Komplexität der Verarbeitungsschritte konzentrieren, indem sie bei technischen Hürden durch ein Werkzeug unterstützt werden. Die Modellierung einer Pipeline auf abstrakter Ebene, die dem Verständnis des Datenflusses entspricht, reicht aus, um die Abfolge der Verarbeitungsschritte mit einer geeigneten Implementierung zu verknüpfen [EQS17]. Allerdings wird derzeit keine umfassende, plattformunabhängige Werkzeugunterstützung für den Aufbau, die Ausführung und die Wartung von Daten-Pipelines in Lakehouses angeboten. Das Ziel dieser Masterarbeit ist es daher, ein Metamodell zu entwerfen, das die Modellierung von Daten-Pipelines in Lakehouses gemäß der Delta-Architektur ermöglicht. Das Metamodell wird aus Anforderungen typischer Anwendungsszenarien für Daten-Pipelines abgeleitet und dient als Grundlage für die prototypische Implementierung eines Modellierungswerkzeugs. Abschließend wird mithilfe einer qualitativen Evaluation ermittelt, inwieweit das Metamodell den zuvor formulierten Anforderungen entspricht.

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Grundlagen: Zu Beginn werden die wichtigsten Grundlagen und zentralen Begriffe eingeführt, auf die nachfolgende Kapitel aufbauen. Insbesondere werden Data Lakes sowie deren Zonenmodelle beleuchtet und Lakehouses mit ihren Vor- und Nachteilen betrachtet.

Kapitel 3 – Verwandte Arbeiten: Um den Beitrag dieser Arbeit von anderen Studien und auf dem Markt verfügbaren Werkzeugen abzugrenzen, stellt dieses Kapitel heraus, welche Vorschläge zur Modellierung von Daten-Pipelines bisher gemacht wurden und wie verfügbare Softwarelösungen einzuordnen sind.

Kapitel 4 – Anforderungen: In diesem Kapitel werden die Anforderungen identifiziert und diskutiert, die sich an ein Metamodell für Daten-Pipelines in Lakehouses stellen.

Kapitel 5 – Konzeption: Auf Basis der im vorherigen Kapitel gesammelten Anforderungen wird ein Metamodell entworfen. Dieses Kapitel stellt den zugrundeliegenden Modellierungsansatz und den Aufbau des Metamodells vor. Zudem werden Konzepte zur Entwicklung eines Modellierungswerkzeugs präsentiert, die an die Model-Driven-Architecture angelehnt sind.

Kapitel 6 – Prototypische Implementierung eines Modellierungswerkzeugs: Neben dem Metamodell geht aus dieser Arbeit ein Prototyp für die Modellierung von Daten-Pipelines in Lakehouses hervor. Die bei der Implementierung eingesetzten Technologien, sowie die Architektur und Funktionsweise des Prototyps werden in diesem Kapitel behandelt.

Kapitel 7 – Evaluation: Dieses Kapitel befasst sich mit der Evaluation des Metamodells basierend auf den in Kapitel 4 gesammelten Anforderungen. Diese sollten bei gelungener modellbasierter Entwicklung auch vom Prototyp erfüllt sein. Ob und inwieweit dies der Fall ist, wird für die einzelnen Anforderungen herausgearbeitet.

Kapitel 8 – Zusammenfassung und Ausblick: Abschließend wird der Beitrag dieser Arbeit zusammengefasst und auf mögliche Anknüpfungspunkte für zukünftige Arbeiten eingegangen.

2. Grundlagen

Dieses Kapitel widmet sich den Grundlagen, die für das Verständnis und die Nachvollziehbarkeit dieser Masterarbeit von Bedeutung sind. Lakehouses können als analytische Datenplattformen betrachtet werden, die wesentliche Vorteile von Data Warehouses und Data Lakes vereinen. Das vorliegende Kapitel orientiert sich an dieser Perspektive und gibt in Abschnitt 2.1 eine Übersicht über Eigenschaften von Data Warehouses, während Abschnitt 2.2 verschiedene Aspekte von Data Lakes fokussiert. Im Unterabschnitt 2.2.1 werden ein Metamodell und ein Zonenreferenzmodell für Zonen in Data Lakes vorgestellt, da diese Modelle eng mit der Konzeption dieser Masterarbeit verbunden sind. Ebenfalls von Relevanz sind die Systemarchitekturen Lambda und Kappa für Data Lakes, die in Unterabschnitt 2.2.2 betrachtet werden. Der letzte Abschnitt 2.3 dieses Kapitels befasst sich mit Lakehouses. Zu Beginn erfolgt eine begriffliche Einordnung, um die Art von Datenplattformen zu charakterisieren, die in dieser Arbeit als Lakehouses betrachtet werden. Daraufhin werden in Unterabschnitt 2.3.2 die grundlegenden Eigenschaften von Lakehouses herausgestellt. Schließlich wird als weniger komplexe Alternative zu den Lambda- und Kappa-Architekturen in Unterabschnitt 2.3.3 die Delta-Architektur erklärt, die mithilfe eines Lakehouses umgesetzt werden kann.

2.1. Data Warehouses

Data Warehouses bieten die Möglichkeit, große Datenmengen zu verwalten und für Analysezwecke bereitzustellen. Ursprünglich wurden sie entwickelt, um Führungskräfte bei der Gewinnung von analytischen Erkenntnissen und der strategischen Entscheidungsfindung in einer wettbewerbsorientierten Wirtschaft zu unterstützen [AGXZ21]. Auch heute noch dienen Data Warehouses als Speicher für Unternehmensdaten und als primäre Datenquelle für die Erstellung, Analyse und Präsentation von Berichten innerhalb von Unternehmen [NM22]. Im Vergleich zu relationalen Datenbanken stellen Data Warehouses eine Weiterentwicklung dar, die speziell auf die Anforderungen von Geschäftsumgebungen mit großen Datenmengen ausgelegt ist.

Die Einspeisung von Daten in ein Data Warehouses erfolgt im Rahmen eines Extraction-Transformation-Load (ETL)-Prozesses. Hierbei werden Daten aus einer Vielzahl von heterogenen externen Datenquellen extrahiert und transformiert, bevor sie in einen üblicherweise relationalen Datenspeicher geschrieben werden [KW18]. Nach der Überführung der Daten in ein vordefiniertes Schema bietet ein Data Warehouse eine „themenorientierte, integrierte, chronologisierte und persistente Sammlung von Daten“ [ISL21]. Auf dieser Grundlage können Data Warehouses eine Reihe vorteilhafter Eigenschaften realisieren, wie zum Beispiel kurze Antwortzeiten bei konsistenter Performanz, geringe Redundanz und die Unterstützung von ACID-Transaktionen (Atomicity, Consistency, Isolation, Durability) [RZ19]. Konsumenten greifen dann auf zugeschnittene Extrakte der Daten zu, welche in Data Marts abgelegt werden und anschließend für Online Analytical Processing (OLAP) genutzt werden können [NM22]. Moderne Data Warehouses werden über

Cloud-Bereitstellungsmodelle realisiert und übernehmen somit deren Vorteile in Hinblick auf Skalierbarkeit und Kosten. Um den Anforderungen an eine moderne Datenplattform gerecht zu werden, unterstützen diese außerdem erweiterte Verwaltungsfunktionen. Diese bieten Unternehmen die Möglichkeit, Datenschutzrichtlinien umzusetzen, über *Time Travel* auf historische Daten zuzugreifen und durch *Zero-Copy Cloning* logische Kopien der Daten zu erstellen, ohne sie physisch duplizieren zu müssen [SGL+23]. Des Weiteren sind moderne Data Warehouses in der Lage, *Schema Evolution* zu unterstützen, was bedeutet, dass sie mit Änderungen des zugrundeliegenden Schemas umgehen können [ADS+20; HQJ21].

In einem Data Warehouse wird das Datenmodell optimal auf bestimmte Anwendungsfälle zugeschnitten, so dass auch Nutzer mit geringen IT-Kenntnissen aufbereitete Daten direkt konsumieren können [SBE+21]. Allerdings geht dieser Vorteil mit einem Informationsverlust einher, da die extrahierten Rohdaten durch die Transformation verändert und aggregiert werden. Zusätzlich verzögert sich die Bereitstellung der Daten durch diesen Verarbeitungsschritt, was Echtzeitanalysen erschwert [SBE+21]. Ein weiteres Problem ergibt sich aus der mangelnden Fähigkeit von Data Warehouses, mit einer hohen Heterogenität der Rohdaten umzugehen. Unstrukturierte Daten, wie Video-, Audio- und Textdokumente, können nicht effizient gespeichert und abgefragt werden, was für viele Anwendungsfälle inzwischen nicht mehr ausreicht [AGXZ21].

2.2. Data Lakes

Der zunehmende Trend zur Digitalisierung hat das Spektrum der Datenquellen für analytische Aufgaben erheblich erweitert. Traditionelle Data Warehouses waren oft nicht in der Lage, mit wachsenden Datenmengen und einer immer größer werdenden Datenvielfalt in Unternehmen umzugehen, so dass ein Bedarf an flexibleren und besser skalierbaren Datenplattformen entstand. Data Lakes haben sich als eine Lösung zur Bewältigung dieser Herausforderungen etabliert [GGH+19]. Nach der Definition von Fang [Fan15] entspricht ein Data Lake einem „riesigen Datenspeicher, der auf kostengünstigen Technologien aufbaut und die Erfassung, Verfeinerung, Archivierung und Erkundung von Rohdaten innerhalb eines Unternehmens verbessert.“ Ravat und Zhao [RZ19] ergänzen in ihrer Definition, dass ein Data Lake strukturierte, semi-strukturierte und unstrukturierte Rohdaten aufnehmen kann, die aus Datenquellen innerhalb oder außerhalb eines Unternehmens stammen. Diese Rohdaten werden als Batch oder Stream in den Data Lake eingespeist und in ihrem nativen Format abgelegt. Nach ihrer Bereitstellung an einem zentralen Speicherort können die Rohdaten von verschiedenen Konsumenten eines Unternehmens genutzt werden [KW18].

Die Datenverarbeitung in einem Data Lake unterscheidet sich von den ETL-Prozessen, wie sie bei Data Warehouses zum Einsatz kommen. Ein ETL-Prozess transformiert die extrahierten Rohdaten, bevor sie in einem vordefinierten Schema abgelegt werden. Dieser Ansatz wird als *Schema-on-Write* bezeichnet. Hingegen verfolgt ein Data Lake den sogenannten *Schema-on-Read*-Ansatz [Mat17]. Bei diesem wird die Erstellung eines Schemas aufgeschoben, bis die Daten aus dem Data Lake für eine bestimmte Nutzung abgefragt werden. Durch die Ablage der Daten in ihrem nativen Format entfallen Vorabkosten für komplexe Vorverarbeitungen und Transformationen [KW18]. Gleichzeitig wird die Flexibilität der Daten erhalten, da ein Datensatz in verschiedene Schemata überführt werden kann [Mat17]. Auf diese Weise unterstützt ein Data Lake die Verarbeitung von Daten nach den vielfältigen Anforderungen verschiedener Nutzergruppen [RZ19].

Trotz der genannten Vorteile des Schema-on-Read-Ansatzes hat er sich insbesondere in der praktischen Anwendung als ineffizient erwiesen. Wenn die Rohdaten für ähnliche Verwendungszwecke wiederverwendet werden, müssen auf ihnen wiederholt dieselben Verarbeitungsschritte durchgeführt werden [GGH+20]. Um diesem Problem zu entgegnen, bietet es sich an, denselben Datensatz mehrfach in unterschiedlichen Verarbeitungsgraden im Data Lake abzulegen [EGG+20]. Zur Verwaltung und logischen Organisation dieser vorverarbeiteten Daten schlägt die Literatur verschiedene Zonenmodelle vor. Diese unterscheiden sich in der Anzahl und Funktion der einzelnen Zonen. Grundsätzlich besteht der Zweck eines Zonenmodells darin, Daten innerhalb eines Data Lakes zu organisieren, damit diese unmittelbar für verschiedene Datenverarbeitungs- und Analyseaufgaben herangezogen werden können. Hierfür definiert ein Zonenmodell, in welchen Verarbeitungsgraden Daten im Data Lake vorliegen und wie diese abhängig von ihrer Zonenzuordnung verwaltet werden [GGH+20]. Außerdem kann ein Zonenmodell einen wichtigen Baustein der Data-Governance-Strategie des Unternehmens darstellen, indem es klare Vorschriften bezüglich Zugriffsrechten und Verantwortlichkeiten für die Daten eines Data Lakes schafft. Dies fördert die Einhaltung von Datenschutzrichtlinien und eine bessere Kontrolle über den Zugriff auf sensible Daten [GGH+19]. Im nachfolgenden Unterabschnitt 2.2.1 werden ein Metamodell sowie ein Zonenreferenzmodell für Data-Lake-Zonen vorgestellt.

Wenn ein Data Lake ohne Data-Governance-Maßnahmen verwendet wird, besteht die Gefahr, dass er sich zu einem sogenannten *Data Swamp* wandelt. Dieser Begriff charakterisiert eine Situation, in der ein Data Lake mit inkorrekten, duplizierten oder irrelevanten Daten gefüllt ist [KW18]. Er kann aus einer mangelnden Kontrolle über Daten, Verarbeitungspipelines, Anwendungen und Nutzern resultieren [Mat17]. Um die Entstehung eines Data Swamps zu verhindern, ist es zwingend erforderlich, dass explizite und aussagekräftige Metadaten in einem Data Lake vorgehalten werden. Die Metadaten stellen zusätzliche Informationen über die gespeicherten Daten bereit und können beispielsweise aus Datenquellen extrahiert werden [NZM+19]. Für die korrekte Bereinigung und Transformation der Daten in einem Data Lake sind diese Metadaten zu berücksichtigen. Zudem können sie Nutzern bei der Einschätzung helfen, welche Daten überhaupt verfügbar sind, was diese bedeuten und wie sie verwendet werden können [NZM+19].

2.2.1. Metamodell und Zonenreferenzmodell für Zonen in Data Lakes

In der Literatur sind zahlreiche Varianten von Zonenmodellen zu finden. Giebler et al. [GGH+20] evaluieren diese anhand von Anforderungen, die sie aus repräsentativen Anwendungsfällen verschiedener Geschäftsbereiche für Data Lakes extrahiert haben. Diese Anwendungsfälle sollen alle Arten von Daten-Analysen abdecken, die sich in den Bereichen Reporting/OLAP und Advanced Analytics ergeben können. Darauf aufbauend schlagen die Autoren ein Metamodell und ein Zonenreferenzmodell für Data-Lake-Zonen vor. Eine Beschreibung dieser Modelle erfolgt in den folgenden Unterabschnitten.

Metamodell für Zonen

Das Metamodell für Data-Lake-Zonen extrahiert die gemeinsamen Eigenschaften einer Zone aus einer Reihe von untersuchten Zonenmodellen und führt diese in einem Entity-Relationship-Diagramm (ER-Diagramm) zusammen. Auf diese Weise ermöglicht es eine standardisierte und

vergleichbare Beschreibung von Data-Lake-Zonen. An zentraler Stelle des Metamodells befindet sich der Entitätstyp ZONE, dem eine Reihe von Attributen und Beziehungen zugeschrieben werden. Neben einem eindeutigen Namen besitzt eine Entität vom Typ ZONE die folgenden vier Attribute:

Data Characteristics: Den Daten einer Zone haften bestimmte Eigenschaften an. Diese lassen sich in die vier Aspekte *Granularität*, *Schema*, *Syntax* und *Semantik* aufgliedern.

Properties: Die Eigenschaften einer Zone charakterisieren, welche Handhabung eine Zone für die ihr zugeordneten Daten vorsieht. Beispielsweise kann eine Zone vorgeben, dass Daten innerhalb dieser Zone über einen längeren Zeitraum zu persistieren sind oder diese bestimmte Data-Governance-Richtlinien des Unternehmens zu berücksichtigen haben.

User Groups: Mehrere Nutzer einer bestimmten Benutzergruppe können mit einer Zone interagieren. Dabei beschränkt sich das Attribut nicht nur auf menschliche Nutzer, wie beispielsweise Data Scientists, sondern auch Systeme und Prozesse werden als Nutzer aufgefasst.

Modeling Approach: Der Modellierungsansatz einer Zone drückt aus, wie ein bestimmtes Schema umgesetzt werden kann. Denkbar wäre zum Beispiel, dass Formate des Quellsystems kopiert oder die Daten in Dateien abgelegt werden.

Zusätzlich zu ihren Attributen ist eine Zone durch Beziehungen gekennzeichnet, die sie zu anderen Zonen oder Systemen außerhalb des Data Lakes eingeht. Aus diesen geht hervor, dass eine Datenübertragung sowohl zwischen Zonen als auch von externen Datenquellen und zu externen Datensenken zulässig ist.

Zonenreferenzmodell

Neben dem Metamodell für Data-Lake-Zonen schlagen Giebler et al. [GGH+20] ein Zonenreferenzmodell für Data Lakes vor. Dieses entsteht durch Instanziierung des Metamodells und soll Orientierung bei der Umsetzung einer zonenbasierten Verwaltung eines Data Lakes bieten. Es entspricht einer von vielen möglichen Instanziierungen, wobei die konkrete Auswahl der Zonen auf Basis der Zusammenführung mehrerer Konzepte der Fachliteratur und gängigen Praxisanforderungen getroffen wurde. In Abbildung 2.1 sind die vom Zonenreferenzmodell vorgesehenen Zonen dargestellt. Die Abbildung zeigt neben den Namen der einzelnen Zonen und ihren Interaktionen untereinander, dass das Zonenreferenzmodell zwischen einem anwendungsunabhängigen und einem anwendungsabhängigen Teil unterscheidet. Diese Einteilung der Zonen kennzeichnet, dass mit der fortschreitenden Verarbeitung der Daten zum einen ein Informationsverlust einhergeht, aber zum anderen auch eine bessere Unterstützung bestimmter Anwendungsszenarien erreicht werden kann. Des Weiteren geht aus der Abbildung hervor, dass jede Zone einen geschützten Teil umfasst. Dessen Zweck ist die Speicherung besonders sensibler Daten, wie zum Beispiel personenbezogener Daten. Er unterscheidet sich vom Rest der Zone, da auf diesem strenge Zugangskontrollen und Überprüfungen durchgeführt werden. Im Folgenden werden die sechs Zonen vorgestellt, da ihre Bedeutung für den weiteren Verlauf der Arbeit relevant ist:

Landing-Zone: Daten, die in einen Data Lake aufgenommen werden, kommen in der LANDING-Zone an. Sie empfiehlt sich, wenn die Anforderungen der eingespeisten Daten von denen der RAW-Zone abweichen. Beispielsweise kann die LANDING-Zone bei einer zu hohen Einspeisungsrate von Streaming-Daten als Mediator zur *Raw*-Zone fungieren.

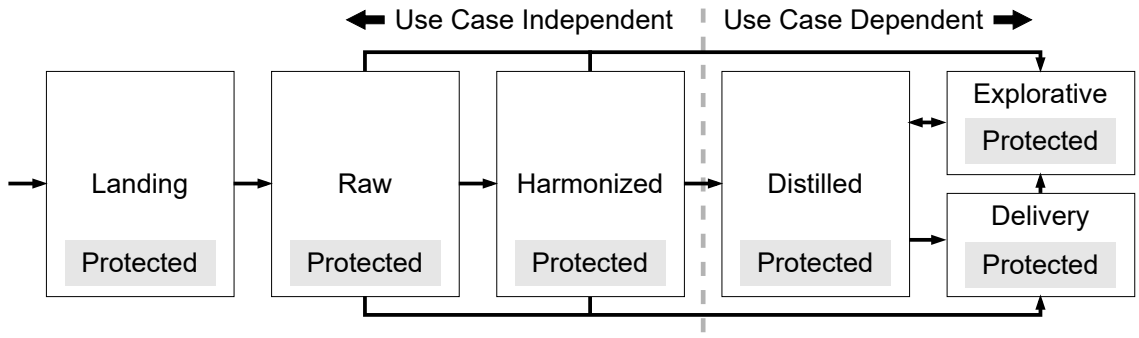

Abbildung 2.1.: Zonenreferenzmodell für Data Lakes [GGH+20].

Raw-Zone: Alle Daten der RAW-Zone zeichnen sich durch einen weitgehend unverarbeiteten Zustand aus. Dort sollten sie dauerhaft persistiert werden, jedoch muss ein Kompromiss zwischen dem verfügbaren Speicherplatz und der Vollständigkeit der Daten gefunden werden.

Harmonized-Zone: Daten der RAW-Zone werden kopiert und in bereinigter Form in der HARMONIZED-Zone abgelegt. In dieser Zone sollten auch die Stammdaten eines Unternehmens zusammengeführt werden und für alle Arten von Analysen zugänglich sein.

Distilled Zone: Um die Effizienz nachgelagerter Analysen zu erhöhen, werden Daten in der DISTILLED-Zone in aufbereiteter Form bereitgestellt. Die DISTILLED-Zone ist die erste Zone, die bereits auf bestimmte Gruppen von Anwendungsfällen abgestimmt ist.

Explorative Zone: Auf die EXPLORATIVE-Zone greifen Data Scientists zu, um die Daten für Data-Science-Anwendungen zu verwenden. Insbesondere zeichnet sich die EXPLORATIVE-Zone dadurch aus, dass sie im Vergleich zu den anderen Zonen weniger streng reglementiert ist, um eine möglichst flexible Nutzung zu gestatten.

Delivery-Zone: Ähnlich wie die EXPLORATIVE-Zone sind auch die Daten der DELIVERY-Zone auf bestimmte Anwendungsfälle ausgerichtet. Die Unterscheidung zur EXPLORATIVE-Zone lässt sich unter anderem an den Benutzergruppen festmachen, da auf die DELIVERY-Zone eine größere Menge an menschlichen und nicht-menschlichen Nutzern zugreifen darf. Außerdem können Daten von der DELIVERY-Zone aus an externe Datenquellen weitergegeben werden.

Die Autoren betonen, dass sich das Zonenreferenzmodell an die spezifischen Anforderungen eines Anwendungsszenarios anpassen lässt, indem beispielsweise ein oder mehrere Zonen weggelassen werden. Abgesehen von der RAW-Zone sind alle Zonen optional. Außerdem ist das Zonenreferenzmodell für die Verwaltung heterogener Batch- und Streaming-Daten geeignet.

2.2.2. Systemarchitekturen

Ein entscheidender Aspekt bei der Konzeption eines Data Lakes sind architektonische Entscheidungen in Bezug auf die Organisation von Batch- und Stream-Verarbeitung [GSSM18]. Diese Verarbeitungsparadigmen können anhand ihrer unterschiedlichen Arbeitsweise und ihres Anwendungsbereichs voneinander abgegrenzt werden. Batch-Verarbeitung wird häufig auf große Mengen historischer Daten angewendet, die dauerhaft in einem Speichersystem abgelegt sind. Dazu wird

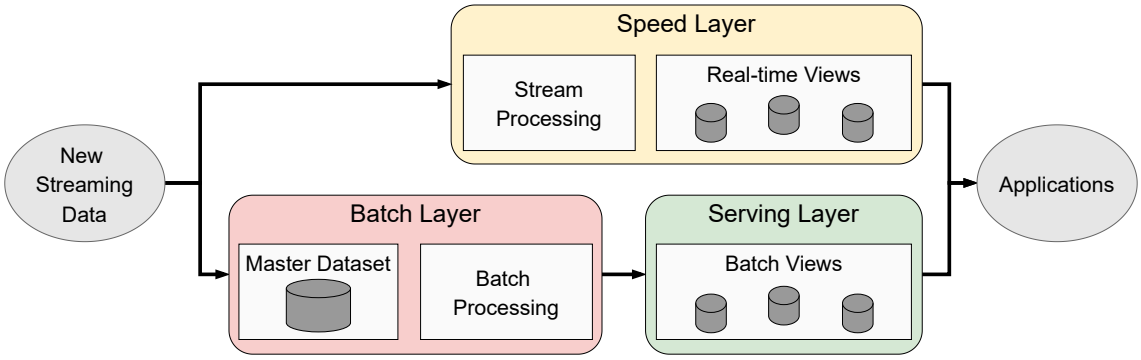

Abbildung 2.2.: Schematische Übersicht der Lambda-Architektur nach Warren und Marz [WM15].

die gesamte Datenmenge in kleinere Teilmengen unterteilt, die als Batches bezeichnet werden. Diese Batches können dann separat auf verschiedenen Knoten in einem verteilten System verarbeitet werden, um die Verarbeitungszeit zu verkürzen. Das Ergebnis liegt typischerweise erst nach Verarbeitung aller Batches vor [GSSM18]. Hingegen wird bei einer Stream-Verarbeitung ein unendlicher Datenstrom in kleine Datenpakete aufgeteilt und direkt im Arbeitsspeicher verarbeitet. Die Ergebnisse sind sofort verfügbar. Außerdem wirkt sich eine Änderung der Verarbeitungslogik unmittelbar auf die Verarbeitung neu ankommender Datenpakete aus [GSSM18]. Im Vergleich zur Stream-Verarbeitung erreicht eine Batch-Verarbeitung eine geringere Fehleranfälligkeit. Wenn es jedoch um eine möglichst geringe Verarbeitungslatenz geht, ist die Stream-Verarbeitung überlegen [CY14]. Viele Anwendungsbereiche erfordern sowohl eine geringe Fehleranfälligkeit als auch eine geringe Verarbeitungslatenz. Um beides zu erreichen, können mehrere Systeme in einer hybriden Architektur kombiniert werden [CY14]. Zwei Beispiele für solche Architekturen heißen *Lambda* und *Kappa*, deren Funktionsweise im Folgenden beleuchtet wird:

Lambda-Architektur Die Lambda-Architektur, welche in Abbildung 2.2 dargestellt ist, besteht aus drei Schichten, die als *Batch-Layer*, *Speed-Layer* und *Serving-Layer* bezeichnet werden. Eingehende Daten können über zwei verschiedene Zweige verarbeitet werden, die sich in ihrem Verarbeitungsmodus unterscheiden. Der Batch-Layer verwaltet den unveränderlichen Master-Datensatz, zu dem neue Daten als Anhänge hinzugefügt werden. In regelmäßigen Abständen gehen die Daten in eine Batch-Verarbeitung ein, deren Ergebnisse als *Batch Views* im Serving-Layer zur Nutzung bereitgestellt werden. Wie sein Name impliziert, verarbeitet der Speed-Layer die Daten in Echtzeit, wobei seine Ergebnisse den Inhalt sogenannter *Real-time Views* stellen. Da die Batch-Verarbeitung lediglich Daten berücksichtigt, die bei ihrem Start verfügbar waren, und zudem viel Zeit in Anspruch nimmt, enthalten die Batch Views in der Regel veraltete Daten. Für eine vollständige und aktuelle Sicht, müssen die Ergebnisse beider Verarbeitungszweige abgefragt und kombiniert werden. Dieser Synchronisierungsschritt gestaltet sich zeitaufwendig und komplex, weil sich beispielsweise die Schemata von Batch und Real-time Views unterscheiden können [CY14; GSSM18; WM15].

Kappa-Architektur Die Kappa-Architektur ist in Abbildung 2.3 visualisiert und stellt eine Alternative zur Lambda-Architektur dar. Sie besitzt keinen Batch-Layer, sondern wendet Stream-Verarbeitung auf allen Daten an. Eingehende Daten werden im Master-Datensatz persistiert, bevor sie als Stream verarbeitet werden. Dies erfolgt typischerweise in der Reihenfolge, in der neue Echtzeitdaten eintreffen. Da die Kappa-Architektur mehrere Prozessierungsjobs

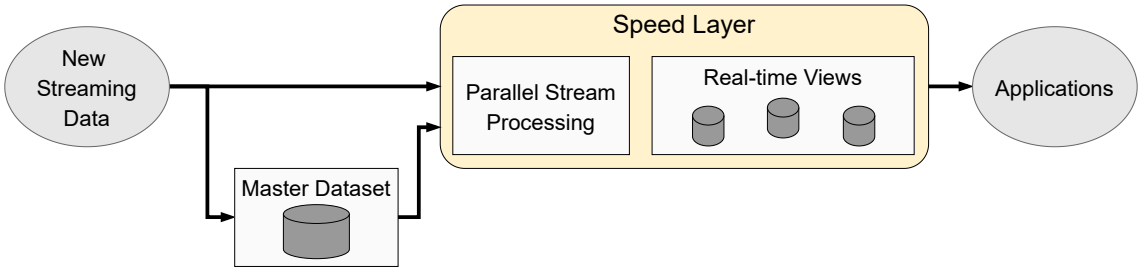

Abbildung 2.3.: Schematische Übersicht der Kappa-Architektur nach Kreps [Kre14].

parallel ausführen kann, ermöglicht sie eine schnelle Verarbeitung der historischen und der aktuellen Daten. Jedoch gelangt die Architektur an ihre Grenzen, wenn die Verarbeitung umfangreicher historischer Daten einen erhöhten Bedarf an Rechenleistung und Speicher bedingt. In solchen Szenarien hat sich die Lambda-Architektur dank ihres Batch-Layers als die geeignetere Wahl erwiesen [GSSM18; KM21].

2.3. Lakehouses

Lakehouses sind ein relativ neues Konzept auf dem Gebiet der Datenplattformen. Im Jahr 2016 begann das Unternehmen Databricks mit der Entwicklung des Lakehouse-Frameworks Delta Lake[1], das drei Jahre später als Open Source veröffentlicht wurde. Delta Lake wird von Databricks als eine ACID-Speicherschicht für Tabellen betrachtet, die auf Cloud-Objektspeichern aufsetzt. Es soll den Aufbau integrierter Datenplattformen ermöglichen, die Eigenschaften moderner Data Lakes mit den Verwaltungsfunktionen traditioneller Data Warehouses kombinieren [ADS+20; AGXZ21]. Neben Delta Lake können auch die beiden alternativen Lakehouse-Frameworks Apache Hudi[2] und Apache Iceberg[3] für diesen Zweck eingesetzt werden. Da der in dieser Masterarbeit entwickelte Prototyp Delta Lake und Apache Iceberg unterstützt, ist eine Beschreibung der technischen Aspekte dieser Lakehouse-Frameworks in Abschnitt 6.1.5 zu finden.

2.3.1. Begriffliche Einordnung

Aus der Literatur und den technischen Fortschritten verschiedener Unternehmen geht hervor, dass es mittlerweile einige Beispiele für die erfolgreiche Umsetzung und den praktischen Einsatz von Lakehouses gibt. Wenn es um die Vision geht, die hinter der Entwicklung von Lakehouses steht, sind sich verschiedene Quellen einig: Diese besteht darin, die vielschichtigen Datenlandschaften und zusammengesetzten Architekturen in Unternehmen zu vereinfachen [AGXZ21; HZ22; SGL+23]. Es ist jedoch anzumerken, dass es bisher keine standardisierte Definition für Lakehouses und keinen Konsens über ihre architektonische Struktur und ihre funktionalen Eigenschaften gibt. Dieser Mangel an Klarheit ist insbesondere auf die Neuartigkeit von Lakehouses zurückzuführen, was zu unterschiedlichen Ansätzen bei ihrer begrifflichen Einordnung führt.

[1] https://delta.io
[2] https://hudi.apache.org
[3] https://iceberg.apache.org

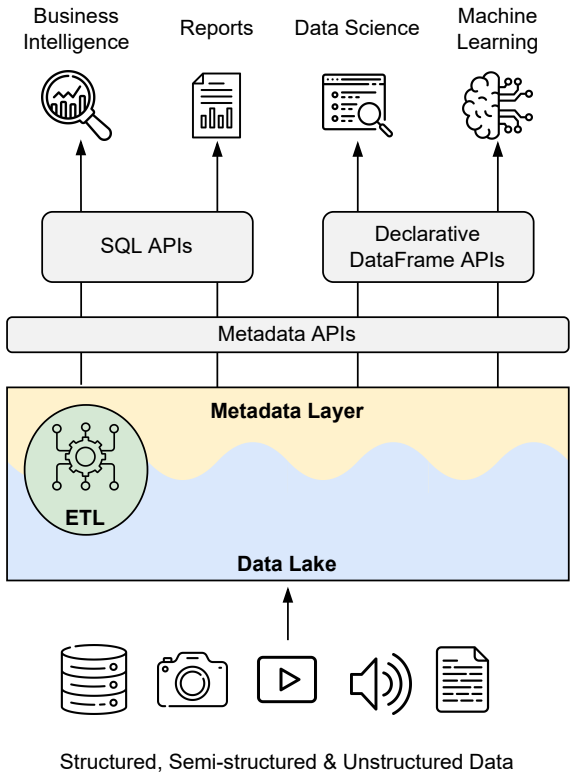

Abbildung 2.4.: Beispiel für die Architektur eines Lakehouses nach Armbrust et al. [AGXZ21].

So wird teilweise die Auffassung vertreten, dass Lakehouses keine neue Art von Datenplattform darstellen, da sich die Fähigkeiten von Data Warehouses und Data Lakes auch auf andere Weise kombinieren lassen. Zur Unterstützung dieser These werden die Merkmale moderner Cloud-basierter Data Warehouses angeführt. Diese haben sich an Data Lakes angenähert, indem sie neben den traditionellen Fähigkeiten eines Data Warehouses beispielsweise auch die Verarbeitung semi-strukturierter Daten und die Erfassung von Streaming-Daten unterstützen [Han21]. Aus der Sicht von Inmon et al. [ISL21] spielen Data Lakes beim Aufbau eines Lakehouses eine entscheidende Rolle. Lakehouses werden als neue Systemarchitektur für Data Lakes betrachtet, die sich durch die Implementierung ähnlicher Datenstrukturen und Verwaltungsfunktionen wie in einem Data Warehouse auszeichnet, aber einen kostengünstigen Speicher integriert, wie er von Data Lakes genutzt wird. Raina und Krishnamurthy [RK21] sehen in einem Lakehouse eine Kombination aus Data Lakes und Data Warehouses. Jedoch raten sie davon ab, den Begriff *Lakehouse* zur Kategorisierung bestimmter Werkzeuge einzusetzen, da die Lakehouse-Plattform Delta Lake beispielsweise sowohl die Kriterien eines Data Lakes als auch eines Lakehouses erfüllt.

Die Entwickler von Delta Lake vertreten in ihren Arbeiten einen anderen Standpunkt und bezeichnen ihr Framework als eines der ersten Lakehouse-Frameworks, das den Aufbau von Lakehouses als integrierte Datenplattformen gestattet [ADS+20; AGXZ21]. Des Weiteren stellen sie eine allgemeine Definition von Lakehouses vor, die von zahlreichen anderen Quellen zitiert wird. Nach Armbrust et al. [AGXZ21] ist ein Lakehouse ein „Datenmanagementsystem, das auf kostengünstigem und direkt zugänglichen Speicher basiert, der auch traditionelle analytische Verwaltungs- und Leistungsmerkmale eines [Datenbank-Managementsystems (DBMS)] bereitstellt, wie ACID-Transaktionen,

Datenversionierung, Auditing, Indizierung, Caching und Abfrageoptimierung.“ Abbildung 2.4 veranschaulicht, wie ihrer Ansicht nach ein Lakehouse architektonisch aufgebaut sein könnte. Schneider et al. [SGL+23] kritisieren die Unschärfe dieser Definition, da sie nicht eindeutig angibt, über welchen Implementierungsansatz die genannten Verwaltungs- und Leistungsmerkmale auf dem Speichersystem realisiert werden sollen. Daher präsentieren sie eine eigene Definition, in der die verschiedenen Arten von analytischen Arbeitslasten im Mittelpunkt stehen, die ein Lakehouse bedienen können sollte. Diese beschreibt ein Lakehouse als „eine integrierte Datenplattform, die denselben Speichertyp und dasselbe Datenformat für Reporting und OLAP, Data Mining und Machine Learning sowie Streaming-Arbeitslasten nutzt.“ Auch betonen die Autoren, dass eine Lakehouse-Architektur eine einzige, integrierte Datenplattform abbilden muss, die nicht durch mehrere separate Datenplattformen implementiert werden sollte. Andernfalls besteht die Gefahr, dass zusätzlich erforderliche Daten-Pipelines und Transformationen die beabsichtigten architektonischen Vereinfachungen untergraben, die eigentlich mit der Lakehouse-Vision verbunden sind.

2.3.2. Merkmale von Lakehouses

Wie sich Lakehouses in der Praxis bewähren und in weiteren Forschungsarbeiten bewertet werden, wird sich in den kommenden Jahren herausstellen. Es zeichnen sich jedoch einige wesentliche Merkmale ab, die im Folgenden erläutert werden.

Als eine zentrale Eigenschaft eines Lakehouses kann angeführt werden, dass die Daten in einem einheitlichen, offenen Dateiformat gespeichert werden [SGL+23]. Ein Beispiel hierfür ist Apache Parquet[4], ein weit verbreitetes, spaltenorientiertes Format zur Ablage von Datensätzen in Objektspeichern. Da die Dateiformate eine effiziente Komprimierung und Kodierung der Daten bieten, ist es möglich, den benötigten Speicherplatz auch bei großen Datenmengen moderat zu halten und effiziente Abfragen der Daten durchzuführen [ISL21]. Insbesondere im Kontext von Machine-Learning-Anwendungen erweist sich diese Art der Datenablage als vorteilhaft, da kosten- und zeitintensive Prozesse wie das Exportieren und Duplizieren von Daten für das Trainieren und Validieren von Machine-Learning-Modellen entfallen, die bei proprietären Datenformaten meist unvermeidbar sind. Wie bei einem Data Lake können solche Anwendungen direkt auf die im Lakehouse gespeicherten Daten zugreifen [ISL21]. Darüber hinaus reduziert die Verwendung offener Dateiformate das Risiko eines sogenannten *Vendor-Lock-Ins*. Dieser Begriff bezeichnet die Abhängigkeit eines Unternehmens von einem bestimmten Anbieter, die das Unternehmen aufgrund des damit verbundenen Aufwands und der Kosten daran hindert, zu einem anderen Anbieter wechseln zu können [ISL21].

Ein weiteres Merkmal von Lakehouses ist ihre Fähigkeit, mehrere Dateien im Speichersystem zu einer zusammenhängenden Datensammlung mit relationalen Eigenschaften zusammenzufassen [SGL+23]. Sofern ein Lakehouse mithilfe eines Lakehouse-Frameworks wie Delta Lake, Apache Hudi oder Apache Iceberg implementiert ist, organisiert und verwaltet es eine Datensammlung auf logischer Ebene in Tabellen. Auf der Speicherebene setzt sich eine Tabelle typischerweise aus einer Sammlung von Dateien zusammen. Diese Sammlung kann wiederum in mehrere separate Dateisätze unterteilt sein, wenn die entsprechende Tabelle nach bestimmten Feldern partitioniert ist [AGXZ21]. Um eine

[4] https://parquet.apache.org

solche relationale Datenablage zu ermöglichen, kann ein Lakehouse technische Metadaten speichern und verwalten. Diese umfassen die verfügbaren Tabellen, ihre Spaltennamen sowie Informationen darüber, in welchen Dateien die Daten-Tupel einer Tabelle gespeichert sind [SGL+23].

In architektonischer Hinsicht befinden sich die Metadaten in einer separaten Schicht über dem Objektspeicher. Durch die Verwendung dieser Metadaten können für den zugrundeliegenden Objektspeicher ähnliche Garantien und Verwaltungsfunktionen durchgesetzt werden, wie sie ein Data Warehouse bietet. So kann die strukturelle Konsistenz von Daten über Datensammlungen hinweg sichergestellt werden, indem ein Lakehouse beispielsweise Unterstützung für Schema-Validierung und Constraint-Checking bietet [SGL+23]. Darüber hinaus wird durch die Sicherstellung von Atomarität und Isolation bei gleichzeitigen Datenzugriffen auf Tabellen verhindert, dass parallel durchgeführte Operationen auf denselben Datensammlungen unvollständige oder inkonsistente Ergebnisse liefern [SGL+23]. Wenn eine Abfrage zum Beispiel mehrere Daten-Tupel in einer Tabelle aktualisieren muss, die sich auf Speicherebene auf mehrere Dateien verteilt, erfolgt die Aktualisierung der betroffenen Dateien atomar. Ein lesende Anwendung muss sich daher nicht mit partiellen Aktualisierungen befassen. Zusammenfassend lässt sich festhalten, dass ein Lakehouse, ähnlich wie ein Data Warehouse, ACID-Eigenschaften bei Datenzugriffen auf eine Tabelle gewährleistet. Mit Hilfe der Metadatenschicht kann außerdem eine Versionierung der Daten vorgenommen werden, was insbesondere die Unterstützung der Time-Travel-Funktion erlaubt [KM21]. Über diese Funktion kann ein Nutzer historische Versionen von Tabellen abrufen und beurteilen, wie sich die Daten im Laufe der Zeit verändert haben [ISL21]. Auf diese Weise können nachträgliche Fehlerkorrekturen oder Berichte über historische Daten erleichtert werden [KM21]. Zudem ist es möglich, Machine-Learning-Modelle exakt zu reproduzieren. Die Time-Travel-Funktion gestattet den Zugriff auf genau die Daten, die zum Trainieren eines Modells verwendet wurden, ohne dass Duplikate der Daten erforderlich sind [ISL21].

Ein weiteres bedeutendes Merkmal von Lakehouses besteht darin, dass Tabellen sowohl als Senken als auch als Quellen für die Batch- und Stream-Verarbeitung dienen können. Atomarität und Isolation beim Zugriff auf eine Tabelle verhindern, dass es zu Anomalien bei gleichzeitigen Datenzugriffen kommt [SGL+23]. Die Vereinheitlichung von Batch- und Stream-Verarbeitung eröffnet neue Möglichkeiten in der Verarbeitung von Batch- und Streaming-Daten. Insbesondere können die komplexen Lambda- und Kappa-Architekturen durch eine einfachere Alternative, die Delta-Architektur, ersetzt werden. Diese wird in Abschnitt 2.3.3 vorgestellt.

Die Merkmale eines Lakehouses werden oft als die Vereinigung der Vorteile von Data Lakes und Data Warehouses dargestellt [AGXZ21; BGK21; OH21; PNM22]. Der Nutzen eines Lakehouses übersteigt jedoch diese Vorstellung. Unternehmen können durch den Einsatz eines Lakehouses einen erheblichen Mehrwert erzielen, da sie verschiedene Arten von analytischen Arbeitslasten von einer einzigen Datenplattform aus verwalten können, wodurch sich der Wartungsaufwand für ihre Dateninfrastruktur verringert. Diese Arbeitslasten fallen in die Bereiche Reporting/OLAP, Advanced Analytics und Streaming [SGL+23]. Innerhalb eines Unternehmens nimmt ein Lakehouse die Rolle einer *Single Source of Truth* ein. Diese Bezeichnung impliziert, dass es zu jedem beliebigen Zeitpunkt nur eine konsistente Version der Daten im Lakehouse gibt, auf die verschiedene Nutzer eines Unternehmens zugreifen. Ein Lakehouse entspricht somit einer konsistenten und konsolidierten Sammlung von Unternehmensdaten und kann als Datenbasis für zuverlässige, datengestützte Entscheidungen herangezogen werden [ISL21].

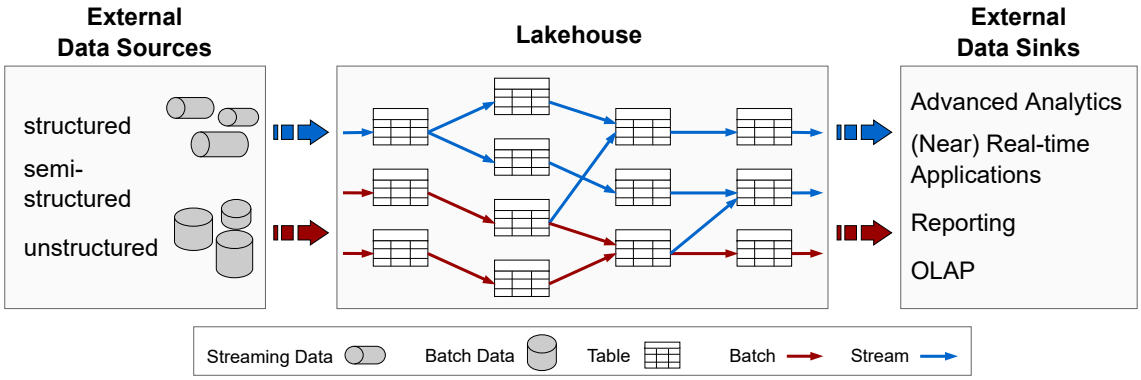

Abbildung 2.5.: Schematische Übersicht über den Aufbau der Delta-Architektur.

2.3.3. Delta-Architektur

Wie im vorigen Abschnitt erwähnt, kann durch die Verwendung eines Lakehouses eine Vereinheitlichung von Batch- und Stream-Verarbeitung erreicht werden. Hierdurch wird die Umsetzung eines neuartigen Architekturmusters ermöglicht, welches als Delta-Architektur bekannt ist [SGL+23] und in Abbildung 2.5 dargestellt ist. Sie zeigt, wie sowohl strukturierte, semi-strukturierte als auch unstrukturierte Batch- und Streaming-Daten in ein Lakehouse eingespeist werden können [AGXZ21]. Diese Daten werden im Rohformat in Tabellen gespeichert und anschließend in mehreren Verarbeitungsschritten weiterverarbeitet, wobei ein Verarbeitungsschritt entweder im Batch- oder im Streaming-Modus durchgeführt wird. Die Zwischenergebnisse der einzelnen Verarbeitungsschritte können im weiteren Verlauf in einer oder mehreren Tabellen persistiert werden. Dabei erfolgt die gerade beschriebene Datenaufnahme und Verarbeitung mithilfe von mehrstufigen Daten-Pipelines, die mehrere Verarbeitungsschritte bündeln. Die Grundidee ihrer Arbeitsweise besteht darin, dass jeder Pipeline-Schritt einer Daten-Pipeline lediglich Daten verarbeitet, die seit seiner letzten Ausführung hinzugefügt oder geändert wurden. Dieses Konzept soll zu einer effizienten Datenverarbeitung innerhalb eines Lakehouses beitragen und den Konsumenten einen schnellen Datenzugriff mit geringer Latenzzeit bieten [ODw21]. Darüber hinaus ist es auch möglich, traditionelle Batch-Verarbeitung in einer Daten-Pipeline durchzuführen, wie sie in Abschnitt 2.2.2 beschrieben wurde.

Aufgrund ihrer Fähigkeit, Batch- und Stream-Verarbeitung auf konsistente Weise zu kombinieren, kann die Delta-Architektur als Alternative zur Lambda- und Kappa-Architektur eines Data Lakes betrachtet werden [SGL+23]. Im Vergleich zu diesen Architekturen bietet die Delta-Architektur eine Reihe von Vorteilen. Dazu gehört eine Reduktion der Komplexität von Daten-Pipelines, die daraus resultiert, dass beide Verarbeitungsmodi auf Ebene des Pipeline-Codes nicht mehr separat behandelt werden müssen. Aus diesem Grund verringert sich auch die Größe der zu pflegenden Codebasis [Lea20]. Im Gegensatz zur Lambda-Architektur ist die Delta-Architektur außerdem transaktionsfähig, wodurch zusätzliche Schritte zur Gewährleistung einer korrekten Datenaufnahme und -verarbeitung entfallen. Zum Beispiel sind Validierungs- und Synchronisierungsschritte überflüssig, die bei der Lambda-Architektur eingesetzt werden, um die Konsistenz von Batch- und Streaming-Daten sicherzustellen. Darüber hinaus haben Datenzugriffe mit ACID-Eigenschaften den Vorteil, dass Inkonsistenzen bei Fehlern in der Daten-Pipeline vermieden und eine automatisierte Fehlerbehebung auf der Grundlage eines konsistenten Datenzustands vereinfacht wird [Lea20].

Ferner geht aus Abbildung 2.5 hervor, dass Tabellen eines Lakehouses verschiedenen Schichten angehören können. Die Delta-Architektur wird auch mit einem Konzept für die logische Organisation von Daten innerhalb eines Lakehouses in Verbindung gebracht [Lea20]. In diesem Zusammenhang werden die Begriffe *Medaillon-Architektur* oder *Multi-Hop-Architektur* verwendet, um den mehrstufigen Aufbau der Tabellenorganisation zu betonen [Dat; HL19]. Indem die Daten in mehrstufigen Daten-Pipelines verarbeitet und die Zwischenergebnisse der einzelnen Pipeline-Schritte in Tabellen gespeichert werden, können Struktur und Qualität der eingespeisten Rohdaten beim Durchlaufen der verschiedenen Schichten der Architektur schrittweise verbessert werden [HL19]. Databricks hat einen dreischichtigen Ansatz für die Organisation eines Lakehouses entwickelt und die Bezeichnungen *Bronze Layer*, *Silver Layer* und *Gold Layer* geprägt [LEs22]. Der Inhalt der einzelnen Schichten kann wie folgt beschrieben werden:

Bronze Layer: Über die Bronze-Schicht werden Daten externer Datenquellen in das Lakehouse aufgenommen und in unverarbeitetem Zustand in Tabellen abgelegt. Diese Rohdaten können auf unterschiedliche Weise verarbeitet werden, ohne dass sie für jede Verarbeitungsiteration erneut vollständig aus den externen Quellsystemen abgefragt werden müssen [HL19].

Silver Layer: Beim Übergang von der Bronze-Schicht zur Silver-Schicht werden verschiedene Transformationen und Bereinigungsverfahren auf den Daten angewendet. Insbesondere können die Daten der Bronze-Schicht gefiltert und kombiniert werden, wobei diese Vorverarbeitungsschritte noch keine Abhängigkeit der Daten zu einem bestimmten Anwendungsfall herstellen. Stattdessen sollen die Tabellen der Silver-Schicht eine einheitliche Sicht auf Geschäftseinheiten, Konzepte und Transaktionen eines Unternehmens bereitstellen [Dat].

Gold Layer: Wenn die Daten die Gold-Schicht erreichen, sind sie bereit für den Konsum durch nachgelagerte externe Datensenken. So dienen die Tabellen der Gold-Schicht beispielsweise als Grundlage für Berichte und Analysen oder sie stellen die Eingabedaten für Machine-Learning-Algorithmen bereit [Dat; HL19].

3. Verwandte Arbeiten

Dieses Kapitel befasst sich mit Publikationen, die mit dem Thema dieser Masterarbeit verwandt sind. Im Rahmen einer Literaturrecherche konnten keine wissenschaftlichen Beiträge gefunden werden, die sich explizit mit der Modellierung von Daten-Pipelines in Lakehouses auseinandersetzen. Deshalb wird das Thema in zwei Teilprobleme aufgegliedert, die in der Literatur bereits behandelt wurden. Abschnitt 3.1 betrachtet Forschungsarbeiten, welche die Modellierung von Daten-Pipelines thematisieren. Deren Ansätze sind auf etablierte Datenplattformen wie Data Warehouses zugeschnitten oder machen keine Aussage in Bezug auf die zu verwendende Datenplattform. Um die Betrachtung des Themas zu komplettieren, wurden für Abschnitt 3.2 Forschungsarbeiten ausgewählt, die Aussagen zur Datenverarbeitung in Lakehouses beinhalten. In Abschnitt 3.3 schließt sich eine Beschreibung des Modellierungsframeworks *Delta Live Tables* an. Dabei handelt es sich um ein kommerzielles Produkt, dessen Funktionsumfang zu großen Teilen als wegweisend für die prototypische Umsetzung der Konzepte eingestuft werden kann, das aber einige von Einschränkungen aufweist. Abschließend zieht Abschnitt 3.4 ein Fazit über die Ergebnisse der Literaturrecherche.

3.1. Modellierung von Daten-Pipelines

Im Fokus von Trujillo und Luján-Mora [TL03] steht die konzeptionelle Modellierung von Extraction-Transformation-Load-Prozessen (ETL-Prozessen), die Daten externer Datenquellen in Data Warehouses einspeisen. Der vorgeschlagene Ansatz sieht vor, diese grafisch in Form von Unified Modeling Language (UML)-Klassendiagrammen zu modellieren. Dazu definieren die Autoren eine Reihe von UML-Stereotypen, die typische Operationen eines ETL-Prozesses widerspiegeln, wie z.B. die Integration verschiedener Datenquellen oder die Aggregation von Attributen. Für jede ETL-Operation wird außerdem eine grafische Notation eingeführt, um Nutzern die Identifizierung bestimmter ETL-Operationen innerhalb eines Modells zu erleichtern. In einem UML-Diagramm sollen die grafischen Elemente zwischen Ein- und Ausgabetabellen der jeweiligen Operation platziert werden. Um umfangreiche ETL-Prozesse zu organisieren und verständlicher darzustellen, verweisen die Autoren auf den Paketmechanismus von UML. Über diesen kann ein ETL-Prozess in einzelne logische Einheiten zerlegt werden, was nicht nur seinen Entwurf, sondern auch seine anschließende Weiterentwicklung wesentlich vereinfachen soll. Außerdem unterstützt der Modellierungsansatz verschiedene Detailebenen. Je nach den Fähigkeiten des Modellierers kann sich das Modell entweder auf einen allgemeinen Überblick über den ETL-Prozess beschränken oder eine detaillierte Beschreibung der einzelnen Transformationen beinhalten.

Raj et al. [RBOW20] schlagen in ihrer Arbeit ein konzeptionelles Modell für vollautomatisierte und fehlertolerante Daten-Pipelines vor. Der Entwurf des Modells geht auf eine explorative Fallstudie zurück, mit welcher die Autoren mehrere Daten-Pipelines eines großen Telekommunikationsunternehmens untersuchen. In den Interviews berichten die Fachleute über typische Herausforderungen, mit denen sie sich bei der Datenverwaltung und bei der Nutzung der Daten-Pipelines konfrontiert

sehen. Das vorgeschlagene Modell zielt darauf ab, bei der Bewältigung dieser Herausforderungen zu unterstützen. Es identifiziert insbesondere die Schritte einer Daten-Pipeline, in denen automatische Mechanismen zur Überwachung, Fehlererkennung und -behebung, sowie zur Fehlereindämmung und -bekanntmachung umgesetzt werden sollten. Die Elemente zur Darstellung des Modells sind in einem separaten Metamodell organisiert. Dieses schreibt vor, dass sich eine Daten-Pipeline aus Knoten und Verbindungen zusammensetzt. Für jeden der Bausteine werden verschiedene Typen unterschieden. Beispielsweise sagt der Typ einer Verbindung aus, ob dieser die Daten zwischen zwei Knoten als Batch oder kontinuierlich überträgt. Auf Ebene des Modells legen die Autoren die konkrete Reihenfolge der auf den Daten durchgeführten Aktivitäten fest. Über den Knoten *Data Collection* werden Daten verschiedener externer Datenquellen im Rohformat in einen Data Lake eingespeist. Nach ihrer Verarbeitung durch den Knoten *Data Processing* liegen die Daten in einem strukturierten oder semi-strukturierten Format vor, was ihre Ablage in einem Data Warehouse ermöglicht. Über die drei Knoten *Data Labeling*, *Data Preprocessing* und *ML/DL models* wird die Umsetzung von Anwendungen aus dem Bereich des Machine Learnings berücksichtigt. Über Annotationen von Verbindungen drückt das Modell aus, wie der Datenfluss zwischen Knoten überwacht und durch Strategien zur Fehlereindämmung abgesichert werden sollte. Die Autoren kommen zum Schluss, dass das Modell als Referenz beim Aufbau von Daten-Pipelines eingesetzt werden sollte. Es verbessert die Kommunikation zwischen verschiedenen Teams eines Unternehmens und verhindert manuellen Aufwand, wenn es um die Bewältigung typischer Herausforderungen im Kontext von Daten-Pipelines geht.

Der Beitrag von Eichelberger et al. [EQSS16] bettet sich in den Kontext des EU-Projekts *QualiMaster* ein, das den Aufbau einer konfigurierbaren und anpassungsfähigen Big-Data-Infrastruktur anstrebt. Für dieses Projekt beschreiben die Autoren einen Ansatz, wie topologische Variabilitätsmodellierung auf Streamverarbeitungspipelines angewendet werden kann. Über ein Variabilitätsmodell lässt sich in der Softwareentwicklung abbilden, wie ein System konfiguriert oder angepasst werden kann, um die Anforderungen verschiedener Kontexte zu erfüllen. Dies soll die Wiederverwendung der Software erleichtern und den manuellen Aufwand im Entwicklungsprozess reduzieren, was insbesondere beim Aufsetzen einer neuen Streamverarbeitungspipeline einen wesentlichen Zeit- und Kostenfaktor darstellt. Strukturell wird eine Streamverarbeitungspipeline als potentiell zyklischer Datenflussgraph betrachtet, der sich aus Datenquellen und Prozessierungsknoten zusammensetzt. Ihre Modellierung und Konfiguration erfolgt mithilfe der Integrated Variability Modeling Language (IVML). Diese bietet sich als Sprache für die Variabilitätsmodellierung an, da sie auch topologische Informationen einer Pipeline abbilden kann. Jeder Prozessierungsknoten ist als typisierte Variable definiert und identifiziert nachgelagerte Knoten über Referenzen zu anderen Variablen. Über eine Konfiguration muss sein Name und eine Algorithmenfamilie festgelegt werden, die Verarbeitungsalgorithmen mit ähnlicher Funktionalität, aber unterschiedlichen Laufzeiteigenschaften zusammenfasst. Erst zur Laufzeit wird durch Eigenschaften der Infrastruktur entschieden, welcher Verarbeitungsalgorithmus einer Algorithmenfamilie tatsächlich auf den Daten ausgeführt wird. Für eine gültige Konfiguration muss der Ersteller einer Streamverarbeitungspipeline die Bedingungen des Variabilitätsmodells erfüllen und die Sprache IVML beherrschen. Die Autoren betonen deren gute Erlernbarkeit, verweisen aber zusätzlich auf eine grafischen Benutzeroberfläche. Über diese können Streamverarbeitungspipelines mithilfe eines Drag-und-Drop-Ansatzes aufgebaut werden.

Eine andere Arbeitsweise verbindet sich mit dem Framework *Pipeline61*. Wu et al. [WZX+16] haben es entwickelt, um das Erstellen von Daten-Pipelines in heterogenen Umgebungen zu vereinfachen. Ohne Werkzeugunterstützung gestaltet sich die Verwaltung und Wartung derartiger Daten-Pipelines mühsam. Verschiedene Pipeline-Phasen können über unterschiedliche Technologien implementiert

sein, was ihre Integration über *Glue Code* erforderlich macht. Bei Verwendung des entwickelten Framework Pipeline61 kann auf Glue Code verzichtet werden. Gemäß des zugrundeliegenden Pipeline-Modells kann eine Pipeline-Komponente entweder in einer Spark-, MapReduce- oder Shell-Umgebung arbeiten. Außerdem kann festgelegt werden, welche Pfade und Formate für die Eingabedaten vorgesehen sind und in welchem Speichersystem die Ausgabe persistiert werden soll. Die einzelnen Pipeline-Komponenten lassen sich auf logischer Ebene nahtlos zu einem Directed Acyclic Graph (DAG) verbinden. Erst zur Laufzeit der Daten-Pipeline werden sie von einem Scheduler getrennt an die passende Umgebung übermittelt und unabhängig ausgeführt. Um Nachvollziehbarkeit in Bezug auf Änderungen und Weiterentwicklungen herzustellen, besitzt Pipeline61 eine integrierte Versionskontrolle. Dazu speichert das Werkzeug den Datenflussgraphen für jede Ausführungsinstanz einer Pipeline und alle historischen Metadaten für verschiedene Versionen einer Pipeline-Komponente. Anhand dieser Informationen können Nutzer die Änderungshistorie einer Pipeline einsehen und Ergebnisse älterer Pipeline-Versionen reproduzieren.

3.2. Datenverarbeitung in Lakehouses

Orescanin und Hlupic [OH21] beschreiben in ihrer Arbeit eine mögliche Architektur eines Lakehouses. In ihrem Vorschlag kombinieren die Autoren einen Data Lake und ein Data Warehouse, um der resultierenden, zusammengesetzten Plattform typische Lakehouse-Eigenschaften zu verleihen. Auf feinerer Ebene betrachtet ist der Speicher des Data Lakes in zwei Bereiche untergliedert. Die eingespeisten Daten externer Datenquellen werden zunächst in der *Landing Area* bereitgestellt, um durch einen Prozess anschließend in der *Foundation Area* abgelegt zu werden. Von dort überführt ein ETL-Prozess die Daten als Batch in den *Base Layer* des Data Warehouses. Um die Daten des Base Layers für Analysezwecke zu verwenden, können sie auf dem dafür vorgesehenen *Performance & Analytics Layer* zur Nutzung bereitgestellt werden. Die Kombination beider Datenplattformen erfordert die Ausführung und Verwaltung einer Reihe von plattformübergreifenden Datenprozessen. Als Lösung schlagen die Autoren den Einsatz eines Process Control Framework (PCF) vor, das sie als alternatives Orchestrierungswerkzeug zu Apache Airflow einordnen. Dieses ist für die Planung und Ausführung der Datenprozesse verantwortlich, wobei es sich unter anderem um deren Parallelisierung und Leistungsoptimierung zur Laufzeit kümmert. Außerdem berücksichtigt das PCF mögliche Abhängigkeiten zwischen den Datenprozessen und Voraussetzungen, die vor ihrem Start erfüllt sein müssen.

Hambardzumyan et al. [HTG+22] stellen die Lakehouse-Implementierung *Deep Lake* vor, die speziell auf Deep-Learning-Anwendungen ausgelegt ist. Deep Lake speichert seinen Inhalt als Tensoren in einem Objektspeicher, z.B. in Amazon S3. Daher kann das Lakehouse sowohl Tabellendaten als auch komplexere Bild- oder Videodaten aufnehmen. Um Deep-Lake-Daten zu nutzen, können sie als Stream zu Deep-Learning-Frameworks übertragen und in eine neue Trainingsiteration eingegeben werden. An deren Ende stehen gelabelte Daten, die zwecks Speicherung zum Deep Lake zurückgeführt werden können. Als Ergänzung zu Deep Lake haben die Autoren ein ETL-Verbindungswerkzeug auf Basis von Airbyte entwickelt. Dieses unterstützt die Synchronisierung der Deep-Lake-Daten mit verschiedenen externen Datenquellen und Datensenken, darunter SQL- und NoSQL-Datenbanken sowie Data Lakes und Data Warehouses. Für die Datenverarbeitung innerhalb des Lakehouses bietet Deep Lake die Option, Transformationen in Python-Skripten zu definieren und parallel auszuführen. Auf programmatischer Ebene entspricht eine Transformation einer einzelnen benutzerdefinierten Python-Funktion, die als Eingabe einen vorhandenen Datensatz erwartet und

als Ausgabe einen neuen Datensatz erzeugt oder eine Aktualisierung der Eingabe vornimmt. Dabei werden sowohl Eins-zu-Eins- als auch Eins-zu-Viele-Transformationen zwischen Ein- und Ausgabedatensätzen unterstützt. Um komplexe Verarbeitungsaufgaben zu bewältigen, besteht außerdem die Möglichkeit, mehrere Transformationen in einer Pipeline zusammenzuführen.

Begoli et al. [BGK21] verwenden ein Lakehouse zur Verwaltung und Analyse heterogener Daten für die biomedizinische Forschung. Dabei sehen die Autoren die größte Schwierigkeit darin, die Daten effizient und konsistent zu sammeln, zu organisieren und für die Nutzung durch Forscher aufzubereiten, ohne Zugangskontrollen zum Schutz sensibler Daten zu vernachlässigen. Aus ihren Erfahrungen leiten sie eine Sammlung von Prinzipien und Leitlinien ab, die den Umgang mit diesen Herausforderungen erleichtern sollen. Im Kontext der datenverarbeitenden Prozesse eines Lakehouses identifizieren die Autoren drei Arten von Workflows, die sich in der Praxis als relevant erwiesen haben und repräsentativ genug sind, um auf beliebige Anwendungsszenarien übertragen zu werden. Der Workflow *Data Intake with Minimal Pre-processing* steht für die Einspeisung von Daten, die methodisch ausgewählt und mit ausreichenden Metadaten versehen sind. Diese müssen nur minimal vorverarbeitet werden, bevor sie im Lakehouse abgelegt werden. Davon abzugrenzen sind unverknüpfte oder unstrukturierte Daten, für deren Ladeprozess der Workflow *Intensive Processing* vorgesehen ist. Die Beschaffenheit der Daten macht eine umfangreiche Verarbeitung erforderlich, im Rahmen derer die Daten bereinigt, verlinkt und mit Metadaten für die nachgelagerte Analyse angereichert werden können. Schließlich bezieht sich der Workflow *Ongoing Re-processing* auf die Weiterverarbeitung der Daten innerhalb des Lakehouses. Er umfasst sowohl die Ableitung neuer Datensätze als auch deren Katalogisierung und effiziente Organisation.

3.3. Delta Live Tables

Delta Live Tables (DLT)[1] ist ein Framework des Unternehmens Databricks, das die Definition, Ausführung und Überwachung von Daten-Pipelines in einem mittels Delta Lake realisierten Lakehouse unterstützt. Es verfolgt einen deklarativen Ansatz, was die Umsetzung komplexer Datenverarbeitungsaufgaben vereinfachen soll, und übernimmt außerdem die automatische Verwaltung der Dateninfrastruktur. Dabei betrachtet DLT eine Pipeline als DAG, der Datenquellen mit Tabellen verknüpft. Um eine neue DLT-Pipeline zu erstellen, muss ein Nutzer den Inhalt einer DLT-Tabelle mithilfe von SQL-Abfragen oder Python-Funktionen in einem Databricks Notebook definieren. Abhängig davon, wie er die Ergebnisse eines Verarbeitungsschrittes verwenden will, kann er zwischen zwei verschiedenen Typen von DLT-Tabellen wählen. Die Daten eines sogenannten *Views* sind nur innerhalb einer Pipeline zugreifbar und können nicht interaktiv abgefragt werden. Hingegen werden die Daten einer Tabelle im Delta-Format persistiert, was eine Weiterverwendung durch andere Daten-Pipelines ermöglicht. Eine weitere Unterscheidungsdimension der DLT-Tabellen ergibt sich anhand der Fähigkeit von DLT, sowohl Batch- als auch Streaming-Daten verarbeiten zu können. Live-Views und Live-Tabellen sind auf Batch-Verarbeitung von Daten ausgelegt. Ändert sich die dahinterliegende Abfrage oder eingespeiste Datenquelle, passt sich der Inhalt der betroffenen DLT-Tabellen bei der nächsten Ausführung entsprechend der Änderung an. Zur Stream-Verarbeitung eignen sich dagegen Streaming-Live-Views oder Streaming-Live-Tabellen, da lediglich die verfügbaren Daten seit der letzten Ausführung prozessiert werden. Die Änderung der definierenden

[1] Delta Live Tables: https://docs.databricks.com/delta-live-tables/index.html

Abfrage spiegelt sich in diesem Fall nicht in bereits verarbeiteten Daten wieder. Zusätzlich zu den bisher vorgestellten Konzepten bietet DLT die Möglichkeit, Datenqualitätseinschränkungen auf dem Inhalt einer Tabelle durchzusetzen. Über eine sogenannte *Expectation* kann ein Nutzer eine Aktion spezifizieren, die dann angewendet wird, wenn das betrachtete Daten-Tupel eine festgelegte Invariante verletzt. Die Aktion sieht dabei entweder vor, das Daten-Tupel zu behalten, zu löschen oder die Daten-Pipeline anzuhalten. Über ein Event-Log kann der Nutzer bei Ausführung der Daten-Pipeline nachvollziehen, durch wie viele Daten-Tupel eine Expectation verletzt wird.

Auf Basis der deklarativen Beschreibung einer Daten-Pipeline kann DLT ableiten, welche Transformationen zwischen den beteiligten DLT-Tabellen zur Laufzeit erforderlich sind. Außerdem kann das Modellierungsframework automatisch analysieren, welche Abhängigkeiten zwischen den DLT-Tabellen existieren, um die Verarbeitungsschritte in passender Reihenfolge abzuarbeiten. Dem Nutzer stehen zur Steuerung der Ausführung zwei Ausführungsmodi zur Wahl. Diese bestimmten insbesondere, wie sich der Umgang mit dem Spark-Cluster gestaltet, auf welchem die Prozessierung der Daten stattfindet. Bei getriggerten Pipelines wird jede Tabelle mit den aktuell verfügbaren Daten aktualisiert. Danach wird das Spark-Cluster gestoppt, damit es zu keinem unnötigen Ressourcenverbrauch kommt. Die geringeren Kosten dieses Ausführungsmodus gehen mit einer höheren Verarbeitungslatenz einher, da neu ankommende Daten erst bei der nächsten Ausführung verarbeitet werden. Wenn Daten jedoch echtzeitnah benötigt werden und ein höherer Ressourcenverbrauch akzeptabel ist, bietet sich eine kontinuierliche Pipeline an. Dank eines ständig laufenden Spark-Clusters können neue Daten mit geringerer Verzögerung prozessiert werden. Auch auf das Aktualisierungsverhalten einzelner Tabellen kann ein Nutzer Einfluss nehmen. Beispielsweise wird über die Option *Refresh All* erreicht, dass im Rahmen einer Ausführung alle Tabellen auf den aktuellen Stand gebracht werden, während durch *Refresh Selection* lediglich der Inhalt ausgewählter Tabellen aktualisiert wird.

Zusammenfassend geht aus der Beschreibung von DLT hervor, dass das Modellierungsframework durch eine Reihe vorteilhafter Eigenschaften gekennzeichnet ist. Dazu zählen die Möglichkeit zur deklarativen Definition von Daten-Pipelines, die Unterstützung zur Überwachung und Durchsetzung von Datenqualitätsbeschränkungen und die automatische Verwaltung der Dateninfrastruktur, durch die erhebliche Komplexität bei der Ausbringung einer Daten-Pipeline auf einem Spark-Cluster wegfällt. Mit DLT verbinden sich jedoch mehrere Nachteile, die vor dem Einsatz des Modellierungsframeworks zu bedenken sind. Da es auf das Lakehouse-Framework Delta Lake beschränkt ist, besteht die Gefahr eines Vendor-Lock-Ins. Unternehmen, die ihre Daten-Pipelines mithilfe von DLT definieren und ausführen, machen sich von Delta Lake abhängig. Alle Ausgaben von DLT müssen im Delta-Format persistiert werden, obwohl die Eigenschaften eines anderen Lakehouse-Frameworks möglicherweise besser zu den Anforderungen eines Unternehmens passen würden. Auch auf struktureller Ebene sind DLT-Pipelines als wenig flexibel einzuordnen. Die Abstraktion als DAG verhindert zyklische Abhängigkeiten von Verarbeitungsschritten. Innerhalb einer Daten-Pipeline ist es deshalb nicht möglich, die Ausgabe eines Pipeline-Schritts in einem vorgelagerten Pipeline-Schritt einzubeziehen. Eine weitere Einschränkung ergibt sich für die Kombination von Ergebnissen mehrerer Abfragen in derselben Tabelle. DLT verlangt, dass jede Tabelle genau einmal definiert wird. Wenn ein Nutzer die Ergebnisse mehrerer Abfragen in derselben Tabelle persistieren möchte, muss er beide Tabellen in einer Abfrage über einen UNION-Befehl zusammenführen, was einen zusätzlichen Verarbeitungsschritt bedingt.

3.4. Fazit

Die gesichteten Publikationen weisen Ähnlichkeiten zum Thema dieser Masterarbeit auf. Aufgrund anderer thematischer Schwerpunkte lassen sich ihre Lösungsansätze jedoch nicht direkt auf die Problemstellung dieser Masterarbeit anwenden. In Bezug auf die Modellierung von Daten-Pipelines verdeutlichen sie den großen Anwendungsbereich von Modellen. Die Vorschläge von Trujillo und Luján-Mora [TL03] und Raj et al. [RBOW20] konzentrieren sich auf die visuelle Darstellung und Kommunikation von komplexen Abläufen. Sie sollen das Verständnis der Beteiligten unterstützen und können als Orientierung herangezogen werden, wenn es um die technische Umsetzung der im Modell visualisierten Prozesse geht. Verglichen dazu sind die Pipeline-Modelle bei Eichelberger et al. [EQSS16] und Wu et al. [WZX+16] als konkreter einzuordnen. Sie sind Teil eines Softwareentwicklungsansatzes und zielen darauf, alle Informationen zu organisieren, die für die programmatische Umsetzung und Automatisierung der Prozesse erforderlich sind. Eine solche Verwendung von Modellen steht auch im Fokus dieser Masterarbeit. Das Metamodell einer Daten-Pipeline soll umfassende Beschreibungsmöglichkeiten vorsehen, so dass auf Modellebene ein genaues Bild der Datenflüsse hervorgeht. Nach einer Anreicherung um technologiespezifische Aspekte soll ein Pipeline-Modell außerdem die Generierung von ausführbarem Pipeline-Code zulassen. Dabei steht als Datenplattform das Lakehouse im Zentrum der Modellierung. Die Literatur zu diesem Themenbereich ist aufgrund der Neuartigkeit von Lakehouses begrenzt. Die drei vorgestellten Arbeiten beziehen sich auf unterschiedliche Lakehouse-Implementierungen. Da bisher keine Referenzarchitektur für Lakehouses existiert, fehlt die Grundlage für eine objektive Bewertung ihrer Eigenschaften. Bezogen auf die datenverarbeitenden Prozesse innerhalb eines Lakehouses bleibt es bei vagen Aussagen, auch wenn deren Notwendigkeit von allen berücksichtigt wird. Die konkretesten Anhaltspunkte, wie diese organisiert und technisch umgesetzt werden könnten, liefern Hambardzumyan et al. [HTG+22]. Es ist jedoch davon auszugehen, dass die Lösung auf das spezialisierte Lakehouse Deep Lake zugeschnitten ist. Diese Masterarbeit fokussiert sich auf Lakehouses, wie sie mithilfe der Lakehouse-Frameworks Delta Lake, Apache Hudi und Apache Iceberg umgesetzt werden können. Obwohl sich die Lakehouse-Frameworks hinsichtlich ihrer technischen Umsetzung und ihres Funktionsumfangs unterscheiden, ermöglichen sie die Umsetzung von Lakehouses als integrierte Datenplattformen.

Auch in der Existenz des kommerziellen Modellierungsframeworks DLT drückt sich die Relevanz von Daten-Pipelines in Lakehouses aus. Dieses Werkzeug kann als Beweis für die technische Durchführbarkeit einer Lösung dienen, es ist jedoch auf die Definition und Ausführung von Daten-Pipelines in Lakehouses beschränkt, die mit dem Lakehouse-Framework Delta Lake realisiert wurden. Diese Masterarbeit zielt darauf ab, die plattformunabhängigen Aspekte einer Daten-Pipeline herauszuarbeiten, um die Grundlage für ein plattformunabhängiges Modellierungswerkzeug zu schaffen. Dadurch soll verhindert werden, dass sich ein Unternehmen von einem bestimmten Anbieter abhängig macht. Des Weiteren soll die Abstraktion einer Daten-Pipeline im Gegensatz zu DLT nicht als DAG erfolgen, um zyklische Abhängigkeiten zwischen Pipeline-Schritten zu gestatten. Dann können auch selbstanpassende Daten-Pipelines umgesetzt werden, bei denen die Konfiguration eines Pipeline-Schritts auf Basis der Verarbeitungsergebnisse nachgelagerter Pipeline-Schritte angepasst wird. Ein weiterer Ansatzpunkt dieser Masterarbeit ergibt sich daraus, dass DLT keine umfassenden Beschreibungsmöglichkeiten vorsieht, mit welchen die Pipeline-Schritte einer Daten-Pipeline den verschiedenen Zonen eines Zonenmodells zugeordnet werden können. Zu diesem Zweck stützt sich diese Masterarbeit auf das Metamodell und das Zonenreferenzmodell für Zonen in Data Lakes, die bereits in Abschnitt 2.2.1 vorgestellt wurden.

4. Anforderungen

Dieses Kapitel umfasst eine Sammlung von Anforderungen, die sich an ein Metamodell für Daten-Pipelines in Lakehouses stellen. Das Metamodell soll in der Lage sein, Daten-Pipelines in Lakehouses abzubilden, weshalb es die Eigenschaften und Fähigkeiten dieser Datenplattform einbeziehen muss. Weitere Eckpunkte für die Anforderungsanalyse lassen sich aus der Forschung zu Data Lakes ableiten, die eine Reihe von Vorschlägen für die Verwaltung und Organisation der Daten innerhalb von Data Lakes liefert. Die Anforderungen gehen jedoch nicht darauf ein, welche Lakehouse-Implementierung zugrunde liegt und welche Technologie zur Ausführung der Verarbeitungslogik verwendet wird, da das Metamodell eine plattformunabhängige Sicht auf alle relevanten Aspekte einer Daten-Pipeline gestatten und insbesondere deren strukturelle Eigenschaften modellieren soll. Alle identifizierten Anforderungen können in Tabelle 4.1 eingesehen werden. Im Folgenden werden diese einzeln erläutert und begründet, wieso sie bei dem Entwurf eines Metamodells für Daten-Pipelines in Lakehouses berücksichtigt werden sollten.

A1: Berücksichtigung der Delta-Architektur

Ein Lakehouse ist dazu in der Lage, ACID-Eigenschaften beim Zugriff auf Tabellen zu gewährleisten [SGL+23]. Aufgrund dieser Fähigkeit ermöglicht ein Lakehouse die Umsetzung der Delta-Architektur, für die sich eine getrennte Behandlung von Batch- und Streaming-Daten erübrigt. Diese sieht vor, dass Tabellen sowohl für Batch-Verarbeitung, als auch als Datenquellen und Datensenken für Stream-Verarbeitung genutzt werden. Deshalb können die in einer Tabelle gespeicherten Daten entweder durch Batch- oder Stream-Verarbeitung weiterverarbeitet werden. Zu diesem Zweck werden Daten-Pipelines eingesetzt, die Verarbeitungsschritte entweder im Batch-

#	Name
A1	Berücksichtigung der Delta-Architektur
A2	Modularer Aufbau
A3	Flexibilität und Änderbarkeit
A4	Beschreibung der Eigenschaften externer Datenquellen
A5	Integration von Daten verschiedener externer Datenquellen
A6	Speicherung von Zwischenergebnissen
A7	Unterstützung komplexer Filter- und Bereinigungsoperationen
A8	Unterstützung selbstanpassender Daten-Pipelines
A9	Bezug zu einem Zonenmodell
A10	Berücksichtigung von Data Governance

Tabelle 4.1.: Übersicht über die identifizierten Anforderungen an ein Metamodell für Daten-Pipelines in Lakehouses.

oder im Streaming-Modus ausführen. Wenn alle Verarbeitungsschritte für die Stream-Verarbeitung konzipiert sind, können auch Anwendungen für nahe Echtzeit unterstützt werden. Solche externen Datensenken verlangen, dass Streaming-Daten mit möglichst geringer Verzögerung verarbeitet und bereitgestellt werden. Es ist jedoch anzumerken, dass Verarbeitungsschritte innerhalb einer einzigen Daten-Pipeline beide Verarbeitungsmodi verwenden können. An das Metamodell stellt sich die Anforderung, dass es die Eigenschaften der Delta-Architektur berücksichtigen muss. Es sollte daher Tabellen im Lakehouse als Datenquellen und als Datensenken betrachten können, sowohl für Batch- als auch Stream-Verarbeitung. Zudem sollte das Metamodell beachten, dass der Verarbeitungsmodus auf Ebene einzelner Verarbeitungsschritte festgesetzt werden kann, statt diesen global auf die gesamte Daten-Pipeline zu beziehen.

A2: Modularer Aufbau

Eine Daten-Pipeline umfasst Datenverarbeitungsprozesse, die zwischen der Datenerzeugung durch externe Datenquellen und dem Datenempfang durch externe Datensenken stattfinden. Konzeptionell besteht sie aus einer Reihe von miteinander verbundenen Verarbeitungsschritten, wobei die Ausgabe ein oder mehrerer Verarbeitungsschritte als Eingabe für einen anderen Verarbeitungsschritt dient [MBO20]. Zu diesem Zweck materialisiert jeder Verarbeitungsschritt seine Zwischenergebnisse in einer Tabelle, auf die andere Verarbeitungsschritte zugreifen können. In Bezug auf einen Verarbeitungsschritt gilt, dass dieser unabhängig von den anderen auf die Erledigung einer einfachen Teilaufgabe abgestimmt ist, die im Rahmen einer größeren, komplexeren Gesamtaufgabe anfällt [RSGJ13]. Die Zerlegung in einzelne Verarbeitungsschritte führt zu einem modularen Aufbau, aus dem sich einige Vorteile sowohl für den Entwurf als auch für die Ausbringung und Wartung einer Daten-Pipeline ergeben. Insbesondere der Entwicklungsaufwand wird reduziert. Durch die Wiederverwendung von Verarbeitungsschritten über mehrere Daten-Pipelines hinweg können Entwickler ihren Arbeitsaufwand verringern, weil sie neue Daten-Pipelines nicht von Grund auf neu definieren müssen. Außerdem gestaltet sich die Änderung einer Daten-Pipeline weniger kompliziert. Bei Bedarf können einzelne Verarbeitungsschritte ausgetauscht, hinzugefügt oder aus einer Daten-Pipeline entfernt werden. Die Möglichkeit, jeden Verarbeitungsschritt einzeln zu testen, vereinfacht auch die Fehlersuche und ermöglicht eine schnellere Behebung von Problemen [MBO20]. Aufgrund dieser Aspekte ist zu fordern, dass das Metamodell einen modularen Aufbau für Daten-Pipelines vorschreibt. Dies beinhaltet, dass es die Definition in sich abgeschlossener Daten-Pipelines erlauben sollte, die sich aus mehreren Verarbeitungsschritten zusammensetzen. Für einen einzelnen Verarbeitungsschritt sollte aus dem Metamodell hervorgehen, dass dieser auf eine bestimmte Aufgabe ausgerichtet und unabhängig von anderen Verarbeitungsschritten ist. Dadurch soll das Metamodell sicherstellen, dass die Reihenfolge der Verarbeitungsschritte innerhalb einer Daten-Pipeline geändert werden kann und Verarbeitungsschritte ausgetauscht, hinzugefügt, entfernt und über mehrere Daten-Pipelines hinweg wiederverwendet werden können. Darüber hinaus sollte das Metamodell die Erstellung von Unterpipelines innerhalb einer Hauptpipeline zulassen, die unabhängig von der Hauptpipeline definiert und geändert werden können. Wenn eine Änderung in einer Unterpipeline erforderlich wird, sollte sich ein Nutzer lediglich mit dieser befassen und keine Auswirkungen auf die Hauptpipeline bedenken müssen.

A3: Flexibilität und Änderbarkeit

Big Data hat das Potential, die Entscheidungsfindung und Effizienz von Unternehmen wesentlich zu verbessern. Um den Wert aus Big Data zu erschließen, müssen Daten-Pipelines vielfältige Verarbeitungsaufgaben übernehmen. Dazu zählen beispielsweise die Datenbereinigung und -transformation, Aggregation, Identifikation von Features und Training von Modellen für Machine Learning [RSGJ13]. Gleichzeitig muss damit umgegangen werden, dass sich die Geschäftsanforderungen eines Unternehmens im Laufe der Zeit ändern und auch die Eigenschaften externer Datenquellen Änderungen unterliegen können. Diese Entwicklungen müssen in Daten-Pipelines berücksichtigt werden, damit sie weiterhin die vorgesehen Resultate erzielen. Aus diesen Gründen muss das Metamodell auf Flexibilität und Änderbarkeit ausgelegt sein. Es sollte den Nutzern erlauben, die Verarbeitungslogik einzelner Verarbeitungsschritte durch benutzerdefinierten Code zu spezifizieren. Dem Code sollten möglichst wenig Einschränkungen auferlegt sein, um ein breites Spektrum an Verarbeitungsaufgaben unterstützen zu können. Genauer bedeutet dies, dass auch die Einbindung externer Bibliotheken und anderer Ressourcen gestattet werden sollte, damit Verarbeitungsschritte nicht nur als Anfragen in einer Abfragesprache spezifiziert werden können. Dabei ist hervorzuheben, dass durch den Code keine Abhängigkeiten zwischen Verarbeitungsschritten eingeführt werden dürfen, da diese die einfache Änderbarkeit und den modularen Aufbau einer Daten-Pipeline, wie er in Anforderung A2 gefordert wird, behindern würden.

A4: Beschreibung der Eigenschaften externer Datenquellen

Lakehouses werden als analytische Datenplattformen eingeordnet, die Big Data speichern und verwalten können [AGXZ21]. Im Kontext von Big Data nehmen externe Datenquellen verschiedene Formen an, die durch die 4 V's *Volume*, *Velocity*, *Variety* und *Value* charakterisiert werden können. Dabei bezieht sich das Volumen auf die riesige Menge der gesammelten Daten. Diese können mit großer Geschwindigkeit erzeugt werden und vielfältige Datentypen aufweisen. Der Wert von Big Data gibt an, ob die Daten im Rahmen einer Analyse wertvolle Erkenntnisse liefern können. Bei der Konstruktion von Daten-Pipelines muss das Abrufen großer Datenmengen von einer Vielzahl heterogener Quellen und die Notwendigkeit, unterschiedliche Verarbeitungsgeschwindigkeiten zu bewältigen, miteinbezogen werden [KW18]. Aus den Eigenschaften externer Datenquellen können sich Anforderungen an die Arbeitsweise der Daten-Pipeline ergeben, die erfüllt werden müssen, um eine korrekte und effiziente Verarbeitung der Daten zu gewährleisten. Angesichts dieser Überlegungen ist es erforderlich, dass das Metamodell eine Möglichkeit zur Modellierung der Eigenschaften externer Datenquellen bietet. Der Umfang dieser beschreibenden Annotationen soll ausreichend sein, um externe Datenquellen zu beurteilen und ein besseres Verständnis für Ein- und Ausgaben einer Daten-Pipeline zu entwickeln. Dadurch sollen Nutzer unterstützt werden, fundierte Entscheidungen bei der Definition und Wartung einer Daten-Pipeline zu treffen. Zum Beispiel soll der Einbezug dieser Informationen helfen, angemessene Datenbereinigungsmaßnahmen auszuwählen, eine korrekte Integration mehrere Datenquellen durchzuführen oder die Qualität der Verarbeitungsergebnisse besser einzuschätzen.

A5: Integration von Daten verschiedener externer Datenquellen

Wie in Kapitel 2 beschrieben, soll ein Lakehouse die Rolle einer *Single Source of Truth* für Unternehmen einnehmen. Es soll eine konsolidierte und konsistente Datenquelle darstellen, die über alle Teams und Abteilungen eines Unternehmens hinweg genutzt werden kann. Dazu ist es nicht ausreichend, Daten aus verschiedenen externen Datenquellen in ein Lakehouse einzuspeisen. Vielmehr müssen diese Daten integriert werden [GGH+20], was eine Reihe von Aufgaben erforderlich macht. Diese beinhalten unter anderem, die vielfältigen Schemata von Datensätzen auf ein einheitliches, globales Schema abzubilden. Auf Ebene einzelner Datentupel wird eine Deduplizierung erforderlich, im Rahmen derer Datentupel identifiziert und zusammengeführt werden müssen, die dieselbe logische Einheit beschreiben. Und schließlich soll eine Datenfusion erreichen, dass Konflikte in einer Sammlung von Datenquellen aufgelöst werden, um die Richtigkeit der integrierten Daten sicherzustellen [DS13]. Die Komplexität dieser Aufgaben erhöht sich erheblich, wenn sie für Big Data auszuführen sind. Insbesondere aus der großen Anzahl an zu integrierenden Datenquellen, aus ihrer Heterogenität und ihrer unterschiedlichen Qualität ergeben sich Herausforderungen [DS13]. Daraus folgt für das Metamodell, dass es fähig sein muss, die Unterschiede zwischen verschiedenen externen Datenquellen zu abstrahieren, so dass Daten-Pipelines unabhängig von den Eigenschaften der externen Datenquellen definiert werden können. Für einzelne Verarbeitungsschritte ist im Metamodell zu beachten, dass diese möglicherweise Daten aus mehreren Tabellen als Eingabe beziehen müssen, um eine integrierte Ausgabe in einer anderen Tabelle bereitzustellen. Dies entspricht einer n:1-Abbildung zwischen n Eingabetabellen und einer Ausgabetabelle eines Verarbeitungsschritts.

A6: Speicherung von Zwischenergebnissen

Daten-Pipelines bestehen in der Regel aus mehreren Verarbeitungsschritten, die jeweils Zwischenergebnisse erzeugen. Es ist sinnvoll, diese zu speichern, da die daraus entstehenden Vorteile den zusätzlichen Aufwand und die Kosten für den erforderlichen Speicherplatz überwiegen [GGH+20]. Ein solcher Vorteil ist die Möglichkeit, die Zwischenergebnisse über mehrere Daten-Pipelines hinweg wiederzuverwenden. Dadurch kann eine zeitaufwendige und ineffiziente Neuberechnung gleicher Ausgaben eingespart werden. Auch im Falle eines Fehlschlagens der Daten-Pipeline sind Zwischenergebnisse relevant, da sie ihre Zuverlässigkeit und Wartbarkeit begünstigen. Tritt während eines Verarbeitungsschritts ein Fehler auf, kann dieser behandelt werden, indem der Verarbeitungsschritt unter Verwendung der gespeicherten Ergebnisse des vorgelagerten Verarbeitungsschritts wiederholt wird. Die Daten-Pipeline kann ihre Verarbeitung an der Stelle fortsetzen, an der sie unterbrochen wurde, ohne bereits abgeschlossene Verarbeitungsschritte erneut ausführen zu müssen. Um eine Speicherung von Zwischenergebnissen einzubeziehen, muss das Metamodell in der Lage sein, die von den Verarbeitungsschritten erzeugten Ausgaben zu erfassen und sie zur Nutzung durch nachgelagerte Verarbeitungsschritte bereitzustellen. Da die Daten eines Lakehouses in Tabellen gespeichert werden, sollte das Metamodell eindeutige Bezüge zwischen den Ein- und Ausgaben der einzelnen Verarbeitungsschritte und den entsprechenden Tabellen herstellen.

A7: Unterstützung komplexer Filter- und Bereinigungsoperationen

Die Qualität von Daten steht in direktem Zusammenhang mit der Genauigkeit der Ergebnisse, die aus ihnen gewonnen werden können. Sind die Daten externer Datenquellen von schlechter Qualität und werden nicht angemessen vorverarbeitet, besteht für Unternehmen das Risiko, falsche Schlussfolgerungen aus den Ergebnissen zu ziehen und bei wichtigen Geschäftsentscheidungen fehlgeleitet zu werden. Daher sollten sie der Fehlerbereinigung in ihren Daten-Pipelines eine hohe Priorität einräumen [LTS+18; TL03]. An das Metamodell richtet sich demnach die Anforderung, dass es auch komplexe Filter- und Bereinigungsoperationen unterstützen sollte. Die von Daten-Pipelines aufgenommenen Daten enthalten häufig verschiedene Anomalien wie fehlende oder doppelte Werte, falsche Datentypen und Inkonsistenzen, was komplexe Transformationen zur Datenbereinigung erforderlich machen kann [PVA19]. Diese sollte das Metamodell unterstützen, indem es benutzerdefinierten Code für die einzelnen Verarbeitungsschritte zulässt. Über den Code sollte es möglich sein, individuelle Verarbeitungslogik zu spezifizieren, die nicht auf vorgegebene Modelle oder Vorlagen beschränkt ist, sondern sich auf eine Datenbereinigungsmaßnahme zuschneiden lässt. Dazu sollte auch die Unterstützung von Machine Learning Modellen gehören, um Daten zu klassifizieren und bei Bedarf zu filtern.

A8: Unterstützung selbstanpassender Daten-Pipelines

In Kapitel 3 wird aufgezeigt, dass Daten-Pipelines in anderen Werkzeugen, wie Pipeline61 und Delta Live Tables, als DAG dargestellt werden. Diese Abstraktion erleichtert es, Abhängigkeiten zwischen Verarbeitungsschritten zu identifizieren und die Reihenfolge abzuleiten, in der Transformationen auf den Daten durchgeführt werden sollen. Insbesondere gewährleistet ein DAG, dass es zu keinen zyklischen Abhängigkeiten kommt. Auch wenn ein DAG für die Modellierung einer Vielzahl von Anwendungsfällen ausreichend ist, kann die Berücksichtigung von Zyklen in bestimmten Szenarien nützlich oder sogar erforderlich sein. Enthält eine Daten-Pipeline einen Zyklus bedeutet dies, dass die Ausgabe eines Verarbeitungsschrittes als Eingabe für einen in der Topologie vorher stattfindenden Verarbeitungsschritt verwendet wird und umgekehrt. Über die Einbettung solcher Datenschleifen können selbstanpassende Daten-Pipelines realisiert werden. Diese sind in der Lage, ihr Verhalten als Reaktion auf Veränderungen in ihrer Umgebung oder bei den eingespeisten Rohdaten automatisch zu ändern und dadurch den aktuell gegebenen Bedingungen besser gerecht zu werden. Dazu kann ein Verarbeitungsschritt Konfigurationen an vorgelagerte Verarbeitungsschritte zurückführen, um eine Anpassung der Verarbeitungsparameter vorzunehmen oder die Auswahl besser geeigneter Algorithmen zu steuern. In Daten-Pipelines für Machine Learning kann beispielsweise die Konfiguration für das regelmäßige Training eines Modells zurückgeführt werden. Aus jeder Trainingsiteration soll ein Modell hervorgehen, dass aufgrund der Einbeziehung der zurückgeführten Konfiguration besser dazu in der Lage ist, die vorgesehene Aufgabe zu bewältigen [GSSM18]. Um sicherzustellen, dass selbstanpassende Daten-Pipelines nicht ausgeschlossen werden, soll die im Metamodell verwendete Abstraktion von Daten-Pipelines nicht auf DAGs beschränkt sein. Stattdessen sollte das Metamodell auch Verarbeitungsschritte zwischen Tabellen gestatten, die zur Bildung von Zyklen führen.

A9: Bezug zu einem Zonenmodell

Zonenmodelle dienen der Verwaltung und logischen Organisation von Daten innerhalb eines Data Lakes. Sie definieren, in welchen Verarbeitungsgraden die Daten vorliegen und wie sie hinsichtlich Zugriffsrechten und Zuständigkeiten verwaltet werden [GGH+20]. Wie in Kapitel 2 beschrieben, wird die Delta-Architektur häufig nicht nur als eine Lösung angeführt, über die eine Vereinheitlichung von Batch- und Stream-Verarbeitung erreicht werden kann. Das Unternehmen Databricks empfiehlt im Rahmen der Delta-Architektur die drei Schichten *Bronze*, *Silver* und *Gold*, um Daten verschiedener Verarbeitungsgrade innerhalb eines Lakehouses logisch abzugrenzen. Dabei erfolgt die Datenverarbeitung zwischen Tabellen verschiedener Schichten über Daten-Pipelines, welche die Struktur und Qualität der eingespeisten Rohdaten während ihrer Bewegung durch die Schichten schrittweise verbessern sollen. Außerdem werden die Daten im Zuge dessen für bestimmte Anwendungsfälle vorbereitet. Beispielsweise kann eine Daten-Pipeline eine Voraggregation der Daten übernehmen, bevor sie einer Reporting-Anwendung bereitgestellt werden. Um die Konsistenz einer Daten-Pipeline mit den zugrundeliegenden Organisations- und Verwaltungsprinzipen des Lakehouse zu wahren, ist es erforderlich, dass das Metamodell den Bezug zu einem Zonenmodell herstellt. Das Metamodell sollte sicherstellen, dass jede Tabelle, die in einem Verarbeitungsschritt als Eingabe oder Ausgabe verwendet wird, einer Zone des Zonenmodells zugeordnet wird. Dabei ist eine Abhängigkeit des Metamodells von einem bestimmten Zonenmodell zu vermeiden. Stattdessen sollte das im Metamodell eingebettete Zonenmodell so definiert werden, dass es den Nutzern Flexibilität bei individuellen Anpassungen einräumt, um die Anwendbarkeit des Metamodells auf beliebige Lakehouse-Umsetzungen zu erhalten. Insbesondere soll es die Anzahl und Art der Zonen nicht vorschreiben, sondern deren grundlegende Eigenschaften beschreiben.

A10: Berücksichtigung von Data Governance

Data Governance befasst sich mit den Verantwortlichkeiten und der Ausübung von Kontrolle bei der Datenverwaltung. Es zielt darauf ab, eine unternehmensweite Datenstrategie umzusetzen und den Wert vorhandener Datenbestände zu maximieren. Gleichzeitig sollen datenbezogene Risiken möglichst klein gehalten werden [ASB19]. Als Beispiel für die Relevanz von Data Governance kann die Auswirkung der Datenschutz-Grundverordnung (DSGVO) angeführt werden. Unternehmen sind aufgrund dieser gesetzlichen Vorschrift dazu verpflichtet, mit personenbezogenen Daten entsprechend sensibel umzugehen und genau zu wissen, wo diese Daten gespeichert und wie diese Daten verwendet werden [ASB19; GGH+20]. Da Daten-Pipelines eine zentrale Rolle bei der Verarbeitung und Verwaltung von Unternehmensdaten einnehmen, ist für das Metamodell zu fordern, dass es sich auf die Vorgaben der Data-Governance-Strategie eines Unternehmens abstimmen lässt und darüber hinaus zu deren Durchsetzung beiträgt. Diese Anforderung schließt mit ein, dass das Metamodell zwischen Daten-Pipelines unterscheiden sollte, die auf sensible Daten zugreifen, und solchen, die sich ausschließlich mit nicht sensiblen Daten befassen. Außerdem sollte das Metamodell unterbinden, dass sensible Daten von Daten-Pipelines auf unzulässige Weise verarbeitet oder in nicht besonders geschützten Teilen des Lakehouses abgelegt werden. Letzteres darf das Metamodell nur dann zulassen, nachdem die Daten durch Verarbeitungsschritte einer Daten-Pipeline desensibilisiert wurden, zum Beispiel durch Anonymisierung [GGH+20].

5. Konzeption

Das Zentrum dieses Kapitels bildet der Vorschlag eines Metamodells für Daten-Pipelines in Lakehouses, das auf Basis der in Kapitel 4 gesammelten Anforderungen entwickelt wurde. Da es sich bei Lakehouses um neuartige Datenplattformen handelt, konnten im Rahmen der Literaturrecherche keine wissenschaftlichen Beiträge gefunden werden, auf die sich der Aufbau des vorgeschlagenen Metamodells stützen ließe. Deshalb greift diese Arbeit auf Modelle zurück, die ursprünglich für Data Lakes entworfen wurden. Abschnitt 5.1 erläutert, weshalb sich diese Data-Lake-Modelle auch auf Lakehouses übertragen lassen und auf welche Weise diese in das Metamodell für Daten-Pipelines in Lakehouses einfließen. Daraufhin wird in Abschnitt 5.2 das im Rahmen dieser Arbeit entworfene Metamodell eingeführt und aufgeschlüsselt, aus welchen Entitätstypen, Beziehungstypen und Attributen sich dieses zusammensetzt. In Abschnitt 5.3 wird vorgestellt, wie das Metamodell verfeinert werden kann. Das verfeinerte Metamodell sieht vor, dass in einer beliebigen Daten-Pipeline fünf verschiedene Typen von Verarbeitungsschritten vorkommen können. Zwei beispielhafte Daten-Pipelines, die sich über dieses verfeinerte Metamodell abbilden lassen, werden in Abschnitt 5.4 präsentiert. Abschließend stellt Abschnitt 5.5 Konzepte vor, die beschreiben, wie die beiden Metamodelle in die Entwicklung eines plattformunabhängigen Modellierungswerkzeugs für Daten-Pipelines in Lakehouses einbezogen werden können. Diese Konzepte sind im Kontext der Model Driven Architecture (MDA) zu sehen, einem modellgetriebenen Ansatz für Softwaredesign, der darauf abzielt, die Anwendungslogik von den darunterliegenden plattformspezifischen Technologien zu trennen. Insbesondere beleuchtet der Abschnitt die einzelnen Artefakte des an MDA angelehnten Ansatzes und skizziert, wie diese von einem Modellierungswerkzeug verwendet werden können.

5.1. Bezug zu vorhandenen Modellen für Data Lakes

Bereits in Abschnitt 2.2 wurde die Relevanz von Zonenmodellen für Data Lakes thematisiert. Während in der Literatur mehrere vielfältige Zonenmodelle für die Zonen in Data Lakes vorgeschlagen wurden, gibt es bisher keine Forschungsarbeiten, die sich detailliert mit der Verwaltung und Prozessierung von Daten in Lakehouses befassen. Die in Abschnitt 2.3.3 beschriebene, von Databricks vorgeschlagene Einteilung der Daten eines Lakehouses in die Schichten *Bronze*, *Silver* und *Gold* stellt zwar den ersten Schritt in Richtung eines Zonenmodells für Lakehouses dar, allerdings mangelt es an einer detaillierten Spezifizierung der Eigenschaften, Aufgaben und Verantwortlichkeiten jeder Schicht. Des Weiteren bestand das Ziel dieser Masterarbeit in der Entwicklung eines plattformunabhängigen Metamodells für Daten-Pipelines. Aus diesem Grund sollte kein Vorschlag eines einzelnen Unternehmens übernommen werden. In dieser Masterarbeit wird daher davon ausgegangen, dass sich an eine Zone in einem Lakehouse ähnliche Anforderungen wie an eine Zone in einem Data Lake stellen. Die Zuordnung eines Datensatzes zu einer Lakehouse-Zone soll signalisieren, dass dieser einen bestimmten Verarbeitungsgrad aufweist, durch eine bestimmte Datenqualität gekennzeichnet ist und bestimmten Sicherheitsmaßnahmen unterliegt. Ähnlich einer Zone in einem Data Lake wird

eine Lakehouse-Zone somit als ein logischer Bereich innerhalb eines Lakehouses aufgefasst, der einen bestimmten Zweck sowie eine Reihe von Data-Governance-Richtlinien erfüllt. Aufgrund dieser Analogie lässt sich das in Abschnitt 2.2.1 eingeführte Metamodell für Zonen in Data Lakes auch auf die Organisation von Daten eines Lakehouses anwenden. Um sicherzustellen, dass die Vorgaben einer Zone beim Lesen und Schreiben von Daten eines Lakehouses in jeder beliebigen Daten-Pipeline eingehalten werden, wird das von Giebler et al. [GGH+20] vorgeschlagene Metamodell für Zonen in das Metamodell für Daten-Pipelines integriert. Die Details dieser Einbettung werden in Abschnitt 5.2 behandelt.

Um auf ausführbare Daten-Pipeline-Instanzen zu schließen, bezieht sich diese Masterarbeit nicht unmittelbar auf das Metamodell für Daten-Pipelines. Stattdessen werden konkrete Daten-Pipeline-Modelle aus einem verfeinerten Metamodell für Daten-Pipelines abgeleitet. Diese Verfeinerung ähnelt der Methode zur Instanziierung des Metamodells für Zonen, die Giebler et al. zur Entwicklung des in Abschnitt 2.2.1 beschriebenen Zonenreferenzmodells für Data Lakes eingesetzt haben. Auf welche Weise diese Masterarbeit an das Zonenreferenzmodell anknüpft und wie die Verfeinerung des Metamodells im Detail erfolgte, geht aus Abschnitt 5.3 hervor. Hinsichtlich der Datenverarbeitung zwischen den Zonen des Zonenreferenzmodells treffen Giebler et al. keine spezifischen Aussagen. Die Autoren erwähnen lediglich, dass die Zonen in Bezug auf eine Batch-Verarbeitung der Daten verschiedene Verarbeitungsschritte kennzeichnen. Bei einer Stream-Verarbeitung hingegen definieren die Zonen eine Abfolge bestimmter Mengen von Verarbeitungsschritten, die auf einem durchlaufenden Datenstrom ausgeführt werden. Das verfeinerte Metamodell für Daten-Pipelines in Lakehouses übernimmt diese Perspektive, modelliert jedoch zusätzlich die zulässigen Übergänge zwischen den Zonen.

5.2. Metamodell für Daten-Pipelines in Lakehouses

Das Metamodell für Daten-Pipelines in Lakehouses ist mittels eines ER-Diagramms beschrieben, um die Struktur und Beziehungen zwischen den Bestandteilen einer solchen Daten-Pipeline offenzulegen. Durch das Weglassen sämtlicher Attribute entsteht eine vereinfachte Darstellung des Metamodells, wie in Abbildung 5.1 präsentiert. Eine Darstellung des vollständigen ER-Diagramms befindet sich im Anhang A.1. Neben dem Aufbau aus Entitäts- und Beziehungstypen sowie deren Verknüpfung untereinander verbergen sich auch hinter den vermerkten Kardinalitäten wichtige Informationen, um die Semantik des ER-Diagramms zu erschließen. Diese sind in der Min-Max-Notation hinterlegt und geben die minimale und maximale Anzahl der Instanzen eines Entitätstyps an, die an der zugeordneten Beziehungsinstanz beteiligt sein können. Darüber hinaus können die Entitäts- und Beziehungstypen grob in drei Bereiche untergliedert werden. Der blau dargestellte Teil des ER-Diagramms bezieht sich auf die Anbindung an externe Datenquellen und Datensenken, während der gelb dargestellte Teil die Ablage und Verwaltung der verarbeiteten Daten in einem Lakehouse fokussiert. Im unteren rot dargestellten Teil sind ER-Elemente angeordnet, die über die Datenverarbeitungsmöglichkeiten einer Daten-Pipeline Aufschluss geben. Diese Untergliederung dient als Leitfaden, um im Rahmen der folgenden Unterabschnitte alle Entitätstypen und deren Beziehungen zueinander vorzustellen.

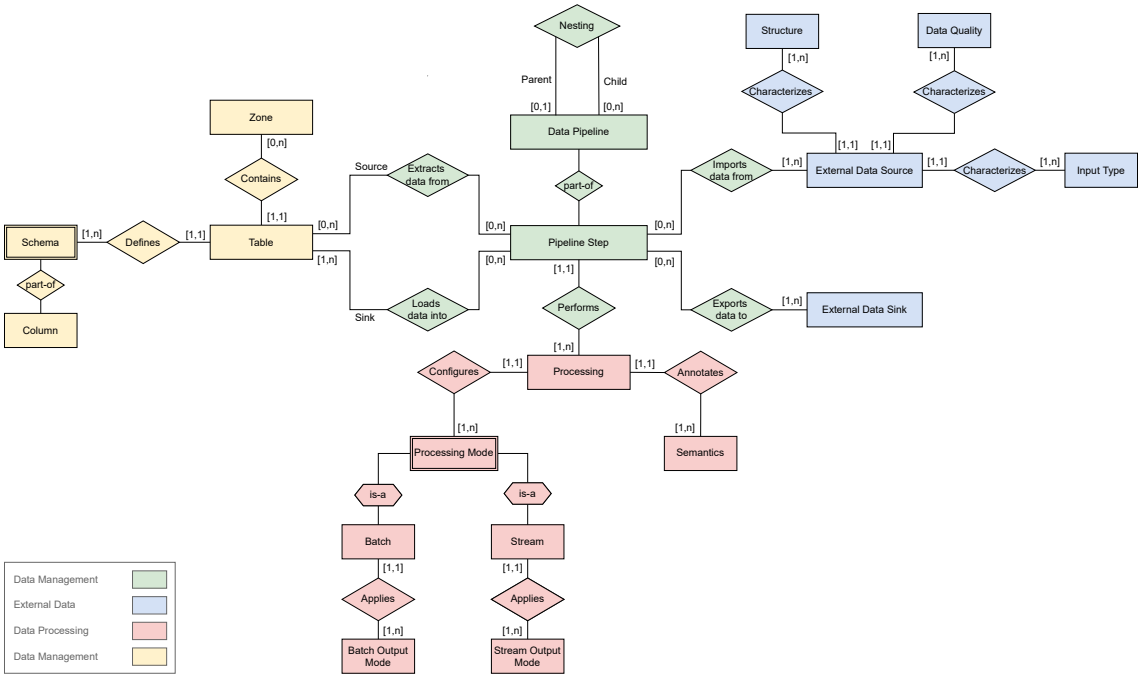

Abbildung 5.1.: Vereinfachte Darstellung des entwickelten Metamodells für Daten-Pipeline in Lakehouses als Entity-Relationship-Diagramm.

Zunächst werden die beiden grün dargestellten Entitätstypen DATA PIPELINE und PIPELINE STEP eingeführt. Anhand dieser lässt sich die grundlegende Struktur einer Daten-Pipeline nachvollziehen:

Data Pipeline: Eine geeignete Ausgangsbasis zur systematischen Vorstellung der Entitätstypen bietet der Entitätstyp DATA PIPELINE. Dieser abstrahiert von einer potentiell langen Kette einzelner Verarbeitungsschritte und ermöglicht somit eine übergeordnete Sicht auf die Datenverarbeitung innerhalb einer Daten-Pipeline. Eine detaillierte Beschreibung dieses Entitätstyps erschließt sich, indem man der *part-of*-Beziehung zum Entitätstyp PIPELINE STEP folgt. Dieser Beziehungstyp kennzeichnet, dass eine Daten-Pipeline eine Aggregation von einzelnen Teilen darstellt. Durch den Beziehungstyp NESTING wird dieser modulare Aufbau betont und es wird die Möglichkeit einer selbstreferenzierenden Beziehung zwischen Entitäten des Typs DATA PIPELINE geschaffen. Im Rahmen einer solchen Selbstreferenzierung zweier Daten-Pipelines nimmt eine der beiden Entitäten die übergeordnete Rolle als *Parent* ein, wobei die andere Entität ihr als *Child* untergeordnet ist.

Pipeline Step: Der Entitätstyp PIPELINE STEP ist im Metamodell an zentraler Position platziert, da er die meisten Beziehungen zu anderen Entitätstypen aufweist. Als Teil einer Daten-Pipeline bestimmt ein Pipeline-Schritt, wie und welche Daten in einem Abschnitt der Gesamtpipeline verarbeitet werden sollen. Auf dieser Abstraktionsebene lassen sich Pipeline-Schritte grundsätzlich in eine der folgenden Kategorien einordnen:

- Ein Pipeline-Schritt, der Daten in ein Lakehouse einspeist, importiert diese aus einer oder mehreren externen Datenquellen. Der Beziehungstyp IMPORTS DATA FROM stellt dabei die Verknüpfung zu diesen externen Entitäten her. Nach ihrer Verarbeitung lädt der Pipeline-Schritt die Daten in eine oder mehrere Tabellen.

- Ein Pipeline-Schritt, der keine Anbindung zu externen Entitäten aufweist, arbeitet mit mindestens einer Lakehouse-Tabelle. Um seiner Verarbeitungsaufgabe nachzukommen, kann er Daten einer oder mehrerer Tabellen extrahieren, was mithilfe des Beziehungstyps EXTRACTS DATA FROM abgebildet wird. Derartige Tabellen fungieren als Datenquelle des Pipeline-Schritts und sind als *Source-Tabellen* im ER-Diagramm annotiert. Die verarbeiteten Daten der Source-Tabellen repräsentieren das Ergebnis des Pipeline-Schritts, welches er daraufhin in einer oder mehreren Tabellen persistiert. Da diesen die Rolle einer Datensenke des betrachteten Pipeline-Schritts zukommt, werden diese im Folgenden als *Sink-Tabellen* bezeichnet. Es ist zu beachten, dass Source- und Sink-Tabellen eines Pipeline-Schritts nicht notwendigerweise unterschiedlich sein müssen.
- Ein Pipeline-Schritt am Ende einer Verarbeitungskette kann verarbeitete Daten einer oder mehrerer Tabellen an externe Datensenken weiterleiten. Die Anbindung an diese erfolgt über den Beziehungstyp EXPORTS DATA TO.

Allen Kategorien von Pipeline-Schritten ist gemein, dass ein einzelner Pipeline-Schritt eine n:m-Abbildung zwischen den Entitäten beschrieben kann, die den logischen Ablageort der Daten modellieren. Zu diesen Entitäten zählen externe Datenquellen und Datensenken sowie Source- und Sink-Tabellen. Im Gegensatz dazu ist jedem Pipeline-Schritt genau eine Entität vom Typ PROCESSING zugewiesen, welche festlegt, wie die eingehenden Daten verarbeitet werden sollen.

5.2.1. Anbindung an externe Entitäten

Das Metamodell soll den gesamten Ablauf der Datenverarbeitung im Kontext eines Lakehouses abdecken, welcher von der Datenaufnahme bis hin zur Datenanalyse oder -präsentation reicht. Um eine umfassende Beschreibung einer Daten-Pipeline bereitzustellen, müssen daher auch externe Entitäten in das Metamodell einbezogen werden. Unter den externen Entitäten sind die folgenden beiden Entitätstypen zu unterscheiden:

External Data Source: Ein Vorteil von Lakehouses besteht darin, dass sie in der Lage sind, große und heterogene Datenmengen aus verschiedenen externen Datenquellen an einem zentralen Ort zusammenzuführen, zu speichern und zu verwalten. Externe Datenquellen werden im Metamodell als Entitäten vom Typ EXTERNAL DATA SOURCE integriert. Die Daten dieser Entitäten können von einem oder mehreren Pipeline-Schritten in ein Lakehouse eingespeist werden. Um im Metamodell die Vielfalt und Komplexität der Datenquellen sowie die damit verbundenen Herausforderungen zu berücksichtigen, sind dem Entitätstyp eine Reihe anderer Entitätstypen und Attribute zugeordnet. Diese sind im Ausschnitt des Metamodells in Abbildung 5.2 dargestellt und werden im Folgenden erläutert:

- Aus dem Entitätstyp STRUCTURE leitet sich ab, wie die Daten einer externen Datenquelle organisiert sind. Die Verarbeitung strukturierter, semi-strukturierter und unstrukturierter Daten erfordert in der Regel unterschiedliche Verarbeitungsschritte. Dieser Umstand sollte insbesondere bei der Definition des Pipeline-Schritts berücksichtigt werden, welcher für die Einspeisung der externen Datenquelle zuständig ist.

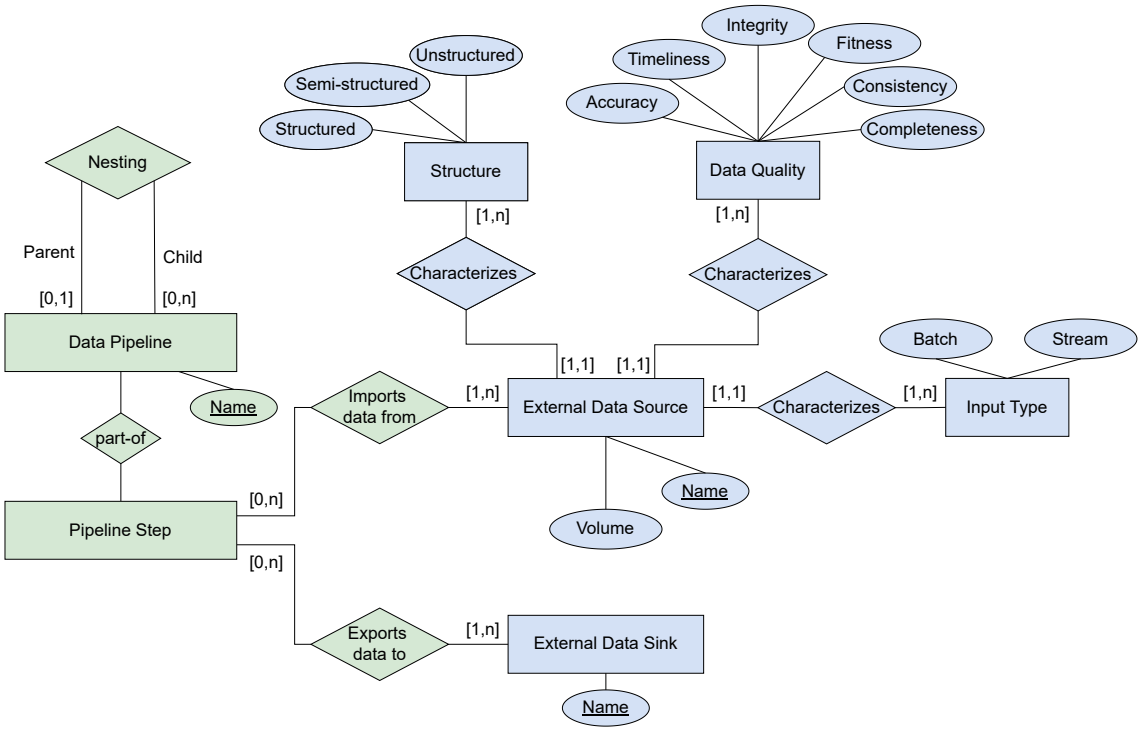

Abbildung 5.2.: Auf die Anbindung an externe Entitäten fokussierter Detailausschnitt des Metamodells als Entity-Relationship-Diagramm.

- Über den Entitätstyp DATA QUALITY einer externen Datenquelle werden unterschiedliche Aspekte der Datenqualität erfasst, wie beispielsweise die Genauigkeit, Konsistenz, Vollständigkeit und Zuverlässigkeit der bereitgestellten externen Daten. Die Datenqualität wird über die Attribute des Entitätstyps in die einzelnen Datenqualitätsmerkmale *Accuracy*, *Consistency*, *Completeness*, *Timeliness*, *Integrity* und *Fitness* aufgeschlüsselt. Diese Merkmale wurden von Cai und Zhu [CZ15] in einem hierarchischen Datenqualitätsrahmenwerk speziell für den Big-Data-Kontext vorgeschlagen und definiert. Für das Metamodell wurden die aufgeführten sechs Datenqualitätsmerkmale ausgewählt, da sie als besonders relevant für die nachgelagerte Verarbeitung durch eine Daten-Pipeline erachtet wurden. Bei Bedarf ist eine Erweiterung des Metamodells um weitere Datenqualitätsmerkmale denkbar.
- Der Entitätstyp INPUT TYPE hält fest, ob es sich bei der Datenquelle um eine Quelle für Batch- oder Streaming-Daten handelt. Die Kategorisierung der Datenquellen basiert auf der Häufigkeit, mit der externe Daten aktualisiert werden. Eine Batch-Datenquelle liefert neue Daten in größeren, meist periodischen Zeitintervallen, während eine Streaming-Datenquelle Daten kontinuierlich in Echtzeit bereitstellt.
- VOLUME bezieht sich auf die Größe der Daten, die von einer externen Datenquelle gespeichert oder erzeugt werden. Der Wert des Attributes kann dabei entweder für die Gesamtgröße der Daten stehen oder die Rate wiedergeben, mit der neue Daten hinzugefügt oder bestehende Daten aktualisiert werden.

External Data Sink: Das Gegenstück einer externen Datenquelle wird im Metamodell über den Entitätstyp External Data Sink repräsentiert. Dieser Entitätstyp bezieht sich auf Datensenken, die durch eine Daten-Pipeline mit Daten versorgt werden müssen. Anwendungen aus den Bereichen Advanced Analytics und OLAP/Reporting erfordern in der Regel direkten Datenzugriff auf die zugrundeliegende Speicherschicht [SGL+23]. Derartige Datensenken müssen allerdings nicht innerhalb einer Daten-Pipeline abgebildet werden, da die Möglichkeit des direkten Lesezugriff eines der Merkmale von Lakehouses darstellt. Eine Daten-Pipeline kann notwendig sein, wenn eine möglichst echtzeitnahe Belieferung der Daten gefordert ist. Daher werden Entitäten des Typs External Data Sink hauptsächlich mit Streaming-Anwendungen in Verbindung gebracht. Je nach Art und Struktur der Daten sowie dem beabsichtigten Anwendungszweck können sich verschiedene Merkmale für diese Datensenken ergeben. Deshalb ist der Entitätstyp im Metamodell berücksichtigt, jedoch sind seine Attribute aufgrund der großen Vielfalt von Merkmalen nicht modelliert.

5.2.2. Datenverarbeitung

Während sowohl Lakehouses als auch Data Lakes für die Speicherung großer Datenmengen konzipiert sind, verfügen Lakehouses über zusätzliche vorteilhafte Fähigkeiten. Insbesondere bei den Entitäten der Datenverarbeitung offenbart sich deshalb, dass das Metamodell auf Lakehouses als Datenplattform ausgelegt ist und sich nicht auf Daten-Pipelines in Data Lakes übertragen lässt. Diese Entitätstypen sind in Abbildung 5.3 im Detail dargestellt und werden nachfolgend erläutert:

Processing: Jeder Pipeline-Schritt innerhalb einer Daten-Pipeline ist über den Beziehungstyp Performs mit genau einer Entität des Typs Processing verbunden. Eine Entität des Typs Processing kann hingegen verschiedenen Pipeline-Schritten in einer oder mehreren Daten-Pipelines zugeordnet werden. Auf diese Weise wird sichergestellt, dass die Verarbeitungskonfiguration unabhängig von den Datenquellen und Datensenken eines Pipeline-Schritts bleibt. Somit kann die Konfiguration ausgetauscht und wiederverwendet werden, ohne dass Auswirkungen auf die Datenquellen und Datensenken eines Pipeline-Schritts berücksichtigt werden müssen. Die Verarbeitung der Eingabedaten wird von der Beziehung zu einer Entität des schwachen Entitätstyps Processing Mode bestimmt, sowie von zwei weiteren ER-Elementen, welche im Folgenden eingeführt werden:

- Das Attribut Code steht für die Möglichkeit, die Verarbeitungslogik eines Pipeline-Schrittes mittels benutzerdefiniertem Code zu spezifizieren. Dieser muss ausdrücken, wie aus den Eingabedaten der Source-Tabellen die Ausgabe für die Sink-Tabellen generiert wird. Bei gleicher Anzahl von Source- und Sink-Tabellen und ohne Angabe von benutzerdefiniertem Code kann angenommen werden, dass alle eingelesenen Daten in die Sink-Tabellen kopiert werden. Wenn jedoch eine unterschiedliche Anzahl von Source- und Sink-Tabellen involviert ist, muss zwingend benutzerdefinierter Code hinterlegt werden.
- Die Interpretation des Codes, welcher die Verarbeitungslogik einer Daten-Pipeline spezifiziert, stellt gemäß Grafberger et al. [GSS21] eine anspruchsvolle Aufgabe dar. Im Gegensatz zu SQL-Abfragen baut der Pipeline-Code häufig nicht auf einer algebraischen Abstraktion auf, wodurch ein schnelles Verständnis der Verarbeitungslogik erschwert wird. Um die Verarbeitung eines Pipeline-Schritts zu annotieren und leichter verständlich

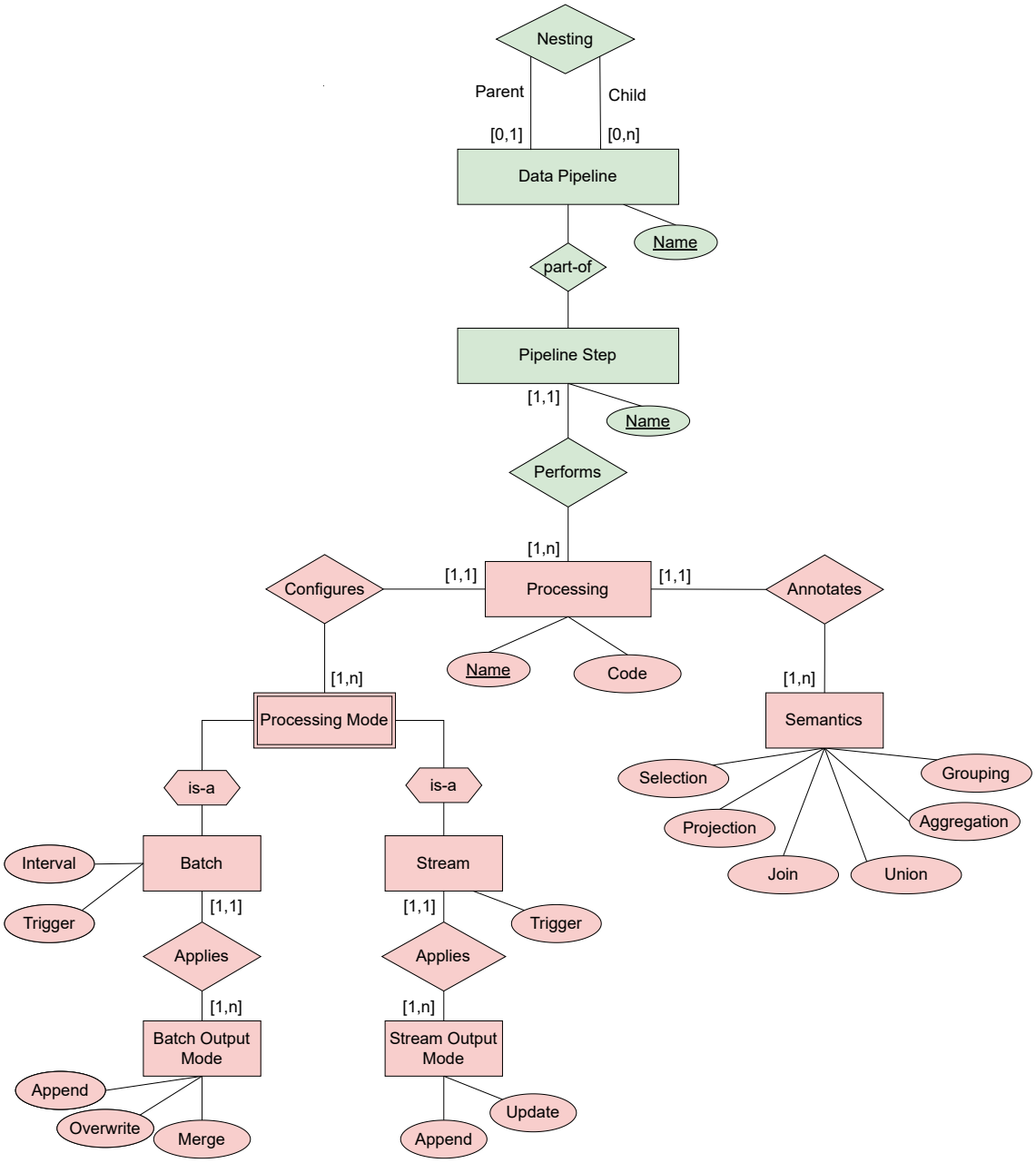

Abbildung 5.3.: Auf die Datenverarbeitung innerhalb einer Daten-Pipeline fokussierter Detailausschnitt des Metamodells als Entity-Relationship-Diagramm.

zu machen, stehen über die Attribute des Entitätstyps SEMANTICS gängige Operationen der relationalen Algebra zur Verfügung. Relationale Algebra bietet eine formale und präzise Beschreibungsmöglichkeit, um Operationen zur Manipulation von Relationen zusammenzufassen und zu kategorisieren. Bei der Auswahl der Attribute orientiert sich das Metamodell an *SPJUA*-Abfragen. Das Akronym *SPJUA* steht für fünf grundlegende Operationen in relationalen Datenbanken, nämlich Selektion, Projektion, Join, Union und Aggregation. Diese fünf Operationen bilden die Basis für komplexere Abfragen, da sie es ermöglichen, Daten aus verschiedenen Tabellen oder Teilen von Tabellen zu verknüpfen, auszuwählen, zu projizieren und zu kombinieren. Sie sind für die meisten Abfragen in relationalen Datenbanken ausreichend [HH10; Yao22]. Durch ihre Verwendung als Attribute des Entitätstyps SEMANTICS soll Nutzern, denen die Arbeit mit Tabellendaten vertraut ist, ein einfacherer Zugang zu dem möglicherweise komplexen benutzerdefinierten Code geboten werden. Insbesondere soll eine Analyse der Art und Reihenfolge von Operationen innerhalb einer Daten-Pipeline erleichtert werden, was als Grundlage für die Bewertung ihrer Korrektheit und Effizienz dienen kann. Es ist jedoch zu beachten, dass eine Entität vom Typ SEMANTICS lediglich annotierenden Charakter hat und keine Verarbeitungslogik spezifiziert. Dies wird durch den Beziehungstyp ANNOTATES betont, der Entitäten des Typs PROCESSING mit Entitäten des Typs SEMANTICS in Beziehung setzt.

Processing Mode: Einer der wesentlichen Vorteile eines Lakehouses im Vergleich zu anderen Datenplattformen besteht in ihrer Fähigkeit, eine vereinheitliche Batch- und Stream-Verarbeitung zu unterstützen. Diese Zusammenführung wird dadurch ermöglicht, dass dieselbe Tabelle sowohl als Quelle als auch als Senke für Pipeline-Schritte beider Verarbeitungsmodi genutzt werden kann. Das Metamodell verdeutlicht diesen Aspekt mittels des generalisierten Entitätstyps PROCESSING MODE. Mithilfe von zwei IS-A-Beziehungstypen wird er in die spezialisierten Entitätstypen BATCH und STREAM unterteilt. Von oben nach unten betrachtet drückt eine IS-A-Beziehung eine Spezialisierungsbeziehung aus. Diese besagt, dass eine Entität ein Spezialfall einer allgemeineren Entität ist und somit alle Attribute und Beziehungen der übergeordneten Entität erbt. Auf diese Weise kann einem Pipeline-Schritt entweder eine Batch- oder eine Stream-Verarbeitung zugeordnet werden. Da die Unterscheidung der Verarbeitungsmodi auf der feingranularen Ebene eines einzelnen Pipelines-Schrittes erfolgt, sind aus Sicht des Metamodells auch Daten-Pipelines zulässig, die beide Verarbeitungsmodi mischen.

Batch: Sofern ein Pipeline-Schritt die importierten oder extrahierten Daten als Batch verarbeitet, sollte der Verarbeitungsmodus als Entität vom Typ BATCH instanziiert werden. Im Gegensatz zu einer inkrementellen Echtzeit-Verarbeitung werden die Daten in einer Batch-Verarbeitung als diskrete Menge eingelesen und das Ergebnis wird erst nach der Verarbeitung des gesamten Batches in eine Sink-Tabelle geschrieben. Dieser Verarbeitungsmodus ist daher nur dann geeignet, wenn die Ergebnisse nicht echtzeitnah zur Verfügung stehen müssen und eine Verzögerung der Datenverarbeitung keine negativen Auswirkungen auf die beabsichtigte Verwendung hat. Um detaillierte Aussagen über das Verhalten einer Batch-Verarbeitung zu ermöglichen, besitzt eine Entität vom Typ BATCH zwei Attribute und ist über eine APPLIES-Beziehung mit einer Entität vom Typ BATCH OUTPUT MODE verbunden. Im Folgenden wird die Bedeutung dieser ER-Elemente erläutert:

- Im Zusammenhang mit einer Batch-Verarbeitung beschreibt der Entitätstyp Batch Output Mode, auf welche Weise die Verarbeitungsergebnisse in einer Sink-Tabelle persistiert werden. Dafür verfügt der Entitätstyp über die drei Attribute Append, Overwrite und Merge, die jeweils einen der drei möglichen Ausgabemodi repräsentieren. Die Auswahl dieser Ausgabemodi erfolgte auf Basis ihrer Unterstützung durch gängige Prozessierungsframeworks und Lakehouse-Frameworks. Bei Verwendung des Append-Modus werden die Ergebnisse der Batch-Verarbeitung an die Daten der Sink-Tabelle angehängt, ohne dabei bereits vorhandene Daten zu überschreiben. Hingegen bewirkt der Overwrite-Modus, dass das Verarbeitungsergebnis zum neuen Tabelleninhalt der Sink-Tabelle wird und alle vorhandenen Daten vollständig ersetzt. Der Merge-Modus wiederum wird dazu genutzt, die Ergebnisse mit den bestehenden Daten zu kombinieren. Beispielsweise unterstützen die beiden Lakehouse-Frameworks Delta Lake und Apache Iceberg im Rahmen einer solchen Zusammenführung Einfügungen und Aktualisierungen auf Zeilenebene, indem eine Gleichheitsbedingung ausgewertet wird, die Spalten einer Source- und einer Sink-Tabelle in Beziehung setzt.
- Das Attribut Interval gibt den zeitlichen Abstand an, in dem die Batch-Verarbeitung eines Pipeline-Schrittes durchgeführt wird. Ein passender Attributwert ist unter Berücksichtigung mehrerer Faktoren zu wählen, darunter die noch akzeptable Aktualität der Daten und die ineffiziente Ressourcennutzung bei zu häufiger Wiederholung. Falls eine Batch-Verarbeitung nicht in Intervallen, sondern zum Beispiel nur nach manueller Auslösung erfolgen soll, könnte dies durch den Attributwert -1 signalisiert werden.
- Durch das Attribut Trigger wird bestimmt, auf welche Weise eine Batch-Verarbeitung initiiert wird. Hierfür könnte ein zeitgesteuerter Trigger verwendet werden, der eine Batch-Verarbeitung gemäß des im Attribut Interval definierten Intervalls durchführt, beispielsweise in einem täglichen, wöchentlichen oder monatlichen Rhythmus. Es besteht auch die Möglichkeit, einen ereignisgesteuerten Trigger zu nutzen, der infolge eines bestimmten Ereignisses außerhalb des Lakehouses oder durch den Abschluss eines anderen Batch-Verarbeitungsschrittes in der Daten-Pipeline ausgelöst wird.

Stream: Alternativ zu einer Batch-Verarbeitung kann der Verarbeitungsmodus über den Entitätstyp Stream spezialisiert werden. Bei diesem Verarbeitungsmodus werden die Eingabedaten in Echtzeit verarbeitet, sobald sie in einer externen Datenquelle oder einer Source-Tabelle verfügbar werden. Im Vergleich zu einer Batch-Verarbeitung ist die Menge der Eingabedaten geringer, da nur die neuesten Daten verarbeitet werden und Berechnungen nicht auf allen Daten stattfinden, die zuvor über einen längeren Zeitraum gesammelt wurden. Durch die inkrementelle Verarbeitung von Streaming-Daten werden fortlaufend Verarbeitungsergebnisse erzeugt und in einer Tabelle gespeichert, während auf der Eingabeseite des Pipeline-Schritts kontinuierlich neue Daten zur Verarbeitung eintreffen können. Dieser Verarbeitungsmodus ist daher für Anwendungsfälle mit Echtzeit-Anforderungen zwingend erforderlich. Details zur Verarbeitungsweise einer Stream-Verarbeitung erschließen sich aus dem Entitätstyp Stream Output Mode und der Konfiguration des Attributs Trigger. Nachfolgend werden diese ER-Elemente vorgestellt:

- Das zeitliche Verhalten einer Stream-Verarbeitung kann über das Attribut Trigger gesteuert werden. Hinter einem Trigger stehen Bedingungen, aus denen sich ableiten lässt, wann und wie oft neue Streaming-Daten verarbeitet werden. Dabei kommen verschiedene Arten von Bedingungen in Frage, je nachdem, über welchen Ansatz der

Trigger umgesetzt wird. Denkbar ist zum Beispiel ein zeitbasierter Trigger, der alle 15 Sekunden eine neue Ausführung des Pipeline-Schritts auslöst. Hingegen bestimmt ein datenbasierter Trigger, dass eine neue Ausführung startet, sobald eine ausreichende Menge neuer Streaming-Daten angekommen ist.

- Der Umgang mit den kontinuierlich erzeugten Streaming-Ergebnissen wird analog zur Batch-Verarbeitung anhand des Entitätstyps Stream Output Mode festgelegt. Die Attribute dieses Entitätstyps, nämlich Append und Update, entsprechen dabei der Schnittmenge der Operationen, die gängige Prozessierungsframeworks wie Apache Spark und Apache Flink unterstützen. Damit soll vermieden werden, dass die Ausgabemodi an den Funktionsumfang bestimmter Technologien gebunden sind. Im Append-Modus werden neu erzeugte Ergebnisse seit dem letzten Trigger an vorhandene Tabellendaten angehängt, ohne diese zu überschreiben. Wenn die Daten einer Sink-Tabelle auf Basis neuerer Ergebnisse aktualisiert werden müssen, ist der Update-Modus zu wählen. Dabei werden nur die Zeilen in die Sink-Tabelle geschrieben, deren Werte sich seit dem letzten Trigger tatsächlich verändert haben und ersetzen dort bereits vorhandene Zeilen. Es ist jedoch darauf zu achten, dass der Update-Modus je nach Lakehouse-Implementierung und verwendetem Prozessierungsframework unterschiedliche Anforderungen und Auswirkungen haben kann. Insbesondere innerhalb des Prozessierungsframeworks Apache Spark bezieht sich der Update-Modus ausschließlich auf Aggregationen. Dabei werden die aggregierten Werte einer Tabelle für die in einer `GROUP-BY`-Klausel aufgeführten Gruppen aktualisiert. Wenn ein Lakehouse durch das Lakehouse-Framework Apache Iceberg umgesetzt ist, das Primärschlüsseleinschränkungen für Spalten unterstützt, und das Prozessierungsframework Apache Flink zum Einsatz kommt, können Aktualisierungen auch jenseits von Aggregationen vorgenommen werden. Hierbei erfolgt die Aktualisierung anhand der eindeutigen Identifikation der betroffenen Zeilen.

5.2.3. Datenverwaltung

Neben der Darstellung der verfügbaren Verarbeitungsmöglichkeiten innerhalb einer Daten-Pipeline zeigt auch die Strukturierung der Datenablage, dass Lakehouses als Datenplattformen im Mittelpunkt des Metamodells stehen. Wie im Detailausschnitt des Metamodells in Abbildung 5.4 visualisiert, umfasst das Metamodell mehrere Entitätstypen, die an der Datenverwaltung beteiligt sind. Diese werden im Folgenden näher vorgestellt:

Zone: Der vorausgegangene Abschnitt 5.1 begründet, wieso sich das Metamodell für Data-Lake-Zonen in das Metamodell für Daten-Pipelines in Lakehouses einbetten lässt. Diese Einbettung entspricht der Übernahme des Entitätstyps Zone in das Metamodell für Daten-Pipelines. Dadurch soll sichergestellt werden, dass die Datenverwaltung innerhalb eines Lakehouses im Einklang mit aktuellen Forschungserkenntnissen eines verwandten und übertragbaren Bereichs steht, wie dem der Zonenmodelle für Data Lakes. Es ergibt sich jedoch eine Änderung im Metamodell für Data-Lakes-Zonen, die auf die Tatsache zurückzuführen ist, dass die Datenspeicherung nicht in einem Data Lake, sondern in den Tabellen eines Lakehouses stattfindet. Schema entfällt als Aspekt des Attributs Data Characteristics, da das Schema im Metamodell für Daten-Pipelines als separater Entitätstyp berücksichtigt wird. Diese Verlagerung unterstreicht die Bedeutung von Tabellen im Lakehouse-Kontext.

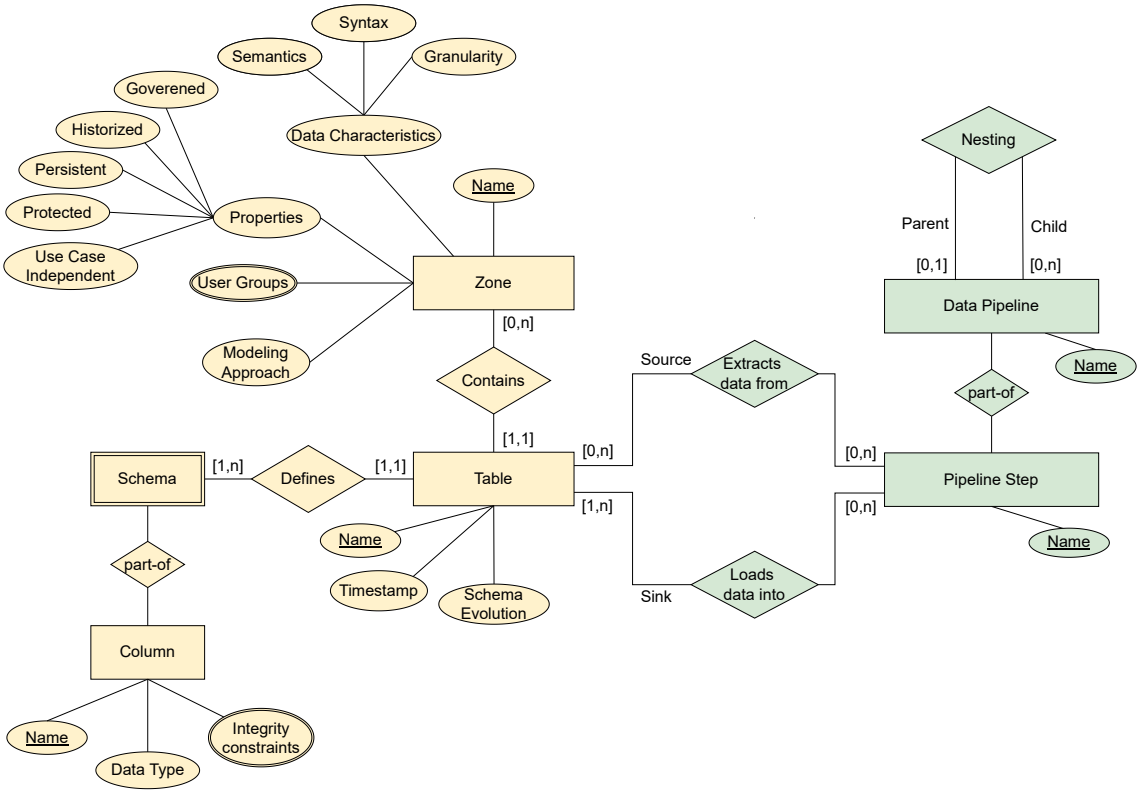

Abbildung 5.4.: Auf die Datenverwaltung innerhalb eines Lakehouses fokussierter Detailausschnitt des Metamodells als Entity-Relationship-Diagramm.

Um mit Tabellendaten zu arbeiten, muss eine Daten-Pipeline das Schema der Tabelle kennen, weshalb die Modellierung des Schemas als Aspekt des Attributs DATA CHARACTERISTICS unzureichend ist.

Table: Die Daten eines Lakehouses sind auf logischer Ebene in Tabellen organisiert. Deshalb ist ein Pipeline-Schritt an mindestens eine Entität vom Typ TABLE angebunden, um aus dieser Daten zu extrahieren oder bereits verarbeitete Daten in diese zu laden. Die Betrachtung eines Lakehouses auf Tabellenebene entspricht einer feineren Auflösung im Vergleich zu der Strukturierung, die durch Lakehouse-Zonen durchgesetzt wird. Eine einzelne Zone kann mehrere Tabellen umfassen. Dabei ist jede Tabelle genau einer Zone zugehörig, was über die CONTAINS-Beziehung zum Ausdruck gebracht wird. Neben einem Schema, das über einen separaten Entitätstyp abgebildet ist, besitzt eine Lakehouse-Tabelle mehrere Attribute:

- Das Attribut SCHEMA EVOLUTION bestimmt, ob Daten in eine Tabelle geschrieben werden dürfen, obwohl sie nicht konform zum vordefinierten Schemas der Tabelle sind. Das Konzept der *Schema-Evolution* erlaubt eine zeitliche Weiterentwicklung und Änderung des vordefinierten Schemas. Dies kann beispielsweise von Vorteil sein, wenn sich die Namen, Datentypen oder die Anzahl von Spalten einer externen Datenquelle häufig ändern. Dennoch sollte bei der Verwendung dieser Option sorgfältig abgewogen werden, welche Auswirkungen eine Schemaentwicklung auf nachgelagerte Verarbeitungsschritte hat. Um hierbei eine höhere Sicherheit zu gewährleisten, empfiehlt sich das Konzept des *Schema-Enforcement*. Dieses sieht vor, dass ein vordefiniertes

Tabellenschema erzwungen wird, indem Daten, die nicht dem Schema entsprechen, abgelehnt werden. Dadurch wird die nachgelagerte Verarbeitung und Analyse der Daten erleichtert. Da sich die Konzepte *Schema Evolution* und *Schema-Enforcement* in Bezug auf eine Tabelle gegenseitig ausschließen, kann aus der Belegung des Attributs *Schema-Evolution* abgeleitet werden, welches der beiden Konzepte in einer bestimmten Tabelle angewendet wird.

- Um die zeitliche Version einer Tabelle zu identifizieren, ist TIMESTAMP als Attribut einer Tabelle in das Metamodell aufgenommen. Durch Erfassung und Speicherung jeder Änderung eines Daten-Tupels in einem Lakehouse ist es möglich, mithilfe der Funktionalität *Time Travel* auf vorherige Versionen einer Tabelle zuzugreifen. Für einen Pipeline-Schritt einer Daten-Pipeline kann durch Angabe des Zeitstempels spezifiziert werden, welche Version einer Source-Tabelle verwendet werden soll.

Schema: Jede Tabelle in einem Lakehouse besitzt ein Schema, unabhängig von der Art und Struktur der darin enthaltenen Daten. Aus der Perspektive des Metamodells kann ein Schema ohne Tabelle, der dieses Schema zugrunde liegt, nicht existieren. Aus diesem Grund ist das Schema als schwacher Entitätstyp modelliert, dessen Entitäten sich ausschließlich durch ihre Beziehung zu einer Entität des Typs TABLE eindeutig identifizieren lassen. Außerdem sind einer Entität vom Typ SCHEMA keine Attribute zugeordnet. Stattdessen baut sich deren Semantik durch eine Reihe von Spalten auf, die Bestandteile eines Schemas darstellen. Dieses Verhältnis wird durch eine PART-OF-Beziehung zwischen einem Schema und den Entitäten des Typs COLUMN ausgedrückt.

Column: Die Spalten eines Schemas legen die vertikale Struktur einer Tabelle fest und können über den Entitätstyp COLUMN angegeben werden. Jede Spalte ist über die folgenden Attribute näher beschrieben:

- Für den Datentyp einer Spalte ist das Attribut DATA TYPE vorgesehen. Wie in jeder relationalen Datenbank besitzt auch eine Spalte innerhalb einer Lakehouse-Tabelle einen definierten Datentyp, der genau spezifiziert, welche Art von Daten in der Spalte gehalten werden können.
- INTEGRITY CONSTRAINTS ist einer Tabelle als mehrwertiges Attribut hinterlegt, was bedeutet, dass es in der Lage ist, mehrere Werte für eine einzelne Entität zu erfassen. Ähnlich zu einer relationalen Datenbank sollen Integritätseinschränkungen dazu beitragen, die Konsistenz, Genauigkeit und Qualität der gespeicherten Daten aufrechtzuerhalten. Als Integritätseinschränkungen können verschiedene Vorschriften eingeordnet werden, darunter Primär- und Fremdschlüssel-Beschränkungen oder Check Constraints, die sicherstellen, dass Datenwerte einer Spalte bestimmte Bedingungen erfüllen. Die verfügbaren Lakehouse-Frameworks unterscheiden sich in Bezug auf die Unterstützung von Integritätseinschränkungen. Zum Beispiel bietet Delta Lake derzeit lediglich informative Primär- und Fremdschlüssel-Beschränkungen an, die jedoch auf den Daten nicht durchgesetzt werden. Hingegen wird in einer Hudi-Tabelle jeder Datensatz eindeutig durch einen Primärschlüssel identifiziert, der sich aus dem Schlüssel des Datentupels und dem Namen der Partition zusammensetzt.

5.3. Verfeinertes Metamodell für Daten-Pipelines in Lakehouses

Durch die Verfeinerung des im vorherigen Abschnitt beschriebenen Metamodells ergibt sich das verfeinerte Metamodell für Daten-Pipelines in Lakehouses, welches in Abbildung 5.5 dargestellt ist. Das verfeinerte Metamodell basiert auf dem ER-Diagramm des ursprünglichen Metamodells und zeigt auf, wie ein Teil der Entitätstypen des Metamodells in verschiedene Typen untergliedert werden können. Um die Lesbarkeit zu verbessern, wurden in der Abbildung einige Entitätstypen und Beziehungstypen sowie sämtliche Attribute und Kardinalitätsbeschränkungen weggelassen. Diese Elemente bleiben im verfeinerten Metamodell unverändert bestehen. Aus den Instanzen des Entitätstyps Zone, die Giebler et al. [GGH+20] in ihrem Zonenreferenzmodell identifizieren, erschließt sich die Existenz der abgebildeten Zonen-Typen. Darauf aufbauend können unter Berücksichtigung der möglichen Zonenzugehörigkeiten einer Tabelle unterschiedliche Unterteilungen des Entitätstyps Table eingeführt werden. Hieraus ergeben sich die Tabellen-Typen Raw Table, Harmonized Table, Distilled Table, Delivery Table und Explorative Table. Hinsichtlich eines Pipeline-Schritts wird angenommen, dass dieser zwischen zwei Tabellen agiert, sofern er sich nicht mit der Dateneinspeisung aus einer externen Datenquelle befasst. Daraus folgt, dass zwischen den Instanzen des Entitätstyps Table verschiedene Instanzen vom Typ Pipeline Step anzusiedeln sind. Deren Benennung als Loading, Harmonization, Distillation, Delivering und Exploration ist den Namen der Zonen entlehnt, in denen die Zwischenergebnisse eines Pipeline-Schritt-Typs persistiert werden. Die gewählten Namen sollen einen Hinweis darauf geben, welche Art von Verarbeitungsaufgaben bei der Überführung von Daten zwischen zwei Zonen oder innerhalb einer Zone anfallen können. Jedoch wird die Verarbeitungsaufgabe, die ein bestimmter Pipeline-Schritt-Typ übernehmen kann, explizit nicht eingeschränkt. Vielmehr zielen die verschiedenen Pipeline-Schritt-Typen darauf ab, eine logische Strukturierung der Verarbeitungsaufgaben einzuführen und Vergleiche zwischen mehreren Daten-Pipelines zu erleichtern. Die verschiedenen Pipeline-Schritt-Typen und ihre jeweiligen Rollen werden im Folgenden vorgestellt:

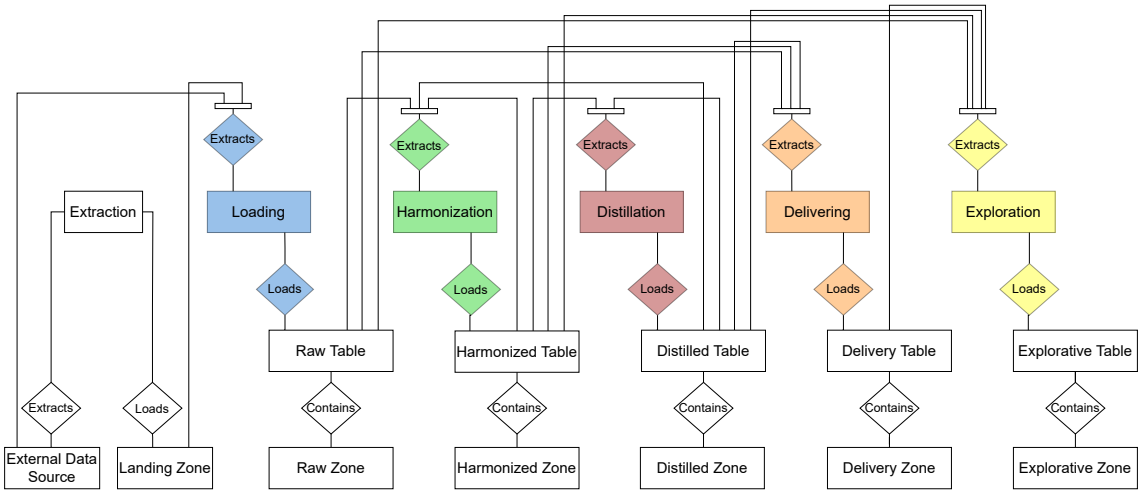

Abbildung 5.5.: Verfeinertes Metamodell für Daten-Pipelines in Lakehouses, das einer Erweiterung des Zonenreferenzmodells für Data Lakes [GGH+20] entspricht. Die Darstellung ist an ein Entity-Relationship-Diagramm angelehnt.

Loading: Der LOADING-Schritt extrahiert Daten ein oder mehrerer externer Datenquellen und speist sie in die RAW-Zone des Lakehouses ein. Er wendet lediglich einfache Transformationen an, weshalb die Daten der RAW-Zone in nahezu unverarbeitetem Zustand vorliegen.

Harmonization: In der HARMONIZED-Zone werden Daten in bereinigter Form abgelegt. Folglich stehen hinter dem HARMONIZATION-Schritt einer Daten-Pipeline Aufgaben, die zur Bereinigung und bedarfsorientierten Aufbereitung der Daten beitragen. Insbesondere übernimmt ein HARMONIZATION-Schritt die Zusammenführung von Daten verschiedener externer Datenquellen, was ihrer Integration in ein gemeinsames Schema entspricht. Giebler et al. [GGH+20] weisen darauf hin, dass von einem einzelnen, globalen Schema abzusehen ist. Aus der Integration sollten mehrere partielle Schemata hervorgehen, die unterschiedliche Datenquellen und Kontexte berücksichtigen. Um Kompatibilität zwischen verschiedenen Datenquellen herzustellen und die Integrationsaufgabe zu bewältigen, können sich im Rahmen einer Harmonization eine Reihe von Teilschritte ergeben. Dazu zählt beispielsweise die Konvertierung von Daten, bei der diese von einem Datentyp in einen anderen überführt werden, und die Korrektur von fehlerhaften oder inkonsistenten Daten.

Distillation: Gemäß dem Zonenreferenzmodell für Data Lakes ist die DISTILLATION-Zone die erste von drei anwendungsabhängigen Zonen. Ein DISTILLATION-Schritt fokussiert infolgedessen die Aufbereitung der Daten, um die Effizienz nachfolgender Analysen zu erhöhen. Zu seinen Aufgaben kann unter anderem die Aggregation von Daten, die Anreicherung eines Datensatzes durch einen anderen oder die Durchführung komplexer Analysen aus dem Bereich des Machine Learnings zählen.

Delivering: Ein DELIVERING-Schritt befasst sich mit der Bereitstellung von Daten in der DELIVERY-Zone. Er ist darauf ausgerichtet, die Datengrundlage für bestimmte Anwendungsfälle verfügbar zu machen. Die Logik hinter einem DELIVERING-Schritt ist begrenzt, da komplexe Verarbeitungen von vorgelagerten HARMONIZATION- und DISTILLATION-Schritten übernommen werden. Was je nach Nutzung anfallen kann, sind einfache Transformationen, um die Daten in eine bestimmte Darstellungsform zu überführen und sich nicht auf deren Semantik auswirken. Der Konsum der Daten aus der DELIVERY-Zone durch externe Datensenken wird nicht mehr als Teil einer Daten-Pipeline betrachtet, da sich diese auf die Datenverarbeitung innerhalb eines Lakehouses konzentriert.

Exploration: Der EXPLORATION-Schritt einer Daten-Pipeline kann dafür eingesetzt werden, um Daten aus anderen Zonen in die EXPLORATIVE-Zone zu überführen, damit Data Scientists diese flexibel nutzen können. Hingegen ist die Verarbeitung und Analyse innerhalb dieser Zone und die Rückführung der erzielten Resultate in die DISTILLATION-Zone nicht Teil des Aufgabenbereichs eines EXPLORATION-Schritts. Denn Data Scientists arbeiten manuell auf den Daten, um Verarbeitungs- oder Analyseergebnisse abzuleiten, die für ihre Zwecke nützlich sind. Eine automatisierte Daten-Pipeline kann lediglich die Zulieferung der Ausgangsdaten einer Analyse übernehmen.

Im Gegensatz zum ursprünglichen Metamodell setzt das verfeinerte Modell definierte Zonen für das Lakehouse fest und erlegt Daten-Pipelines auf diese Weise eine Reihe von Einschränkungen auf. Insbesondere müssen die Source- und Sink-Tabellen eines Pipeline-Schritts in bestimmten Zonen liegen, damit ein Pipeline-Schritt einer Daten-Pipeline Daten zwischen diesen transferieren darf. Abbildung 5.6 veranschaulicht diese Einschränkungen des verfeinerten Metamodells, indem sie einen Überblick über die unterstützten Pipeline-Schritt-Typen bietet. In der Abbildung werden die

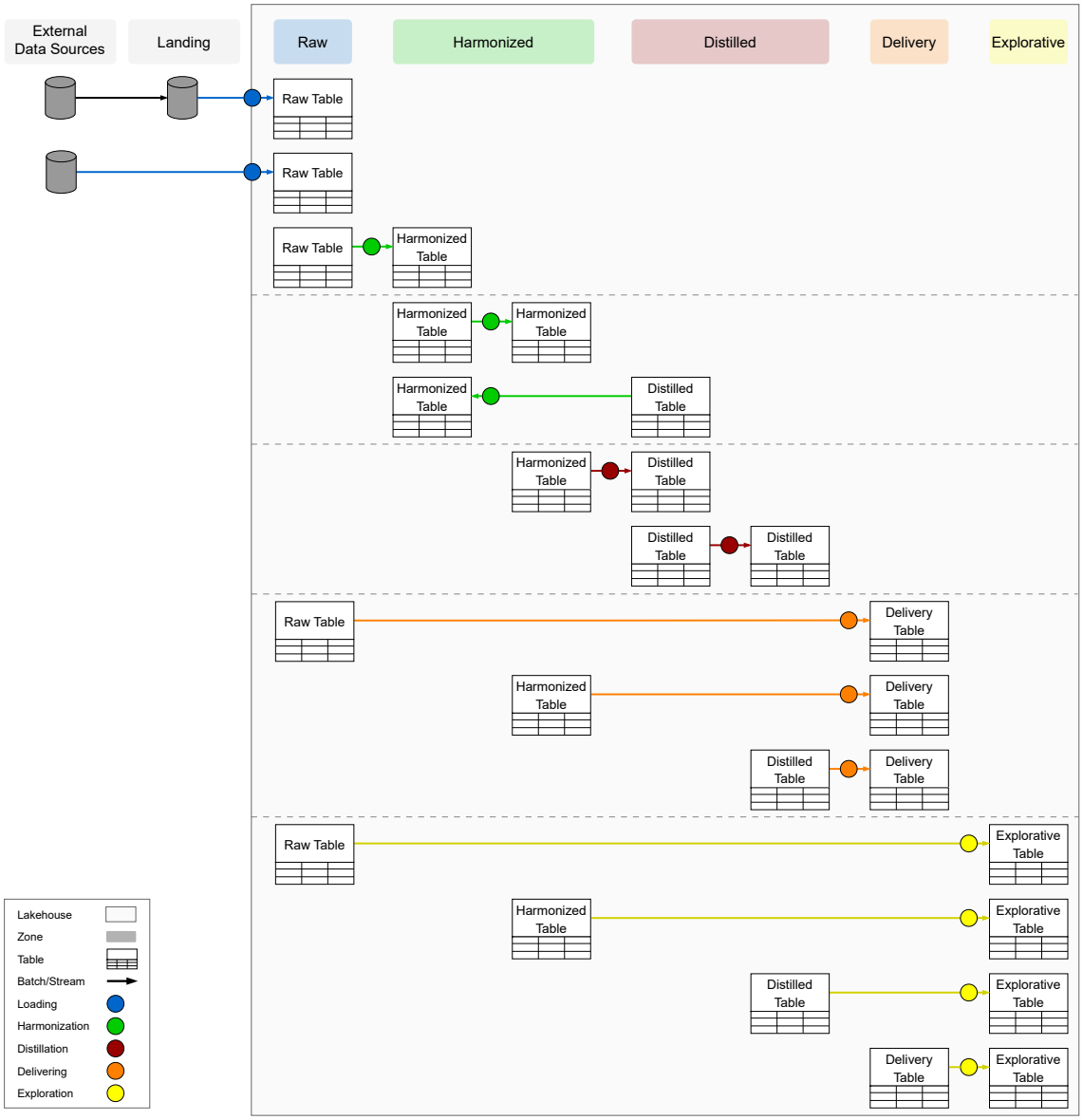

Abbildung 5.6.: Übersicht über die verschiedenen Pipeline-Schritt-Typen, die das verfeinerte Metamodell für Daten-Pipelines in Lakehouses zulässt. Jeder Typ stellt Anforderungen an Source- und Sink-Tabellen in Bezug auf ihre Zonenzugehörigkeit.

zulässigen Pipeline-Schritte einzeln aufgeführt und nicht zu einer abgeschlossenen Daten-Pipeline zusammengefasst. Alle dargestellten Pipeline-Schritte beschreiben eine 1:1-Abbildung zwischen einer externen Datenquelle und einer Tabelle oder zwischen zwei Tabellen. Diese Darstellung ist allerdings vereinfacht, da das verfeinerte Metamodell ebenso Abbildungen zwischen Tabellen mit einem Verhältnis von 1:n, n:1 und n:m unterstützt, wie es auch beim ursprünglichen Metamodell der Fall ist. Außerdem verdeutlicht die Abbildung, dass die Landing-Zone nicht als Zone innerhalb eines Lakehouses aufgefasst wird, was eine Abweichung vom Zonenreferenzmodell für Data Lakes darstellt. Die Lakehouse-Vision setzt eine integrierte Datenplattform voraus, wobei vorgelagerte Technologien, wie Streaming-Plattformen zur Aufnahme von schnell erzeugten Streaming-Daten, nicht als logischer Bestandteil des Lakehouses betrachtet werden. Deshalb muss die Aufnahme von Daten in die Landing-Zone durch separate Prozesse erfolgen. Der Wirkungsbereich einer Daten-Pipeline beschränkt sich auf die Dateneinspeisung und Datenverarbeitung in einem Lakehouse.

5.4. Beispiele für Daten-Pipeline-Modelle

Die Erstellung von Daten-Pipeline-Modellen erfolgt durch Instanziierung des im vorherigen Abschnitt vorgestellten, verfeinerten Metamodells für Daten-Pipelines. Dabei werden Pipeline-Schritte einer ausführbaren Daten-Pipeline spezifiziert. In den folgenden Unterabschnitten werden zwei beispielhafte Daten-Pipeline-Modelle vorgestellt, die beide für die Verarbeitung von Daten aus dem Internet of Things (IoT)-Bereich konzipiert sind. Als externe Streaming-Datenquelle werden mehrere IoT-Sensorplattformen angenommen, die eine Reihe von Umgebungsmesswerten an ihrem jeweiligen Installationsorten erfassen können, wie beispielsweise Temperatur, Luftfeuchtigkeit oder Lautstärke. Dabei integriert eine einzelne IoT-Sensorplattformen mehrere Sensoren, von denen jeder auf die Erfassung einer Umgebungsgröße ausgelegt ist. In Intervallen von wenigen Sekunden sendet eine IoT-Sensorplattform die neuesten Messwerte als einzelne Nachrichten zusammen mit dem dazugehörigen Erfassungszeitpunkt an einen der Landing-Zone zugeordneten Event Hub. Mehrere Nachrichten einer IoT-Sensorplattform mit dem gleichen Erfassungszeitpunkt stammen folglich von unterschiedlichen Sensoren einer IoT-Sensorplattform. Die Gerätedaten der IoT-Sensorplattformen sind über ein Repository zugreifbar und unterliegen keiner kontinuierlichen Veränderung. Sie beinhalten zum Beispiel technische Details der verbauten Sensoren in einer IoT-Sensorplattform oder die Koordinaten ihres Standorts. Die Qualität des Repositorys ist als gering einzustufen, da die Eintragung der Daten teilweise manuell erfolgt. In beiden Datensätzen dient jeweils eine ID der eindeutigen Identifizierung einer bestimmten IoT-Sensorplattform und eines bestimmten Sensortyps auf dieser Plattform.

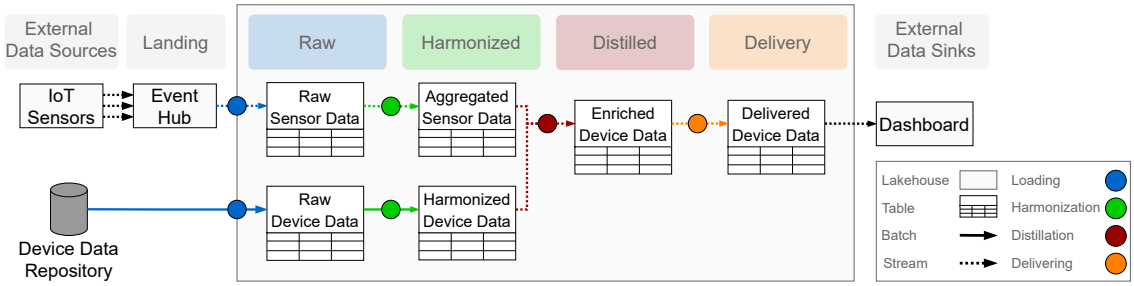

Abbildung 5.7.: Beispiel für das Daten-Pipeline-Modell einer Daten-Pipeline zur Anreicherung von IoT-Sensordaten.

Daten-Pipeline zur Anreicherung von IoT-Sensordaten

Abbildung 5.7 zeigt ein Daten-Pipeline-Modell, das Messwerte und Gerätedaten mehrerer IoT-Sensorplattformen für die Visualisierung in einem Dashboard kombiniert. Das Beispiel demonstriert, dass Batch- und Streaming-Daten im Rahmen einer Daten-Pipeline gemeinsam verarbeitet und die Resultate in einer Tabelle bereitgestellt werden können. Dazu umfasst die Daten-Pipeline sechs Pipeline-Schritte:

- Über einen Loading-Schritt werden die einzelnen Nachrichten einer IoT-Sensorplattform als Stream aus der Landing-Zone extrahiert und in die Raw-Zone des Lakehouses eingespeist. Dort werden sie in unverarbeitetem Zustand in einer Tabelle persistiert. Die Nachrichten der Sensoren einer IoT-Sensorplattform bilden jeweils eine Zeile dieser Tabelle.
- Ein weiterer Loading-Schritt lädt die Gerätedaten als Batch aus dem Repository in eine andere Tabelle der Raw-Zone. Die Ausführung dieses Batch-Verarbeitungsschrittes findet entsprechend eines definierten Zeitintervalls statt, das auf die potentielle Änderung der gespeicherten Gerätedaten abgestimmt ist.
- Auch die Überführung der Daten in die Harmonized-Zone erfolgt getrennt für Batch- und Streaming-Daten über zwei Harmonization-Schritte. Die Streaming-Daten werden dabei aggregiert. Dazu extrahiert der verantwortliche Harmonization-Schritt noch nicht verarbeitete Rohdaten als Stream, gruppiert diese nach der ID der IoT-Sensorplattformen sowie nach der ID der Sensortypen und mittelt die Messwerte anschließend minütlich anhand ihres Erfassungszeitpunktes. Die Ergebnisse der Aggregation werden in einer Tabelle der Harmonized-Zone gespeichert.
- Ein zweiter Harmonization-Schritt persistiert die harmonisierten Geräterohdaten in der Harmonized-Zone. Er arbeitet im Batch-Modus und führt Datenbereinigungsmaßnahmen auf den Geräterohdaten aus, um eventuelle Fehler der manuellen Eintragung zu beheben.
- Nachdem beide Datensätze harmonisiert sind, erfolgt ihre Zusammenführung über einen Distillation-Schritt. Dieser konsumiert beide Tabellen als Stream, um sie über einen Join unter Einbezug der IDs der IoT-Sensorplattformen und der Sensortypen zu verknüpfen.
- Der anschließende Delivery-Schritt entspricht einer Kopie der Tabelle aus der Distilled-Zone und deren Ablage in der Delivery-Zone. Je nach Anforderungen der Dashboard-Anwendung können einfache Transformationen auf den Daten ausgeführt werden, um diese in eine anwendungsgerechte Darstellungsform zu bringen.

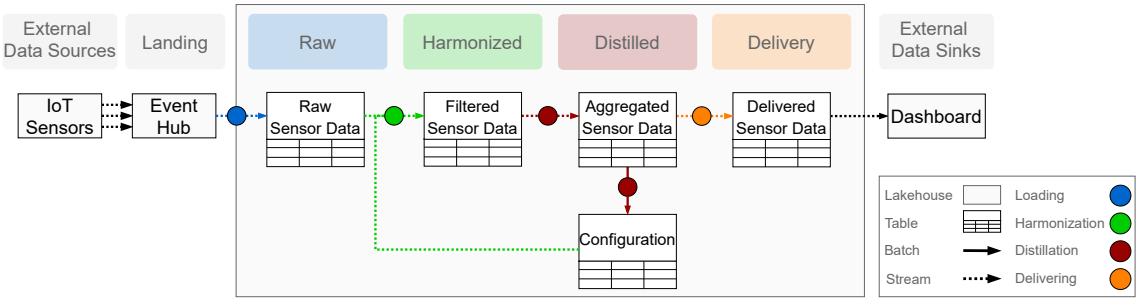

Abbildung 5.8.: Beispiel für das Daten-Pipeline-Modell einer selbstanpassenden Daten-Pipeline.

Selbstanpassende Daten-Pipeline

In Abbildung 5.8 ist ein Beispiel einer selbstanpassenden Daten-Pipeline dargestellt, die sich aus fünf Pipeline-Schritten zusammensetzt. Sie beschränkt sich auf die Verarbeitung der kontinuierlich erzeugten Messwerte und bezieht keine Batch-Daten zu technischen Details der IoT-Sensorplattformen mit ein. Vergleichbar mit dem vorherigen Beispiel besteht das Ziel der Daten-Pipeline darin, die Streaming-Daten für die Anzeige auf einem Dashboard minütlich zu aggregieren. Für dieses Beispiel wird jedoch angenommen, dass in den Messwerten häufig signifikante Ausreißer enthalten sind, die nicht in eine Verarbeitung einfließen sollten. Daher wird ein Mechanismus zur Selbstanpassung der Daten-Pipeline implementiert, der die Ergebnisse eines nachgelagerten Pipeline-Schritts nutzt, um die Verarbeitungslogik eines vorgelagerten Pipeline-Schritts besser zu konfigurieren. Zunächst ist ein Loading-Schritt erforderlich, der neu eingetroffene Messwerte aus der Landing-Zone in eine Tabelle der Raw-Zone überführt. Der darauffolgende Harmonization-Schritt ist Teil eines Zyklus, der außerdem zwei Distillation-Schritte umfasst. Ziel dieses Zyklus ist es, die Schwellenwerte zur Filterung der Rohdaten während des Harmonization-Schritts so zu wählen, dass signifikante Ausreißer, die in neuen Streaming-Daten vorhanden sein könnten, zuverlässig eliminiert werden. Da die Messwerte der IoT-Sensorplattformen zeitlichen Schwankungen unterliegen, ist es nicht sinnvoll, sich bei dieser Aufgabe auf feste Schwellenwerte zu verlassen. Stattdessen bezieht der Harmonization-Schritt die aktuell anzubringenden Schwellenwerte aus einer separaten Konfigurationstabelle und soll mithilfe dieser zu hohe oder zu niedrige Messwerte von der weiteren Verarbeitung ausschließen. Die Ergebnisse der Filterung werden in einer Tabelle der Harmonized-Zone gespeichert. Anschließend nutzt ein Distillation-Schritt die fehlerbereinigten Daten als Eingabe, um diese wie im vorherigen Beispiel unter Verwendung der eindeutigen IDs der IoT-Sensorplattformen und der Sensortypen im Streaming-Modus minütlich zu aggregieren und in einer Tabelle der Distilled-Zone abzulegen. Diese aggregierten Daten werden nicht nur zu Visualisierungszwecken an das Dashboard bereitgestellt, sondern auch periodisch im Batch-Modus über einen Distillation-Schritt verarbeitet. Er kann beispielsweise darauf ausgelegt sein, die von den einzelnen Sensoren innerhalb der letzten Stunde gesammelten Daten zu analysieren. Diese aus historischen Daten gewonnene Information wird dann genutzt, um Rückschlüsse auf die akzeptable Schwankung der Messwerte zu ziehen, die aufgrund sich ändernder Umgebungsbedingungen an den Installationsorten der IoT-Sensorplattformen zu erwarten ist und nicht durch signifikante Ausreißer verursacht wird. Für die Konfigurationstabelle können daraus absolute Schwellenwerte abgeleitet werden, die in die Filterung neu eintreffender Streaming-Daten eingehen.

5.5. Konzepte zur Entwicklung eines Modellierungswerkzeugs

Die manuelle Definition von Daten-Pipelines birgt je nach Anzahl und Abhängigkeiten der einzelnen Pipeline-Schritte ein hohes Maß an Komplexität und Fehleranfälligkeit. Deshalb wird in diesem Abschnitt ein Konzept zur Entwicklung eines Modellierungswerkzeugs vorgestellt, das Nutzer bei der Definition von Daten-Pipelines in Lakehouses unterstützen soll. Aus den nachfolgenden Unterabschnitten geht außerdem hervor, wie und zu welchem Grad sich das Konzept an die Model Driven Architecture (MDA) anlehnt. Für den Kontext einer MDA-Spezifikation ist die Bedeutung bestimmter Begriffe genau festgelegt. Brambilla et al. [BCW17] stellen eine Definition für *System*, *Modell* und *Plattform* bereit. Diese Sichtweise wird im Folgenden übernommen und für Daten-Pipelines in Lakehouse präzisiert:

- Ein *System* ist zentraler Gegenstand einer MDA-Spezifikation. Es kann beispielsweise ein Programm oder eine Kombination von Teilen verschiedener Systeme sein. Da Daten-Pipelines im Fokus der in dieser Arbeit vorgestellten Konzepte stehen, stellen sie im Zusammenhang mit MDA die zu modellierenden Systeme dar.
- Jede Darstellung eines Systems oder seiner Umgebung wird als *Modell* bezeichnet. Die in den Abschnitten 5.2, 5.3 und 5.4 vorgestellten Modelle sind folglich auch im MDA-Kontext als Modelle einzuordnen.
- Eine *Plattform* ist als eine Reihe von Teilsystemen und Technologien zu sehen, deren Funktionalitäten auf die Erreichung eines bestimmten Ziels ausgerichtet ist. Bezogen auf die hier betrachteten Daten-Pipelines steht hinter dem Begriff die Menge von Technologien, die zur Ausführung einer Daten-Pipeline in einem Lakehouse erforderlich sind, beispielsweise ein Lakehouse-Framework und ein Prozessierungsframework.

5.5.1. Anforderungen an das Modellierungswerkzeug

Für die Umsetzung eines Modellierungswerkzeugs bieten sich grundsätzlich eine Reihe von Vorgehensweisen an. Das entwickelte Konzept ist maßgeblich über drei Anforderungen motiviert, die grundlegende Eigenschaften des Modellierungswerkzeugs formulieren. Die Anforderungen sind abstrakt gehalten und sollen die Grundidee des Modellierungswerkzeugs vermitteln. Deshalb beschränkten sie sich auf die folgenden Punkte:

- Das Modellierungswerkzeug soll plattformunabhängig sein. Hinter dieser Anforderung steht der Anspruch, nicht an eine bestimmte Hardware- oder Softwareplattform gebunden zu sein. Bezogen auf eine Daten-Pipeline innerhalb eines Lakehouse bedeutet dies, dass das Modellierungswerkzeug nicht auf ein bestimmtes Lakehouse-Framework oder Prozessierungsframework eingeschränkt sein soll. Stattdessen soll es auf Erweiterbarkeit ausgelegt sein, so dass neue Technologien oder Frameworks mit möglichst geringem zusätzlichem Aufwand unterstützt werden können.
- Das Modellierungswerkzeug soll es ermöglichen, ausführbare Daten-Pipelines deklarativ zu definieren. Durch die Definition einer Daten-Pipeline über einen deklarativen Ansatz sollen sich Nutzer des Modellierungswerkzeugs auf die Logik konzentrieren können, die

für zur Bewältigung der Datenverarbeitungsaufgabe erforderlich ist. Sie sollen sich nicht um möglicherweise komplexe Implementierungsdetails kümmern müssen, sondern auf das Ergebnis, welches über eine Daten-Pipeline erzeugt werden soll.

- Das Modellierungswerkzeug soll einen möglichst hohen Automatisierungsgrad aufweisen. Diese Anforderung bezieht sich auf das Maß, in dem der Modellierungsprozess von der Dateneingabe eines Nutzers bis hin zur Bereitstellung eines ausführbaren Code-Artefakts automatisiert ist. Dadurch soll der Zeitaufwand für den Aufbau einer Daten-Pipeline und das Risiko von fehlerhaftem Code gering gehalten werden.

5.5.2. Einordnung der Modelle in eine vierstufige MOF-Hierarchie

Die Meta Object Facility (MOF) ist eine Spezifikation der Object Managment Group (OMG) und dient der Erstellung von Metamodellen und Modellen in der Softwareentwicklung. Durch ihre jeweilige Definition in einem gemeinsamen Regelwerk soll MOF eine systematische Modelltransformation erleichtern. Dazu arbeitet MOF nach einer 4-stufigen Hierarchie, in der ein System durch Modelle verschiedener Modellebenen spezifiziert wird. Die Modellebenen bauen hierarchisch aufeinander auf und unterscheiden sich in ihrem Abstraktionsgrad [Poe06]. Eine Daten-Pipeline, wie sie im Fokus dieser Arbeit steht, lässt sich als MOF-basiertes System darstellen. Wie in Abbildung 5.9 visualisiert, können die in den Abbildungen 5.1, 5.5, 5.7 und 5.8 eingeführten Modelle einer Daten-Pipeline verschiedenen Modellebenen der MOF-Hierarchie zugeordnet werden. Die Zuordnung begründet sich in der Bedeutung der verschiedenen Modellebenen:

M3: Die oberste und abstrakteste Modellebene M3 besteht aus einer Reihe von Meta-Meta-Klassen, die auch als MOF-Klassen bezeichnet werden. Sie dienen der Klassifikation von Elementtypen, aus denen sich ein Metamodell der darunterliegenden Modellebene M2 zusammensetzt [Poe06]. Da das Metamodell und verfeinerte Metamodell für Daten-Pipelines als ER-Diagramme modelliert sind, verbirgt sich hinter der MOF-Klasse der obersten Modellebene das Metamodell für ER-Diagramme. Insbesondere enthält dieses die Entitäten *Entity* und *Relationship*, um die grundlegenden Bausteine eines ER-Diagramms auf Modellebene M2 zu beschreiben [LS14].

M2: In der zweiten Modellebene M2 sind Metamodelle anzusiedeln, die zur Typisierung von Modellen der nächst niedrigeren Modellebene M1 vorgesehen sind. Genauer definiert ein MOF-Metamodell eine Reihe von Entitätstypen, Beziehungstypen und deren zulässige Verknüpfung, um die Erstellung von Modellen einer bestimmten Domäne zu ermöglichen [Poe06]. Das Metamodell in Abbildung 5.1 und das verfeinerte Metamodell in Abbildung 5.5 gehören dieser Modellebene an. Beide M2-Metamodelle können dazu verwendet werden, um ein M1-Modell einer Daten-Pipeline der darunterliegenden MOF-Modellebene zu definieren. Diese M1-Modelle können sich in den Typen ihrer Zonen, Tabellen und Pipeline-Schritte unterscheiden, da das Metamodell für Daten-Pipelines aus Abbildung 5.1 diese Entitätstypen nicht weiter untergliedert und beliebige Verfeinerungen dieser zulässt.

M1: Diese Ebene besteht aus Modellen, die unter Verwendung der M2-Metamodelle definiert und manipuliert werden können [Poe06]. Demzufolge sind die Daten-Pipeline-Modelle der Abbildungen 5.7 und 5.8 als Beispiele für die Modellebene M1 anzusehen. Entsprechend des

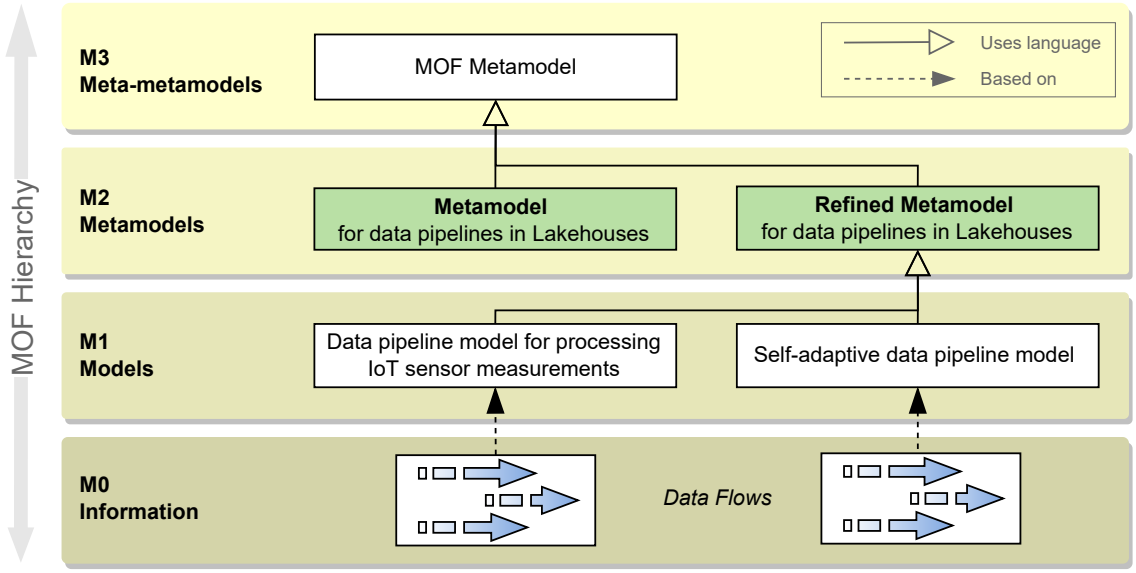

Abbildung 5.9.: Schematische Übersicht über die Zuordnung der vorgestellten Metamodelle und Modelle zu den vier Modellebenen der MOF-Hierarchie.

betrachteten MOF-Systems muss sich ein Ersteller einer Daten-Pipeline an die Vorgaben eines M2-Metamodells halten, um valide Daten-Pipeline-Modelle zu definieren. Deren Ausführung ist nicht Teil dieser, sondern der darunterliegenden Modellebene.

M0: Die unterste Ebene stellt die niedrigste Abstraktionsebene dar und besteht aus spezifischen Modellinstanzen. Diese repräsentieren die eigentlichen Daten als physische Ressourcen, die im System gespeichert oder bearbeitet werden [Poe06]. Eine Daten-Pipeline stellt sich auf dieser Ebene als Datenstrom dar, wobei die einzelnen Datensätze bei ihrer Einspeisung und Überführung zwischen Tabellen gemäß den Schritten der Daten-Pipeline verarbeitet werden.

5.5.3. Modelle und Transformationen der MDA

Um eine Software zu entwickeln, sind eine Reihe von modellbasierten Ansätzen denkbar. Die OMG hat in diesem Kontext die Model Driven Architecture vorgeschlagen, welches sich zu einem der bekanntesten Modellierungsframeworks entwickelt hat [BCW17]. MDA zielt darauf ab, die Qualität, Konsistenz und Wartbarkeit von Softwaresystemen zu verbessern. Dies soll durch die Verwendung von Modellen erreicht werden, die als primäre Artefakte im Vordergrund der Softwareentwicklung stehen. Über Modelle werden die Anforderungen, die Struktur, das Verhalten und die Funktionalität von Systemen auf verschiedenen Abstraktionsebenen festgehalten. Dabei stellt MDA im Gegensatz zu MOF keine Spezifikation der OMG dar. Vielmehr beschreibt MDA einen Ansatz zur modellbasierten Softwareentwicklung, der auf OMG-Spezifikationen wie UML oder MOF aufbaut [BCW17]. Der Zusammenhang zwischen MDA und MOF erschließt sich anhand der Systemspezifikationen, die aus einem MDA-basierten Entwicklungsprozess hervorgehen. MDA sieht vor, dass diese als eine Reihe von Modellen und Transformationen organisiert sind. Indem jedes Modell konform zur Sprache eines übergeordneten Metamodells erstellt wird, können systematische Transformationen zwischen den Modellen durchgeführt werden. Die Untergliederung des mehrschichtigen Architekturrahmens der

MDA erfolgt über die in Abschnitt 5.5.2 aufgezeigte MOF-Hierarchie. MDA-Modelle verschiedener MOF-Modellebenen unterscheiden sich im Abstraktionsgrad, den ihre Sicht auf das zu modellierende System einnimmt [FPKJ10]. Dieser Aspekt wird über die vertikale Dimension von Abbildung 5.10 veranschaulicht. Auch in der horizontalen Dimension lassen sich die dort abgebildeten Modelle untergliedern, was den multiperspektivischen Ansatz von MDA zum Ausdruck bringt. Horizontal sind die Modelle anhand ihrer Technologieabhängigkeit voneinander abzugrenzen. Von links nach rechts werden die Modelle hinsichtlich bestimmter Plattformen oder Technologien zunehmend spezifischer. Dies bewirkt eine Trennung von geschäftlichen und technischen Aspekten eines Systems. Dadurch kann ein Unternehmen, in dem ein MDA-basierter Ansatz verwendet wird, flexibel auf neue technologische Entwicklungen reagieren kann, ohne die Geschäftslogik anpassen zu müssen. Wie die Zusammenhänge zwischen MDA-Modellen im Detail aussehen und welcher Zweck sich jeweils mit ihnen verbindet, soll im Folgenden herausgearbeitet werden. Analog zur Vorstellung der MOF-Hierarchie in Abschnitt 5.5.2 wird dabei das System einer Daten-Pipeline und deren Modellierung auf den MDA-Kontext übertragen:

CIM: Das Computation Independent Model (CIM) isoliert den technologieunabhängigen Kern eines Systems und kann auch als Geschäfts- oder Domänenmodell bezeichnet werden. Es beschreibt auf abstrakter Ebene den Kontext, die Anforderungen und den Zweck einer Systemlösung. Seine wesentlichen Kennzeichen bestehen in der fehlenden Bindung an eine konkrete Implementierung oder Plattform [BCW17]. Als berechnungsunabhängiges Metamodell der MOF-Modellebene M2 dient das in Abbildung 5.1 dargestellte Metamodell für Daten-Pipelines. Dieses führt ein Vokabular ein, welches für beliebige Daten-Pipelines in Lakehouses anwendbar ist. Es zeigt Beziehungen zwischen Entitätstypen auf, die den Aufbau von Daten-Pipelines für Anwendungsfälle im Lakehouse-Kontext gestatten sollen.

PIM: Ein Platform Independent Model (PIM) drückt sich verglichen zu einem CIM spezifischer aus. Es bildet die Struktur und das Verhalten eines Systems ab, ohne sich von einer konkreten Plattform oder Technologie abhängig zu machen. Aufgrund dieser Unabhängigkeit kann ein PIM für verschiedene Plattformen verwendet werden [BCW17]. Als plattformunabhängiges Metamodell kann das verfeinerte Metamodell für Daten-Pipelines aus Abbildung 5.5 betrachtet werden. Es ist ebenso in der MOF-Modellebene M2 anzusiedeln und entsteht durch Instanziierung des Entitätstyps Zone aus dem berechnungsunabhängigen Metamodell. Da im PIM also konkrete Zonen festlegt werden, ist es nicht mehr als berechnungsunabhängig einzustufen. Beispielsweise passt es nicht zu einem Unternehmen, dessen Lakehouse in mehr als fünf Zonen untergliedert ist und Daten-Pipelines zwischen allen Zonen erfordert. Das plattformunabhängige Metamodell für Daten-Pipelines sieht lediglich die fünf im Zonenreferenzmodell für Data Lakes enthaltenen Zonen vor. Auf der darunterliegenden MOF-Modellebene M1 sind außerdem die beiden Daten-Pipeline-Modelle der Abbildungen 5.7 und 5.8 als plattformunabhängige Modelle einer Daten-Pipeline anzusehen. Sie halten sich an die strukturellen Vorgaben, die das plattformunabhängige Metamodell in Bezug auf eine Daten-Pipeline macht, ohne Aussagen über plattformspezifische Aspekte zu treffen.

PSM: Ein Platform Specific Model (PSM) beinhaltet eine detaillierte Darstellung des betrachteten Systems, die für eine bestimmte Plattform spezifisch ist. Anhand des Modells müssen alle erforderlichen Informationen über das Verhalten und die Struktur des Systems festgelegt werden, um dieses in ausführbaren Code überführen zu können [BCW17]. Für eine Daten-Pipeline innerhalb eines Lakehouses muss ein plattformspezifisches Modell beispielsweise

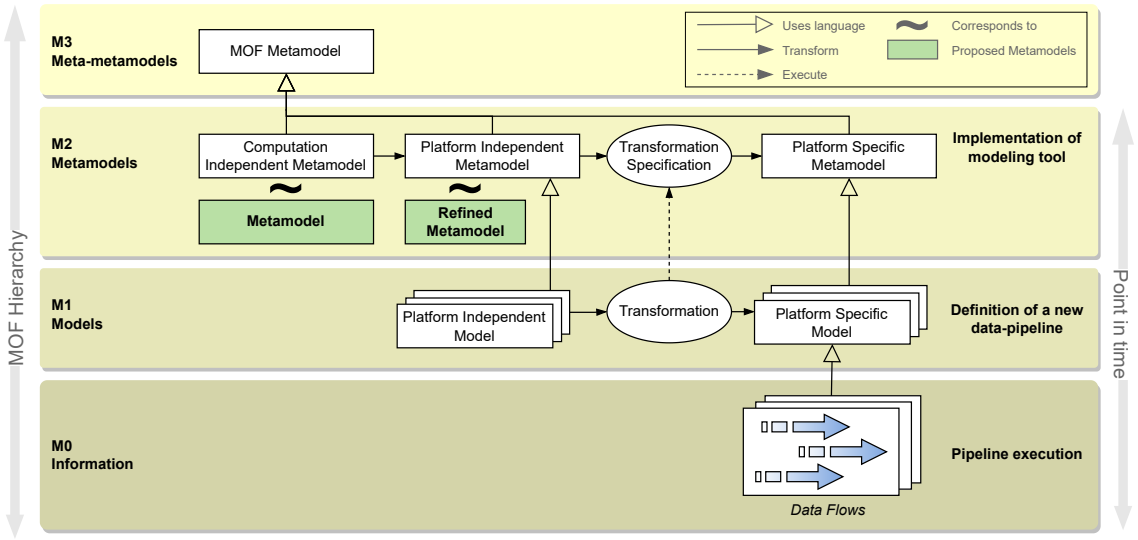

Abbildung 5.10.: Schematische Übersicht über die MDA-Transformationen, die bei der Implementierung eines Modellierungswerkzeugs gemäß den vorgestellten Konzepten ermöglicht werden müssen. Im Rahmen des MDA-basiertes Ansatzes kann das vorgeschlagene verfeinerte Metamodell die Rolle eines plattformunabhängigen Metamodells übernehmen.

die Eigenheiten eines bestimmten Lakehouse-Frameworks berücksichtigen. Ein plattformspezifisches Metamodell oder Modell wird im Rahmen dieser Arbeit nicht explizit definiert. Aus den in Abschnitt 5.5.4 beschriebenen Transformationen zwischen MDA-Modellen kann abgeleitet werden, wie sich ein solches vom plattformunabhängigen Metamodell abgrenzt.

Abbildung 5.10 zeigt nicht nur die verschiedenen Arten von MDA-Modellen, sondern auch, wie diese über Transformationen aufeinander abgebildet werden können. Aus einem CIM können über Transformationen verschiedene PIMs abgeleitet werden, die wiederum in verschiedene PSMs transformiert werden können. Dabei wird ein Transformationsprozess anhand von Informationen gesteuert, die in einem Mapping fixiert sind. Ein Mapping spezifiziert Korrespondenzen zwischen den Elementen zweier Modelle. Es gibt an, wie die Elemente des eingegebenen Modells auf die Elemente des resultierenden Modells abgebildet werden sollen. Um plattformspezifische Details in den Transformationsprozess einzubinden, verwendet MDA sogenannte *Marks*. Diese Annotationen werden auf Elemente eines PIMs angewendet, um festzulegen, wie diese in Elemente eines PSMs umgewandelt werden sollen. Eine bestimmte Plattformzuordnung des PSMs kann folglich erzielt werden, indem auf das PIM eine bestimmte Annotation angewendet wird. Der Mehrwert einer MDA ergibt sich, indem die Regeln eines Mappings auf Ebene der Metamodelle definiert werden. Dies ermöglicht es, die Transformation aller Instanzen von Typen des plattformunabhängigen Metamodells in Instanzen von Typen des plattformspezifischen Metamodells zu automatisieren. Zeitaufwendige manuelle Transformationen auf Ebene der Modelle sollen auf diese Weise vermieden werden. Dabei ist die Überführung eines PIMs in ein PSM nicht auf einen Schritt beschränkt. Über einen mehrstufigen Prozess kann ein PIM schrittweise auf ein immer detaillierteres PSM abgebildet werden. Am Ende dieser Kette steht Code, der tatsächlich ausführbar ist [BCW17; FPKJ10].

5.5.4. Einbezug der MDA-Konzepte in einem Modellierungswerkzeug

Die MDA-Konzepte, die im vorherigen Abschnitt 5.5.3 eingeführt wurden, können in einem Modellierungswerkzeug für Daten-Pipelines in Lakehouses berücksichtigt werden. Ohne auf konkrete Implementierungsdetails einzugehen, wird im Folgenden skizziert, wie das Modellierungswerkzeug arbeitet und welche Funktionalitäten einem Nutzer geboten werden können. Verglichen mit einem MDA-Modellierungswerkzeug ist es als spezialisierter einzuordnen, da es nicht darauf ausgelegt sein soll, Transformationen beliebiger Modelle zu unterstützen. Stattdessen ist seiner Implementierung das plattformunabhängige Metamodell und das plattformspezifische Metamodell für Daten-Pipelines in Lakehouses hinterlegt, sowie die Spezifikation der Transformation zwischen diesen. Um Plattformabhängigkeit herzustellen und auf Ebene der Modelle die Generierung von ausführbarem Code zu ermöglichen, sind drei verschiedene Annotationsarten vorgesehen:

- Das Festlegen des Lakehouse-Frameworks bestimmt, in welchem Format die von der Daten-Pipeline gelesenen und geschriebenen Daten abgelegt sind. In einem plattformspezifischen Modell ergänzt die Annotation alle Entitäten vom Typ *Table*. Die Optionen für diese Annotation sind auf die gängigen Lakehouse-Frameworks Delta Lake, Apache Hudi und Apache Iceberg beschränkt [JKP+23].
- Ein Prozessierungsframework ist für die Verarbeitung der Lakehouse-Daten verantwortlich. Über dessen Funktionalität kann eine Daten-Pipeline auf die Daten einer Lakehouse-Tabelle zugreifen und die Ergebnisse eines Pipeline-Schrittes wieder in einer Tabelle persistieren. In einem plattformspezifischen Modell ergänzt die Annotation alle Entitäten vom Typ *Processing*. Anzumerken ist, dass nicht jedes Prozessierungsframework für diesen Einsatz in Frage kommt. Bereits auf der Ebene des berechnungsunabhängigen Metamodells ist beispielsweise festgehalten, dass dieses sowohl die Batch- als auch Stream-Verarbeitung von Daten unterstützen muss. Ein geeignetes Prozessierungsframework muss den gesamten Funktionsumfang abdecken, der im berechnungsunabhängigen Metamodell für die Verarbeitung eines Pipeline-Schrittes vorgegeben ist. Als Beispiel für ein Prozessierungsframework kann Apache Spark angeführt werden, das in Abschnitt 6.1.4 beleuchtet wird.
- Für jeden Pipeline-Schritt kann benutzerdefinierter Code in einer bestimmten Programmiersprache angegeben werden, um die Logik eines Pipeline-Schrittes zu spezifizieren. In einem plattformspezifischen Modell ergänzt die Annotation alle Entitäten vom Typ *Processing*. Die Optionen für diese Annotationsart sind im Allgemeinen an das ausgewählte Prozessierungsframework gekoppelt. Dieses ist dafür zuständig, den Code auf den Lakehouse-Daten auszuführen. Im Falle von Apache Spark kommen zum Beispiel die Programmiersprachen Python oder Scala in Frage.

Allen drei Annotationsarten ist gemeinsam, dass sie zwar bestimmte Entitäten ergänzen, aber ein Nutzer sie nicht auf der feingranularen Ebene einzelner Entitäten setzen muss. Es wird angenommen, dass eine bestimmte Daten-Pipeline mit genau einem Lakehouse-Framework und mit genau einem Prozessierungsframework arbeitet. Auch die Angabe des benutzerdefinierten Codes soll in einer Daten-Pipeline einheitlich in einer Programmiersprache erfolgen.

Ein Nutzer des Modellierungswerkzeugs arbeitet ausschließlich auf den Artefakten der MOF-Ebene M1. Das Werkzeug muss deshalb eine Möglichkeit bieten, plattformunabhängige Modelle zu definieren. Dies kann grafisch oder über einen textbasierten Ansatz erfolgen. Ein plattformunabhängiges Modell ist nur dann valide, wenn es sich an die Vorgaben des plattformunabhängigen Metamodells

hält. Folglich muss das Modellierungswerkzeug bereits beim Aufbau eines neuen plattformunabhängigen Modells auf Konformität zum Metamodell prüfen oder diese Prüfung in einem nachgelagerten Validierungsschritt anbieten. Erst wenn der Nutzer ein konformes, plattformunabhängiges Modell aufgebaut hat, kann er die Transformation in ein plattformspezifisches Modell durchführen. Hierfür verlangt das Modellierungswerkzeug die Festlegung des Lakehouse-Frameworks, des Prozessierungsframework und der Programmiersprache. Diese Angaben sind zur Steuerung des Transformationsprozesses zwingend erforderlich, damit ein vollständiges plattformspezifisches Modell resultiert. Indem das Modellierungswerkzeug mehrere Transformationsschritte aneinanderreiht, kann es das plattformspezifische Modell sukzessive um weitere Implementierungsdetails verfeinern. Zusätzlich zu den bereits genannten Annotationen können weitere Benutzereingaben erforderlich werden, damit die Generierung von ausführbarem Pipeline-Code erfolgen kann.

Um die Daten-Pipeline auszuführen, übergibt das Modellierungswerkzeug den Pipeline-Code an eine Orchestrierungsengine. Diese ist dafür verantwortlich, die Ausführung der Daten-Pipeline auf koordinierte und effiziente Weise zu verwalten. Genaue Ausführungsanweisungen sind im generierten Pipeline-Code festgehalten. Beispielsweise kann sie eine automatische Wiederholung von fehlgeschlagenen Pipeline-Schritten veranlassen. Auch die Bekanntmachung des Fehlers an einen menschlichen Verantwortlichen kann zu den Aufgaben der Orchestrierungsengine zählen. Daneben bietet sie einen Mehrwert, wenn es um die Überwachung der Pipeline-Ausführung geht. In das Modellierungswerkzeug kann eine Visualisierung des Datenflussgraphen eingebettet werden, um den aktuellen Zustand der Datenbewegungen im Lakehouse darzustellen. Die dazu erforderlichen Informationen stammen von der Orchestrierungsengine, die zwar nicht in die Datenverarbeitung involviert ist, aber Kenntnis vom Ausführungszustand des Pipeline-Codes besitzt. Wenn es beim Ausführen des Codes eines Pipeline-Schrittes zu einem Fehler kommt und eine automatische Wiederholung fehlschlägt, werden durch den Pipeline-Schritt keine Daten mehr verarbeitet. Die korrespondierende Kante im Datenflussgraph kann farblich gekennzeichnet werden, um auf den Fehler aufmerksam zu machen.

Das skizzierte Modellierungswerkzeug berücksichtigt alle Anforderungen, die in Abschnitt 5.5.1 aufgeführt sind. Entscheidend dafür ist die Anlehnung des Modellierungswerkzeugs an MDA-Konzepte, deren Beitrag zur Erfüllung der Anforderungen im Folgenden herausgestellt wird:

- Das Modellierungswerkzeug ist als plattformunabhängig zu bezeichnen, da seine Arbeitsweise keine Abhängigkeit zum Funktionsumfang bestimmter Technologien herstellt. Es verwirklicht eine strikte Trennung der allgemeinen Beschreibung einer Daten-Pipeline in einem plattformunabhängigen Modell von plattformspezifischen Implementierungsdetails. Um das Modellierungswerkzeug für ein neues Lakehouse-Framework oder Prozessierungsframework zu verwenden, ist deshalb keine Änderung an bereits vorhandenen Daten-Pipeline-Modellen vorzunehmen. Stattdessen müssen die Eigenheiten der neuen Technologien in Form von Annotationen beschrieben werden, damit diese im Zuge des Transformationsprozesses an plattformunabhängige Modelle angebracht werden können.
- Das Modellierungswerkzeug realisiert einen deklarativen Ansatz zur Definition von Daten-Pipeline-Modellen. Unabhängig davon, ob die Eingabe eines Nutzers grafisch oder textbasiert erfolgt, genügt es, über das Modellierungswerkzeug den gewünschten Datenfluss zwischen Tabellen zu beschreiben. Der Nutzer muss sich hingegen nicht mit den einzelnen Schritten befassen, die zur praktischen Realisierung des Datenflusses erforderlich sind. Diese erschließen sich aus dem generierten Pipeline-Code.

- Aufgrund der Fähigkeit zur automatisierten Codegenerierung ist der Automatisierungsgrad des Modellierungswerkzeugs als hoch einzustufen. Ein Nutzer muss keinen Pipeline-Code schreiben, um die Verarbeitungslogik einer Daten-Pipeline zu spezifizieren. Der an MDA angelehnte Ansatz sorgt dafür, dass die manuellen Eingaben auf das plattformunabhängige Modell einer Daten-Pipeline und auf die Auswahl eines Lakehouse-Frameworks, eines Prozessierungsframeworks und einer Programmiersprache begrenzt werden können.

6. Prototypische Implementierung eines Modellierungswerkzeugs

Dieses Kapitel stellt die prototypische Implementierung eines Modellierungswerkzeugs für Daten-Pipelines in Lakehouses vor. Das Modellierungswerkzeug bietet Nutzern die Möglichkeit, Daten-Pipelines auf deklarative Weise zu definieren, auszuführen und zu überwachen. Außerdem können Nutzer den Inhalt von Tabellen mithilfe von benutzerdefinierten Abfragen visualisieren. Um diese Funktionalität zu unterstützen, kombiniert der Prototyp verschiedene Technologien. In Abschnitt 6.1 werden deren Eigenschaften beschrieben und insbesondere erläutert, welche Rolle ihnen in der prototypischen Implementierung zukommt. Der darauffolgende Abschnitt 6.2 geht auf den angewendeten Implementierungsansatz ein und stellt einen Bezug zu den in Kapitel 5 eingeführten Konzepten zur Entwicklung eines Modellierungsframeworks her. Dabei wird aufgezeigt, inwieweit sich der für den Prototyp verwendete Ansatz an den entwickelten Konzepten orientiert und auf welcher Modellhierarchie der Prototyp aufbaut. Architektonisch ist der Prototyp in ein Frontend, einen Webserver und ein Backend unterteilt. Durch die Interaktion dieser Komponenten können den Nutzern die zuvor erwähnten Funktionalitäten bereitgestellt werden. In Abschnitt 6.3 wird die Funktionsweise des Prototyps zusammengefasst, indem beschrieben wird, wie eine Benutzereingabe im Webserver und Backend weiterverarbeitet wird. Abschnitt 6.3.1 konzentriert sich auf die Definition und Ausführung von Daten-Pipelines, Abschnitt 6.3.2 auf die Überwachung und Abschnitt 6.3.3 auf die Abfrage von Verarbeitungsergebnissen. Abschließend führt Abschnitt 6.4 Eigenschaften des Prototyps auf. Dabei wird darauf eingegangen, wie und in welchem Umfang sich eine Eigenschaft im Prototyp widerspiegelt und welche Verbesserungsmöglichkeiten bestehen.

6.1. Auswahl der verwendeten Technologien

Die Implementierung des Prototyps erfolgte mittels der Programmiersprache *Python*. Um die angestrebte Funktionalität des Modellierungswerkzeugs umzusetzen, greift der Prototyp auf Python-Bibliotheken und Frameworks zurück, welche in unterschiedlichen architektonischen Komponenten Verwendung finden. Eine Übersicht darüber bietet Abbildung 6.1. Im Frontend, im Webserver und in der Orchestrierungsebene kommen Python Dash, Python Quart und Prefect zum Einsatz. Hinsichtlich der Umsetzung des Lakehouses und der Prozessierungsebene ist der Prototyp auf die Austauschbarkeit der Technologien ausgelegt. Für Lakehouses werden Delta Lake und Apache Iceberg unterstützt, für die Prozessierung Apache Spark. Im Folgenden werden diese Technologien vorgestellt und erläutert, warum sie sich für die prototypische Implementierung eignen.

6.1.1. Python Dash

Ein Nutzer des Prototyps sollte ausschließlich mit dem Frontend interagieren müssen, um Daten-Pipelines zu definieren und auszuführen. Daher sollte das Design der Benutzeroberfläche intuitiv sein und beim Entwurf von Daten-Pipelines unterstützen. Insbesondere sollte ein Nutzer keine Kenntnis von den internen Abläufen des Prototyps besitzen müssen. Um diese Anforderungen berücksichtigen zu können, bietet sich das Open-Source Framework Python Dash[1] an. Es unterstützt Entwickler bei der Erstellung von interaktiven Dashboards, Datenvisualisierungen und Analysewerkzeugen. So können Web-Anwendungen aus reinem Python-Code abgeleitet werden, ohne direkte Verwendung von Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) und JavaScript auf Entwicklerseite. Jede Dash-Anwendung besteht aus zwei grundlegenden Bestandteilen, nämlich einem Layout und einer Reihe von Callback-Funktionen. Die statische Sicht einer Dash-Anwendung wird durch ihr Layout bestimmt, für das eine Vielzahl vorgefertigter Komponenten verfügbar sind, wie Buttons, Dropdown-Menüs oder Checkboxes. Jeder Layout-Komponente können eine oder mehrere Callback-Funktionen zugewiesen werden. Diese Funktionen sind für das dynamische Verhalten einer Dash-Anwendung verantwortlich und werden als Reaktion auf bestimmte Benutzereingaben ausgeführt. Dabei unterstützt Dash sowohl clientseitige als auch serverseitige Callbacks.

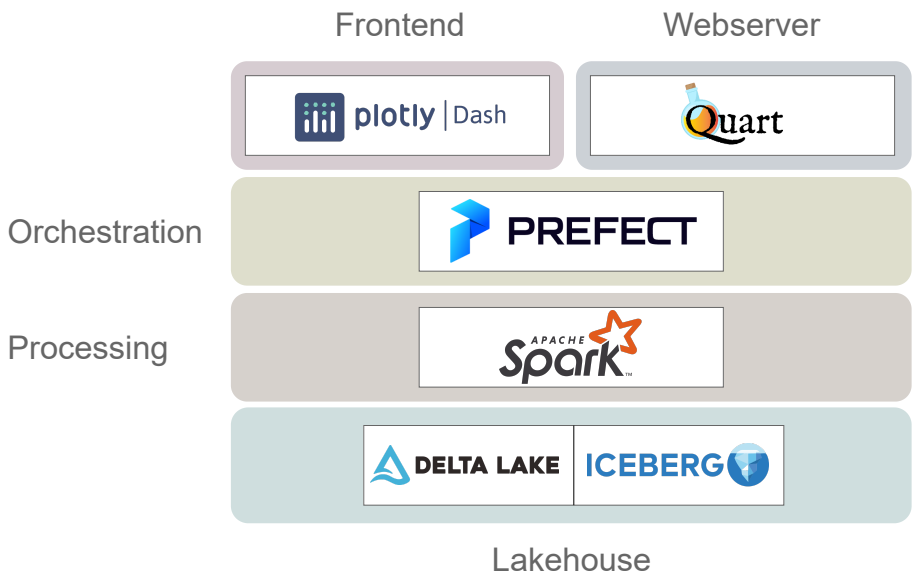

Abbildung 6.1.: Schematische Übersicht über die mehrschichtige Architektur des Prototyps. Jeder Komponente sind die eingesetzten Technologien zugeordnet.

[1] https://dash.plotly.com

6.1.2. Python Quart

Python Quart[2] ist ein Web-Framework, das darauf ausgerichtet ist, asynchronen Code in Webserver-Funktionen zu unterstützen. Asynchron implementierte Webserver-Funktionen erlauben die parallele Verarbeitung mehrerer Anfragen, was zu einer höheren Effizienz und Geschwindigkeit führt. Im Vergleich zu synchronen Ansätzen, bei denen Anfragen nacheinander abgearbeitet werden, kann Quart somit Engpässe und Verzögerungen reduzieren. Darüber hinaus kann Quart durch die Skalierung auf mehrere Server eine verbesserte Lastverteilung und Ausfallsicherheit gewährleisten und deshalb trotz wachsender Nutzerzahl eine schnelle Verarbeitung von Anfragen ermöglichen. Zudem kann die serverseitige Verarbeitung im Zusammenhang mit Daten-Pipelines zeitaufwendig sein, weshalb sichergestellt werden sollte, dass das Frontend auch bei länger dauernden Anfragen reaktionsfähig bleibt. Auch diese Anforderung kann durch die asynchrone Implementierung der Webserver-Funktionalität des Quart-Webservers erfüllt werden. Aufgrund der beschriebenen Eigenschaften eignet sich Quart als Web-Framework für den Einsatz im Webserver des Prototyps.

6.1.3. Prefect 2

Prefect 2[3] ist eine Open-Source-Plattform, die es ermöglicht, Workflows zu definieren, auszuführen und zu skalieren. Ein Workflow umfasst in diesem Zusammenhang eine potentiell mehrstufige Abfolge von Aufgaben und ihren Abhängigkeiten. Im Rahmen des Prototyps entspricht ein Workflow einer Daten-Pipeline, hier hebt die Bezeichnung als Workflow deren Orchestrier- und Ausführbarkeit hervor. Für den Prototyp spielt Prefect eine entscheidende Rolle, da mit dem Orchestrierungsframework mehrere Workflows parallel ausgeführt, verwaltet und überwacht werden können. Auf konzeptioneller Ebene setzt sich ein Workflow in Prefect aus zwei verschiedenen Elementen zusammen, sogenannten *Tasks* und *Flows*. Ein Task steht für eine Einheit, welche die Workflow-Logik in wiederverwendbare, individuelle Komponenten kapselt und als Teil eines Workflows ausgeführt wird. Mehrere Tasks werden in einem Flow-Objekt zusammengeführt. Flows stellen die kleinste ausführbare Einheit in Prefect dar und dienen als Container für die Workflow-Logik. Sowohl Tasks als auch Flows werden auf der Implementierungsebene als separate Funktionen abgebildet. Zur Definition beider Elemente kann das Application Programming Interface (API) von Prefect für Python genutzt werden. Im nachfolgenden Abschnitt 6.3.1 werden Codefragmente der prototypischen Implementierung präsentiert, wobei Listing 6.1 eine Task-Definition und Listing 6.2 eine Flow-Definition zeigt.

Nach der Definition eines Workflows kann dieser an die Workflow-Orchestrierungsengine von Prefect, namens *Prefect Orchestration Engine*, übergeben werden. Diese fungiert als Ausführungsschicht für Workflows, wobei die Ausführung beispielsweise auf lokalen Maschinen, in Cloud-Umgebungen oder in Kubernetes-Clustern stattfinden kann. Eine ihrer Hauptaufgaben besteht darin, die Ausführung der einzelnen Tasks eines Workflows basierend auf ihrer Reihenfolge und Abhängigkeiten, die vom Nutzer im Code festgelegt wurden, zu planen und zu steuern. Dabei richtet die Prefect-Orchestrierungsengine den Fokus auf eine effiziente, zuverlässige und reproduzierbare Ausführung der Workflows. Sie übernimmt unter anderem die bedarfsgerechte Skalierung von Workflows, die Behandlung von Fehlern während der Ausführung eines Tasks und die automatische Wiederholung

[2] https://pgjones.gitlab.io/quart
[3] https://docs.prefect.io

fehlgeschlagener Tasks, so dass Workflows zumindest bei temporären Fehlern fortgesetzt werden können. Während die Prefect-Orchestrierungsengine einen Workflow ausführt, können Nutzer dessen aktuellen Status über die Überwachungs- und Koordinierungsfunktionen der Prefect-API überwachen oder mit ihm interagieren. Dieser Status wird von der Prefect-Orchestrierungsengine verfolgt und auf dem aktuellem Stand gehalten. Eine andere Möglichkeit zur Einsicht der von der Prefect-Orchestrierungsengine bereitgestellten Informationen besteht darin, das Prefect User Interface (UI) zu verwenden, eine lokale, Open-Source-Benutzeroberfläche. Über diese können Nutzer neben dem Workflow-Status auch zusätzliche Informationen einsehen, wie zum Beispiel die zeitliche Historie aller bisherigen Workflow-Ausführungen, sowie ein Radardiagramm, das die Abhängigkeiten zwischen den einzelnen Tasks eines Workflows darstellt.

Neben der bereits beschriebenen Funktionalität haben zwei weitere Eigenschaften dazu beigetragen, dass Prefect als Orchestrierungsframework im Prototyp eingesetzt wird. Diese Eigenschaften können als Anforderungen an ein Orchestrierungsframework für Daten-Pipelines in Lakehouses aufgefasst werden. Wenn bei einer Weiterentwicklung des Prototyps eine Alternative zu Prefect in Betracht gezogen wird, müssen insbesondere die beiden folgenden Eigenschaften des Prefect-Frameworks auch vom neuen Framework erfüllt werden:

Vereinheitlichung von Batch- und Stream-Verarbeitung: Prefect bietet eine Orchestrierungsengine an, die sowohl für die Batch-Verarbeitung von Daten als auch für Echtzeit-Streaming-Pipelines geeignet ist. Die Entscheidung von Prefect, eine solche Engine zu entwickeln, erschließt sich aus den Nachteilen, die bei einer Trennung von Batch- und Streaming-Arbeitslasten durch zwei separate Systeme entstehen. Wie in Abschnitt 2.2.2 erläutert, können diese Nachteile bei der Verwendung von Lambda- und Kappa-Architekturen auftreten. Aus Sicht von Prefect besteht der Unterschied zwischen Batch- und Stream-Verarbeitung nicht in der Logik der einzelnen Tasks, sondern in der unterschiedlichen Häufigkeit ihrer Ausführung. Die Prefect-Orchestrierungsengine kann mit beiden Ausführungshäufigkeiten umgehen und soll als Orchestrierungsengine ausreichend sein, um eine Vielzahl von Datenverarbeitungsaufgaben zu bewältigen [Gel21].

DAG-freie Workflows: Gängige Workflow-Orchestrierungsengines wie Apache Airflow und Luigi erfordern, dass Workflows die Strukturvorgaben eines Directed Acyclic Graphs (DAG) einhalten. Prefect hingegen abstrahiert einen Workflow nicht als DAG. Um DAG-freie Workflows zuzulassen, wurde die Prefect-Orchestrierungsengine hinter eine API verlagert, wodurch die Eigenheiten der Orchestrierungsengine vom eigentlichen Workflow-Code getrennt werden. Dadurch entfällt zum einen der Übersetzungsschritt, der normalerweise erforderlich ist, um den Workflow-Code an die Strukturvorgabe eines DAGs anzupassen. Ein Workflow kann daher in nativem Python-Code definiert werden. Zum anderen ermöglicht die Einführung der API, dass Workflow-Erkennung und -Ausführung nicht mehr als getrennte Schritte betrachtet werden müssen, sondern beides gleichzeitig stattfinden kann. Deshalb soll Prefect in der Lage sein, beliebig lange Workflow-Ausführungen zu unterstützen, die auch Schleifen und bedingte Anweisungen umfassen dürfen. Zusätzlich können die Workflows dynamisch parametrisiert werden, so dass es möglich ist, Parameterwerte für eine bestimmte Workflow-Ausführung zu überschreiben, ohne den Workflow neu bereitstellen zu müssen [Gel22; Run22].

6.1.4. Apache Spark

Apache Spark[4] ist ein Open-Source-Framework, das für die verteilte Verarbeitung und Analyse von Big Data entwickelt wurde. Eines seiner wichtigsten Fähigkeiten ist die arbeitsspeicherinterne Datenverarbeitung. Im Gegensatz zu plattenbasierten Verarbeitungssystemen, bei denen Daten zunächst von Festplattenlaufwerken abgerufen werden müssen, können über Spark große Datenmengen im Arbeitsspeicher verarbeitet werden, was die Leistung von Big-Data-Anwendungen steigert. Architektonisch basiert Spark auf einem Master-Worker-Modell, bei dem der Master-Knoten die Koordination der Verarbeitungsaufgaben übernimmt, während die Worker-Knoten die Aufgaben ausführen. Aufgrund seiner Architektur kann Spark nicht nur lokal, sondern auch auf mehreren Knoten eines Clusters bereitgestellt werden. Hierfür können große Datensätze zerlegt und auf mehrere Knoten verteilt werden, damit sie anschließend von Spark parallel verarbeitet werden können. Auf jedem der Knoten wird ein Spark-Worker ausgeführt, der die ihm zugewiesene Datenportion verarbeitet. Durch die Verwendung von Spark auf einem Cluster ergeben sich zahlreiche Vorteile: Sollte beispielsweise ein Knoten ausfallen, kann ein anderer Knoten die Aufgaben des ausgefallenen Knotens übernehmen. Auf diese Weise wird sichergestellt, dass die Verarbeitung auch bei Ausfällen von Hardware oder anderen Problemen fortgesetzt werden kann. Neben der Fehlertoleranz bietet ein Spark-Cluster auch die Möglichkeit zur horizontalen Skalierung. Falls die Arbeitslast auf den vorhandenen Knoten zu groß wird, kann die Verarbeitungsgeschwindigkeit und -kapazität des Clusters durch Hinzufügen weiterer Knoten erhöht werden.

Dank seines breiten Funktionsumfangs eignet sich Spark für eine Vielzahl von Anwendungsfällen. Zum Beispiel ermöglicht Spark ML, eine Bibliothek innerhalb von Spark, den Aufbau von skalierbaren Machine-Learning-Pipelines. In solche Pipelines können Aufgaben wie das Trainieren, Testen und Bereitstellen von Modellen integriert werden. Neben der Verarbeitung großer Datenmengen im Batch-Modus bietet Spark auch die Möglichkeit, Echtzeit-Datenströme zu verarbeiten. Dies wird über das Modul Spark Structured Streaming[5] realisiert. Da es auf derselben Verarbeitungsengine wie Spark aufbaut, sind die bereits erwähnten Vorteile von Spark bezüglich seiner arbeitsspeicherinternen, parallelen Datenverarbeitung auf Spark Structured Streaming übertragbar. Um Streaming-Daten in Echtzeit zu verarbeiten, werden eingehende Daten gemäß eines konfigurierbaren Intervalls in Micro-Batches aufgeteilt. Im Rahmen von Spark Structured Streaming stellt ein Micro-Batch ein kleines Datenpaket dar, das als eine Einheit verarbeitet wird. Nach dem Einlesen der Streaming-Daten können auf einem Micro-Batch eine Reihe von Transformationen durchgeführt werden, wie z.B. Filterungen, Aggregationen oder Joins von Streaming-Daten.

Spark ist zu verschiedenen Werkzeugen und Frameworks kompatibel, darunter Apache Kafka und den Lakehouse-Frameworks Delta Lake, Apache Hudi und Apache Iceberg. Es ist jedoch hervorzuheben, dass Apache Spark kein obligatorisches Prozessierungsframework für den Prototypen darstellt. Im Gegensatz zu den bisher vorstellten Bausteinen Python Dash, Python Quart und Prefect könnte Spark durch andere Prozessierungsframeworks ersetzt werden, ohne den in Abschnitt 6.2 beschriebenen Implementierungsansatz zu ändern. Die Abhängigkeit des Prototyps von Spark manifestiert sich hauptsächlich in seinem Backend, das die Pipeline-Schritte in PySpark-Code implementiert. Bei PySpark handelt es sich um eine Python-Bibliothek von Spark, mit der sich Spark-Funktionen in Python-Code einbinden lassen. Im Folgenden werden die grundlegenden Anforderungen an

[4] https://spark.apache.org

[5] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

ein Prozessierungsframework zusammengefasst, die erfüllt sein müssen, damit es als Alternative zu Apache Spark im Backend des Prototyps eingesetzt werden kann. Denkbar wäre auch eine Kombination von Frameworks, welche die Anforderungen gemeinsam abdecken:

- Performanz durch Unterstützung arbeitsspeicherinterner Datenverarbeitung
- Horizontale Skalierbarkeit für dynamische Anpassung an Arbeitslasten
- Lesen und Schreiben von Daten als Batch und Stream in verschiedenen Lakehouse-Formaten
- Verarbeiten von Daten im Batch- und Streaming-Modus
- Unterstützung von Verarbeitungsaufgaben für Echtzeitanwendungen wie z.B. Joins von Streaming Daten
- Unterstützung im Bereich Reporting/OLAP, z.B. durch Filterung und Aggregation
- Unterstützung im Bereich Advanced Analytics, z.B. durch Training von Modellen

6.1.5. Lakehouse-Frameworks

Hinter den Open-Source-Projekten Delta Lake, Apache Hudi und Apache Iceberg stehen Frameworks, die für den Aufbau von Lakehouses als integrierte Datenplattformen konzipiert sind. Die Lakehouse-Frameworks sind selbst nicht als eigenständige Datenplattformen anzusehen, sondern müssen mit einer Hosting-Infrastruktur wie einem Cloud-Objektspeicher oder einer lokalen Speicherlösung kombiniert werden, um die Daten und Metadaten eines Lakehouses dauerhaft zu speichern [SGL+23]. Ein Lakehouse-Framework entspricht einem leichtgewichtigen Nachbau von Data-Warehouse-Verwaltungsfunktionen. Der Einsatz eines solchen Frameworks verhindert das unkontrollierte Verteilen von Daten eines zusammenhängenden Datensatzes über verschiedene Datendateien, wodurch eine konsistente Verarbeitung dieser Daten als Einheit erleichtert wird. Zur Durchsetzung der Data-Warehouse-Verwaltungsfunktionen auf dem zugrundeliegenden Speichersystem, wie beispielsweise den ACID-Eigenschaften bei Datenzugriffen oder dem Schema-Enforcement, greifen alle genannten Lakehouse-Frameworks auf einen vergleichbaren Ansatz zurück [AGXZ21]: Sie stützen sich auf technische Metadaten, die Informationen über verfügbare Tabellen, deren Struktur sowie deren Partitionierung bereitstellen. Außerdem beinhalten die Metadaten ein Protokoll, welches das Hinzufügen und Löschen von Datendateien nachverfolgt, um Isolierung und Atomizität bei Datenzugriffen zu gewährleisten [SGL+23]. Im Gegensatz zu einem Data Lake kann ein Lakehouse deshalb verschiedene Datendateien als einen einzigen Datensatz, nämlich als Tabelle, interpretieren. Dabei nutzen die drei Lakehouse-Frameworks unterschiedliche Tabellenformate und Zugriffsprotokolle, um die erforderlichen Metadaten zu organisieren. Aus diesem Grund werden sie häufig mit leistungsstarken Tabellenformaten für die Verwaltung großer Datenmengen gleichgesetzt. Zusätzlich zeichnen sich die Lakehouse-Frameworks durch ihre umfassende Integration mit etablierten Prozessierungsframeworks für Batch- und Streaming-Daten aus, wie z.B. Apache Spark und Apache Flink [SGL+23]. Mithilfe eines Prozessierungsframeworks können gleichzeitige Lese- oder Schreibzugriffe auf den Tabellen durchgeführt werden, um die Tabellendaten für verschiedene Analysen aufzubereiten oder über Abfragesprachen wie z.B. Structured Query Language (SQL) abzufragen. Um zu demonstrieren, dass der Prototyp unabhängig von einem spezifischen Lakehouse-Framework ist, berücksichtigt seine Implementierung zwei populäre Lakehouse-Frameworks: Delta

Lake und Apache Iceberg. Folglich kann er mit Tabellen arbeiten, die im Delta- oder im Iceberg-Format angelegt sind. Die nachfolgende Auflistung erläutert, welche Metadatenstrukturen die beiden Lakehouse-Frameworks vorsehen:

Delta Lake: Delta Lake wurde von Databricks entwickelt und ist inzwischen ein Open-Source Projekt. Dieses Lakehouse-Framework organisiert Metadaten in zwei verschiedenen Dateitypen: *Delta-Logs* und *Checkpoints*. Delta-Logs werden als JSON-Objekte im Ordner `_delta_log` im Unterverzeichnis jeder Tabelle angelegt und zeichnen Änderungen an Tabellen sequentiell auf. Um einen effizienten Zugriff auf diese Daten zu ermöglichen, werden die Delta-Logs regelmäßig zu Checkpoints komprimiert, für deren Ablage im Parquet-Format ein benutzerdefiniertes Verzeichnis festgelegt werden kann. Ein Checkpoint fasst alle Änderungen an einer Tabelle bis zu seinem Erstellungszeitpunkt zusammen, mit Ausnahme redundanter Operationen. Diese fließen nicht in einen Checkpoint ein und werden entfernt [YAH19].

Apache Iceberg: Apache Iceberg ist ein Open-Source-Projekt, das von Netflix entwickelt wurde. Das Lakehouse-Framework verfolgt Änderungen an einer Tabelle durch drei Metadatenschichten, die jeweils mit einem der drei folgenden Dateitypen verbunden sind: *Metadata Files*, *Manifest Lists* und *Manifest Files*. Der Zustand einer Tabelle lässt sich anhand ihres Metadata Files ableiten, welches unter anderem das Tabellenschema, die Partitionierungskonfiguration und Momentaufnahmen des Tabelleninhalts enthält. Bei jeder Zustandsänderung einer Tabelle wird ein neues Metadata File erstellt, um anschließend die alten Metadaten im Rahmen eines atomaren Austauschs zu ersetzen. Im Kontext von Apache Iceberg stellt ein Snapshot den Zustand einer Tabelle zu einem bestimmten Zeitpunkt dar und kann für den Zugriff auf den gesamten Satz von Datendateien einer Tabelle verwendet werden. Die Datendateien eines Snapshots werden durch eine oder mehrere Manifest Files verfolgt, welche die Partitionsdaten und Metriken jeder Datendatei einer Tabelle enthalten. Der Inhalt eines Snapshots entspricht der Vereinigung aller Manifest Files. Ein einzelnes Manifest File kann auf Datendateien verweisen, die lediglich eine Teilmenge einer Tabelle enthalten. Deshalb werden Manifest Files eines Snapshots in Manifest Lists zusammengeführt. Anhand dieser Listen kann festgestellt werden, welche Manifest Files für eine bestimmte Operation erforderlich sind, so dass unnötiges Lesen aus Manifest Files vermieden werden kann [Hug].

Trotz ihrer unterschiedlichen Metadatenstrukturen bieten Delta Lake und Apache Iceberg ähnliche Funktionen. In Tabelle 6.1 sind einige ausgewählte funktionale Merkmale der beiden Lakehouse-Frameworks aufgeführt. Die Tabelle zeigt, dass beide Frameworks ACID-Zugriffe, Time Travel und Schema-Evolution unterstützen, was eine Ähnlichkeit zu modernen Data Warehouses darstellt, wie in Abschnitt 2.1 beschrieben wird. Ein Unterschied zu diesen ist jedoch die fehlende Unterstützung für Primärschlüsseleinschränkungen und benutzerdefinierte Indizes. Im Vergleich zu Delta Lake bietet Apache Iceberg den Vorteil, nicht nur die Schema-Evolution, sondern auch die Partition-Evolution zu unterstützen. *Partition-Evolution* steht für die Möglichkeit, die Partitionierung von Iceberg-Tabellen zu aktualisieren, wobei historische Partitionsversionen aufbewahrt werden, um die Änderungen rückwirkend nachvollziehbar zu machen. Für Abfragen ergibt sich dadurch keine zusätzliche Komplexität, da die Partitionsversion nicht explizit angegeben werden muss. Eine Besonderheit von Delta Lake ist in der standardmäßigen Unterstützung von Integritätseinschränkungen in SQL-Statements zu sehen, wodurch eine automatische Validierung der Datenqualität erleichtert werden soll. Diese Einschränkungen werden auf Schemaebene festgelegt und automatisch geprüft, wenn neue Daten in eine Tabelle eingefügt werden. Es gibt derzeit zwei Arten von Integritätseinschränkungen:

Merkmal	Delta Lake 2.1.0	Apache Iceberg 1.0.0
ACID-Zugriffe	✓	✓
Time Travel	✓	✓
Schema Evolution	✓	✓
Partition Evolution	×	✓
Validierung der Datenqualität	✓	×
Primärschlüsseleinschränkung	×	×
Benutzerdefinierte Indexierung	×	×

✓erfüllt × nicht erfüllt

Tabelle 6.1.: Übersicht über beispielhaft ausgewählte funktionale Merkmale von Delta Lake 2.1.0 und Apache Iceberg 1.0.0.[6]

Das `NOT NULL` Constraint erzwingt, dass Werte in einer Spalte nicht NULL sein dürfen, während das `CHECK` Constraint die Spezifikation eines einfachen booleschen Ausdrucks erlaubt, welcher für neue Datensätze wahr sein muss, damit diese eingefügt werden können.

Aus Sicht des Prototyps sind alle drei Lakehouse-Frameworks gleichwertig, da sie ACID-Zugriffe auf Tabellen unterstützen und somit keine Anomalien bei gleichzeitigen Zugriffen auftreten. Dadurch kann der Prototyp Pipeline-Schritte einer oder mehrerer Daten-Pipelines unabhängig voneinander ausführen. Die Lakehouse-Frameworks definieren das Tabellenformat, in dem die durch das Prozessierungsframework Apache Spark verarbeiteten Streaming- oder Batch-Daten im lokalen Dateisystem abgelegt werden. Um die Kompatibilität des Prototyps mit den Lakehouse-Frameworks Delta Lake und Apache Iceberg zu gewährleisten, müssen die Formatoptionen `delta` und `iceberg` in der Spark-Konfiguration für das Lesen und Schreiben von Daten berücksichtigt werden. Die funktionalen Unterschiede zwischen den Lakehouse-Frameworks sollten sich in den verfügbaren Optionen des Frontends widerspiegeln. Ein Beispiel hierfür ist die Möglichkeit, für Delta-Tabellenspalten den `NOT NULL` Constraint zu definieren, während diese Möglichkeit für Iceberg-Tabellen nicht zur Verfügung stehen sollte. Wenn sich ein funktionaler Unterschied jedoch nur auf die Verarbeitung von Tabellendaten auswirkt, liegt es in der Verantwortung des Nutzers, die Eigenheiten eines Frameworks im benutzerdefinierten Code für Pipeline-Schritte zu berücksichtigen.

6.2. Implementierungsansatz

In diesem Abschnitt wird der Implementierungsansatz des Prototyps vorgestellt und dabei insbesondere auf dessen Bezug zu den in Abschnitt 5.5 vorgeschlagenen Konzepten zur Entwicklung eines Modellierungswerkzeugs eingegangen. Aufgrund des hohen Implementierungsaufwands war es im Rahmen dieser Masterarbeit nicht möglich, die Konzepte unverändert umzusetzen. Dennoch berücksichtigt der Prototyp die Grundideen der Meta Object Facility (MOF)-Hierarchie und der Model Driven Architecture (MDA), indem er Modelle verschiedener Abstraktionsebenen verwendet. Er hält sich an die Vorgaben des in Abschnitt 5.2 beschriebenen Metamodells, das für ihn ebenso die Rolle eines Computation Independent Models (CIM) übernimmt. Gemäß der

[6] https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison

MDA achtet der Prototyp auch auf eine strikte Trennung zwischen plattformunabhängigen und plattformspezifischen Aspekten einer Daten-Pipeline. Darüber hinaus verfolgt die Implementierung das Ziel, die im Abschnitt 5.5.1 genannten Anforderungen an ein Modellierungswerkzeug zu erfüllen. Zusammengefasst verlangen diese ein plattformunabhängiges Modellierungswerkzeug, über das Daten-Pipelines deklarativ definiert und ausgeführt werden können. Im Gegensatz zu den in Abschnitt 5.5 vorgeschlagenen Konzepten ermöglicht der Prototyp jedoch keine automatische Transformation eines Platform Independent Models (PIM) zu einem Platform Specific Model (PSM). Der Implementierung ist keine Spezifikation dieser Transformation auf MOF-Ebene M2 hinterlegt. Aus diesem Grund ist es nicht möglich, ein PSM mit plattformspezifischen Details zu verfeinern und daraus ausführbaren Pipeline-Code zu generieren. Wie der Prototyp hingegen aufgebaut ist und wie er Modelle für die Definition und Ausführung von Daten-Pipelines einsetzt, wird in den folgenden Unterabschnitten erläutert.

6.2.1. Abbildung des Modells für Daten-Pipelines auf ein JSON-Schema

Zur Abstraktion einer Daten-Pipeline bezieht sich der Prototyp auf das in Abschnitt 5.3 eingeführte verfeinerte Metamodell für Daten-Pipelines in Lakehouses. Dieses dient im Kontext der MDA als plattformunabhängiges Metamodell und sieht vor, dass eine Daten-Pipeline aus Loading-, Harmonization-, Distillation-, Delivery- und Exploration-Schritten besteht. Aufgrund ihres prototypischen Charakters beschränkt sich die Implementierung auf drei dieser fünf Pipeline-Schritt-Typen. Der Prototyp lässt lediglich die Definition von Loading-, Harmonization- und Distillation-Schritten zu. Daher ist ein über ihn aufgebautes und befülltes Lakehouse logisch in die drei Zonen Raw, Harmonized und Distilled untergliedert. Um dieses eingeschränkte Metamodell in eine für die Programmierung verwendbare Form zu überführen, wurde aus diesem manuell ein JSON-Schema abgeleitet. Im Prototyp bildet es die Grundlage für die Strukturierung des Frontends und des Backends. Ein einzelner Schritt innerhalb einer Daten-Pipeline ist im Schema durch die Eigenschaften `sources`, `processing` und `sinks` in drei Bestandteile unterteilt. Um eine ausführbare Daten-Pipeline in einer JSON-Instanz darzustellen, muss für jeden dieser Bestandteile ein Deployment-Abschnitt angegeben werden, der beispielsweise das Lakehouse-Framework oder das Prozessierungsframework spezifiziert. In Bezug auf den MDA-Kontext bündelt das JSON-Schema somit mehrere plattformspezifische Metamodelle. Entgegen der MDA resultieren diese nicht aus der Spezifikation einer komplexen Transformation des plattformunabhängigen Metamodells, sondern durch Anfügen ausgewählter Deployment-Abschnitte. In der nachfolgenden Liste werden die wesentlichen Vorgaben des JSON-Schemas zusammengefasst.

- Das JSON-Schema beschreibt genau eine Daten-Pipeline.
- Eine Daten-Pipeline kann aus beliebig vielen `loading-`, `harmonization-` und `distillation-`Schritten bestehen.
- Unabhängig von seinem Typ wird jeder Pipeline-Schritt durch die Eigenschaften `sources`, `processing` und `sinks` strukturiert.
- Über die Eigenschaften `sources` und `sinks` muss mindestens eine Datenquelle und eine Datensenke für jeden Pipeline-Schritt spezifiziert werden. Eine Obergrenze ist nicht hinterlegt. Dabei ist eine Datenquelle im Fall eines `loading`-Schritts eine externe Datenquelle. Ansonsten repräsentieren `sources` und `sinks` Tabellen eines Lakehouses.

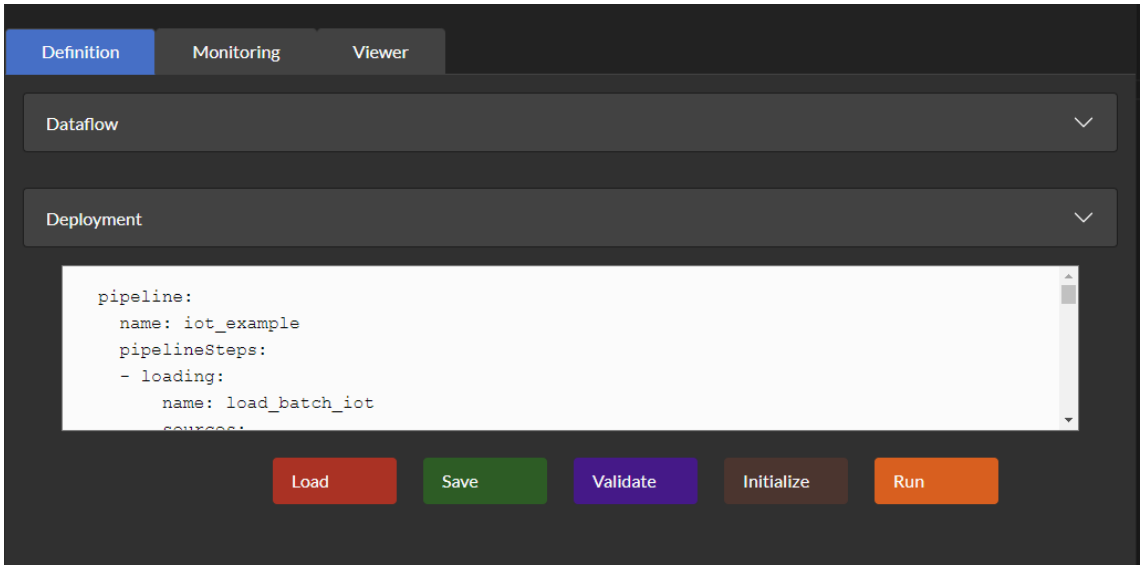

Abbildung 6.2.: Grafische Benutzeroberfläche des Prototyps.

- Für die Eigenschaften `sources` und `sinks` ist die Eigenschaft `name` obligatorisch. Aus den Namen von Datenquellen und Datensenken erschließt sich die Verknüpfung der Pipeline-Schritte.
- Die Eigenschaft `processing` beschreibt entweder eine Batch- oder Stream-Verarbeitung.
- Jede der Eigenschaften `sources`, `processing` und `sinks` besitzt die Eigenschaft `deployment`. Diese ist obligatorisch, wenn eine ausführbare Daten-Pipeline definiert werden soll.
- Als externe Datenquellen werden Kafka-Topics, CSV- oder JSON-Dateien unterstützt.
- Die Eigenschaft `deployment` von `sources` und `sinks` ist auf die Lakehouse-Frameworks Delta Lake und Apache Iceberg beschränkt.
- Die Eigenschaft `deployment` von `processing` ist auf das Prozessierungsframework Apache Spark beschränkt.

Das JSON-Schema wird einem Nutzer des Prototyps nicht explizit offengelegt. Vielmehr ermöglicht der Prototyp die Erstellung von Daten-Pipeline-Modellen auf der MOF-Ebene M1. Dafür kommt grundsätzlich ein textbasierter oder ein graphischer Ansatz in Frage. Um den Visualisierungsaufwand gering zu halten und die Erstellung von Modellen nicht an ein prototypisches Werkzeug zu binden, wurde in dieser Masterarbeit auf eine Benutzeroberfläche zur graphischen Modellierung verzichtet. Wie in Abbildung 6.2 zu sehen ist, bietet der Prototyp stattdessen ein Textfeld zur Definition von Daten-Pipeline-Modellen im YAML-Format[7]. Im Vergleich zu JSON zeichnet sich YAML durch seine gute Lesbarkeit und Kompaktheit aus. Diese Eigenschaften sollen Nutzern helfen, sich beim Erstellen und Ändern einer Daten-Pipeline im Text zu orientieren. Beide Formate sind in Bezug auf ihre Ausdrucksstärke ähnlich mächtig. Eine valide YAML-Definition entspricht im Prototyp genau einem Daten-Pipeline-Modell, dessen Struktur durch das zuvor eingeführte JSON-Schema validiert werden kann. Eine beispielhafte YAML-Definition kann in Anhang A.1 eingesehen werden. In

[7] https://yaml.org

dieser wird eine Daten-Pipeline definiert, die aus einem HARMONIZATION-Schritt besteht, der Daten einer Source-Tabelle im Streaming-Modus verarbeitet und die Verarbeitungsergebnisse in einer Sink-Tabelle ablegt. Der HARMONIZATION-Schritt wurde aus dem mehrschrittigen Daten-Pipeline-Modell entnommen, das in Abschnitt 5.4 beschrieben wurde.

6.2.2. Abbildung einer YAML-Definition auf Prefect-Elemente

Der Prototyp ist in der Lage, die validierte YAML-Definition einer Daten-Pipeline auf Prefect-Elemente abzubilden. Mithilfe einer Teilmenge dieser Prefect-Elemente können daraufhin die einzelnen Pipeline-Schritte ausgeführt werden. Die Zusammenhänge zwischen den beteiligten konzeptionellen und technischen Elementen sind in Abbildung 6.3 dargestellt. Wie vom JSON-Schema vorgegeben, beschreibt eine YAML-Definition genau eine Daten-Pipeline, die aus mehreren Pipeline-Schritten bestehen kann. Im Prototyp wird jeder Pipeline-Schritt durch einen Prefect Flow repräsentiert, welcher durch den Code einer Python-Funktion definiert wird. Im Rahmen eines Pipeline-Schritts fallen mehrere Aufgaben an, weshalb sich ein Flow in mehrere Prefect Tasks aufgliedert. Auch für diese Prefect-Elemente existiert auf Implementierungsebene jeweils eine Python-Funktion. Was im Code hingegen fehlt, ist die Entsprechung einer abgeschlossenen Daten-Pipeline. Der Prototyp kennt lediglich einzelne, unzusammenhängende Pipeline-Schritte, die sich anhand ihres Namens einer Daten-Pipeline zuordnen lassen. Idealerweise sollten Daten-Pipelines als Flows abgebildet werden, mit untergeordneten Flows für die einzelnen Pipeline-Schritte. Auch wenn Prefect die Verschachtelung von Flows ermöglicht, würde bei der Implementierung mit Apache Spark

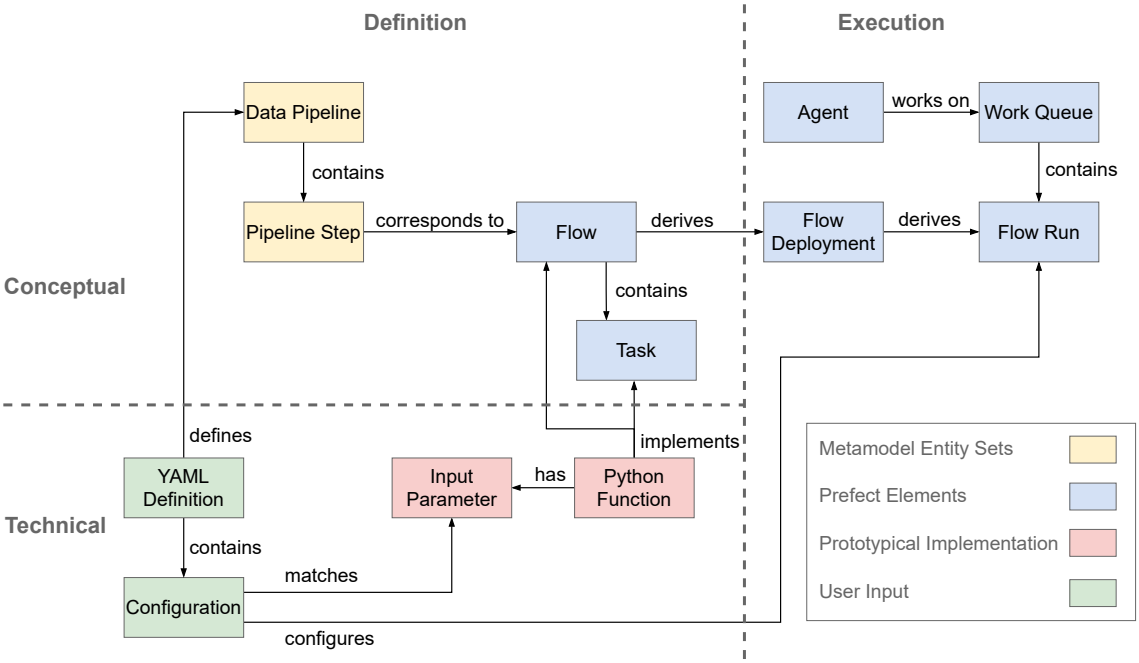

Abbildung 6.3.: Schematische Übersicht über Elemente, die bei der Abbildung der YAML-Definition einer Daten-Pipeline auf ausführbare Prefect-Elemente beteiligt sind. In der Darstellung wird zwischen Elementen auf konzeptioneller Ebene und auf Implementierungsebene unterschieden.

dadurch zusätzliche Komplexität entstehen. Die parallele Ausführung der Pipeline-Schritte müsste in diesem Fall von mehreren Spark-Workern in einem Spark-Cluster übernommen werden. Das *Fugue-Projekt*[8] demonstriert die Kompatibilität von Prefect mit Spark-Clustern. Dieses erleichtert es, in Python definierte Flows zur Beschleunigung oder Skalierung in einem Spark-Cluster auszuführen. Da der Einsatz eines Spark-Clusters den Rahmen dieser Masterarbeit überstiegen hätte, verwendet der Prototyp einzelne, unzusammenhängende Spark-Masters, die einfacher zu konfigurieren sind und jeweils für einen Flow zuständig sind.

Um die Ausführung eines Flows über die Prefect-API zu initiieren, nutzt der Prototyp sogenannte *Flow Deployments*, die in der Prefect-Umgebung hinterlegt sind. Ein Flow Deployment ist ein serverseitiges Konzept, das Informationen darüber enthält, wo der Code eines Flows gespeichert ist und wie dieser ausgeführt werden soll. Jedes Flow Deployment basiert auf genau einer Flow-Definition, was zu den drei Flow Deployments `Loading`, `Harmonization` und `Distillation` führt. Beim Ableiten eines neuen Flow Runs aus einem Flow Deployment werden benutzerdefinierte Parameterwerte aus der YAML-Definition extrahiert und an die zugrundeliegende Python-Funktion übergeben. Um ihre Ausführung einzuleiten, werden sie daraufhin in eine *Work Queue* einsortiert. Jede Work Queue kann einem oder mehreren *Agents* zugeordnet werden, die nach auszuführenden Flow Runs fragen, sobald sie verfügbar werden. Agents und Work Queues stellen lokal konfigurierbare Prefect-Elemente dar, die eine Brücke zwischen der lokalen Ausführungsumgebung und der Orchestrierungsumgebung bilden. Um die Arbeitsverteilung zu steuern und eine Einordnung der Flow Runs in eine Standard-Work-Queue zu vermeiden, fügt der Prototyp die Flow Runs gemäß ihrem Pipeline-Schritt-Typ zu den Work Queues `Loading`, `Harmonization` und `Distillation` hinzu.

6.3. Funktionsweise

Im vorherigen Abschnitt wurde der Implementierungsansatz skizziert, während dieser Abschnitt tiefer in die Logik des Prototyps eintaucht. Um ihn auf korrekte Weise für die Definition, Ausführung und Überwachung einer Daten-Pipeline, sowie zur Abfrage ihrer Verarbeitungsergebnisse einzusetzen, ist ein Verständnis seiner internen Arbeitsweise hilfreich. Die genannten Funktionalitäten dienen der Gliederung des Frontends in die Reiter *Definition*, *Monitoring* und *Viewer*, woran sich die folgenden Unterabschnitte orientieren. Jeder Unterabschnitt konzentriert sich auf einen Reiter und beleuchtet, wie Benutzereingaben im Webserver und Backend verarbeitet werden.

6.3.1. Definition und Ausführung

Eine der Intentionen hinter der Entwicklung des Prototyps besteht darin, die deklarative Definition und Ausführung von Daten-Pipelines zu unterstützen. Um diese Funktionalität des Prototyps zu verwenden, reicht es aus, wenn der Nutzer im Reiter *Definition* der Dash-Anwendung arbeitet, da dieser alle erforderlichen Steuerelemente zusammengefasst. Abbildung 6.4 veranschaulicht diesen Bestandteil des Frontends in einem größeren architektonischen Kontext und skizziert den Nachrichtenaustausch zwischen den Komponenten, der für die Definition und Ausführung

[8] https://github.com/fugue-project/prefect-fugue

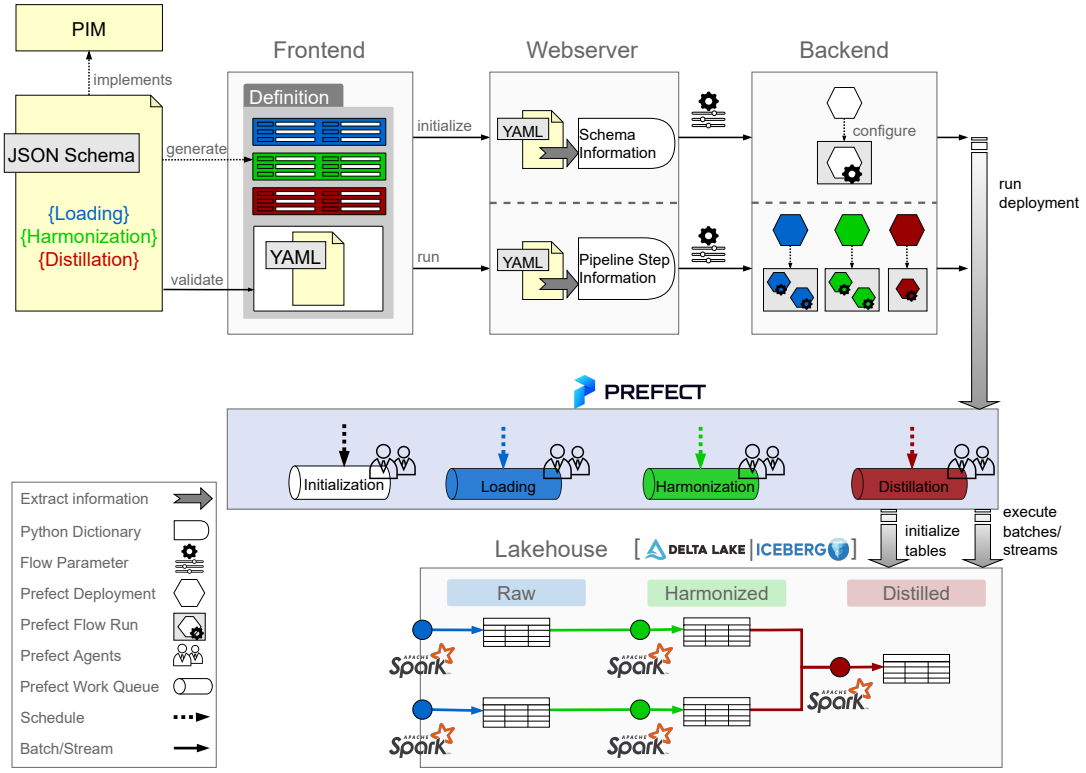

Abbildung 6.4.: Schematische Übersicht über die Architektur des Prototyps zur Definition und Ausführung einer Daten-Pipeline. Die Pipeline-Schritte einer YAML-Definition werden als Flow Runs in Work Queues der Prefect-Umgebung einsortiert und von Agents ausgeführt.

einer Daten-Pipeline benötigt wird. In den folgenden Unterabschnitten wird beschrieben, wie die YAML-Definition einer Daten-Pipeline vom Webserver genutzt wird, um passend konfigurierte Flow Deployments auszuführen.

Frontend

Der Reiter *Definition* stellt den primären Zugangspunkt für die Verwendung des Prototyps dar. Er bündelt eine Reihe von Dash-Elementen, welche es einem Nutzer ermöglichen, neue Daten-Pipelines im YAML-Format zu definieren und auszuführen. Die Anordnung dieser Elemente soll den Bezug auf das in Abschnitt 5.5 vorgestellte verfeinerte Metamodell für Daten-Pipelines herausstellen und den Erstellungsprozess einer Daten-Pipeline übersichtlich gestalten. Im oberen Bereich des Reiters grenzen die beiden Accordion-Elemente *Dataflow* und *Deployment* klar zwischen den Parametern ab, die den von der Daten-Pipeline zu realisierenden Datenfluss definieren, und jenen, die Plattformabhängigkeiten herstellen und die Ausführung der Daten-Pipeline auf ein bestimmtes Lakehouse-Framework und Prozessierungsframework einschränken. Das Layout beider Accordion-Elemente wird bei der Initialisierung der Dash-Anwendung automatisch aus dem in Abschnitt 6.2.1 einführten JSON-Schema generiert. Hierfür nutzt die Dash-Anwendung eine Reihe von aufeinander aufbauenden Python-Funktionen, die verschiedene Hierarchiestufen und Verschachtlungen des JSON-Schemas abbilden können. Ausgehend von der Definition der drei Pipeline-Schritte LOADING,

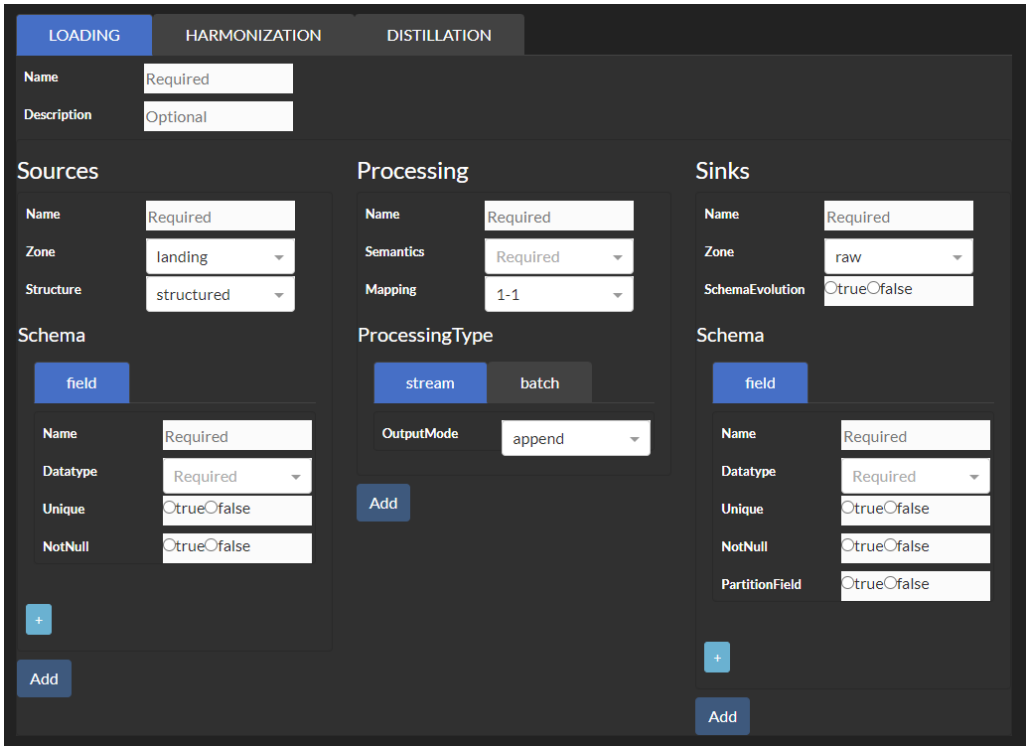

Abbildung 6.5.: Detailansicht des Reiters *Definition*, welche die Eingabefelder des Accordion-Elements *Dataflow* zur Definition eines LOADING-Schritts darstellt.

HARMONIZATION und DISTILLATION wandeln die Python-Funktionen sämtliche verschachtelte Objekte des JSON-Schemas in passende Dash-Elemente um, wobei Typinformationen aus dem JSON-Schema berücksichtigt werden. Beispielsweise bietet es sich an, eine Enumeration als Dropdown-Element in der Dash-Anwendung bereitzustellen. Eine Detailansicht der vollständig initialisierten Benutzeroberfläche ist in Abbildung 6.5 zu finden. Diese zeigt die Eingabefelder des Accordion-Elements *Dataflow*, die für die Definition eines LOADING-Schritts benötigt werden. Wie bereits in Abbildung 6.2 dargestellt wurde, schließt sich im unteren Bereich des Reiters *Definition* ein großes Textfeld an, das für den Aufbau der YAML-Definition vorgesehen ist.

Es ist anzumerken, dass die zahlreichen Eingabefelder des Reiters als zusätzliche Unterstützung für die Nutzer dienen. Sofern ein Nutzer bereits mit dem vom Prototyp verwendeten YAML-Format vertraut ist, kann er eine neue Daten-Pipeline direkt im zuvor genannten Textfeld über eine Texteingabe definieren. Andernfalls kann er zunächst über die Eingabefelder des Accordion-Elements *Dataflow* die Anzahl, den Typ und die grundlegenden Eigenschaften der Pipeline-Schritte einer Daten-Pipeline festlegen. Um einen einzelnen Pipeline-Schritt zu definieren, müssen die drei Bestandteile *Sources*, *Processing* und *Sinks* zusammengestellt werden. Wie in Abbildung 6.5 zu sehen ist, verfügen die Eingabefelder jedes Bestandteils über einen *Add*-Button, mit dem die Benutzereingaben an der richtigen Stelle in die YAML-Definition eingefügt werden können. Mehrere Source- und Sink-Tabellen können in einen einzelnen Pipeline-Schritt eingebunden werden, indem die Eingabefelder zur Definition der Tabellen wiederholt ausgefüllt und deren Inhalte über den *Add*-Button schrittweise an die YAML-Definition angehängt werden. Zusätzlich zur Definition der Bestandteile sind plattformabhängige Informationen zwingend erforderlich, damit die Daten-Pipeline

ausgeführt werden kann. Diese muss der Nutzer über die Eingabefelder des Accordion-Elements *Deployment* für die einzelnen Bestandteile aller Pipeline-Schritte ergänzen. Dabei besteht die Möglichkeit, die Verarbeitungslogik eines Pipeline-Schritts über benutzerdefinierten PySpark-Code festzulegen. Beim Schreiben des Codes muss der Nutzer darauf achten, Platzhalter der Form <source1>, <source2>, <sink1>, <sink2> statt der eigentlichen Tabellennamen zu verwenden. Dadurch kann der Code auf bestimmte Tabellen verweisen, ohne sich von deren Namen abhängig zu machen, was seine Austauschbarkeit und Wiederverwendbarkeit erhöht.

Während und nach diesem Erstellungsprozess kann ein Nutzer seine bisher erzeugte YAML-Definition durch Klicken auf einen *Validate*-Button validieren. Bei erfolgreicher Validierung ist sichergestellt, dass die YAML-Definition den Vorgaben des JSON-Schemas entspricht. Wenn die Validierung fehlschlägt, muss der Nutzer die YAML-Definition überarbeiten. Hilfestellung bei der Fehlersuche erhält er durch die angezeigten Validierungsfehler. Erst wenn alle Validierungsfehler behoben sind, kann der Nutzer die Daten-Pipeline zur Initialisierung und Ausführung an das Backend übergeben. Im Gegensatz zu den anderen Dash-Elementen des Reiters *Definition* werden die dafür zuständigen Callback-Funktionen nicht clientseitig durch die Dash-Anwendung ausgeführt. Stattdessen senden sie die YAML-Definition in einer HTTP-POST-Nachricht an den Quart-Webserver, der diese für die weitere Verarbeitung entgegennimmt.

Webserver

Bei welcher Webserver-Funktion eine eingehende HTTP-POST-Nachricht ankommt, hängt davon ab, ob sie über den *Initialize*- oder den *Run*-Button gesendet wurde. Obwohl sich die beiden Webserver-Funktionen in ihren Aufgaben unterscheiden, ist ihr erster Verarbeitungsschritt identisch. Zu Beginn extrahieren sie die YAML-Definition aus der Nachricht und instanziieren mit dieser als Eingabeparameter die Python-Klasse Pipeline. Das Hauptziel der Klasse Pipeline ist es, die verschachtelte Darstellungsform der Daten-Pipeline zu vereinfachen und so aufzubereiten, dass sie vom nachgelagerten Backend verarbeitet werden kann. Dies erfordert eine Durchmischung der plattformunabhängigen und plattformabhängigen Parameter, die für einen Nutzer im Frontend und im YAML-Format strikt getrennt gehalten werden. Zu diesem Zweck extrahiert eine Instanz der Klasse Pipeline alle für eine Ausführung relevanten Parameternamen und deren zugehörige Werte aus der YAML-Definition. Anschließend organisiert und speichert sie diese in verschiedenen Python-Dictionaries, strukturiert nach den unterschiedlichen Pipeline-Schritt-Typen und deren Bestandteilen. Neben der Extraktion erfolgt eine teilweise Umbenennung der Parameternamen, damit sie mit den von Apache Spark verwendeten Parameternamen übereinstimmen. Folglich ist die Abbildung einer Daten-Pipeline durch die Klasse Pipeline nicht mehr plattformunabhängig. Im Kontext des aus der MDA abgeleiteten Implementierungsansatzes repräsentiert eine Pipeline-Instanz das Platform Specific Model (PSM) einer Daten-Pipeline.

Eine vom Nutzer angeforderte Initialisierung führt zur Erstellung aller Tabellen im Lakehouse, auf welche die Daten-Pipeline im Laufe ihrer Ausführung zugreifen wird, sofern diese noch nicht existieren. Hierfür sind insbesondere Schemainformationen für die Source- und Sink-Tabellen notwendig. Die Pipeline-Instanz übergibt der Webserver-Funktion deshalb ein Python-Dictionary mit Spaltennamen und zugehörigen Datentypen, das daraufhin als Eingabeparameter für die Funktion `run_deployment` der Prefect-API verwendet wird. Dadurch wird im Backend ein neuer Flow Run des hinterlegten Flow Deployments Initialization gestartet.

Eine vergleichbare Vorgehensweise verbindet sich mit der Webserver-Funktion, die aufgerufen wird, wenn der Nutzer eine Daten-Pipeline über den *Run*-Button ausführen möchte. Diese iteriert über die einzelnen Pipeline-Schritte und ruft von der entsprechenden PIPELINE-Instanz alle für die Ausführung benötigten Informationen ab. Auf diese Weise wird für jeden in der YAML-Definition spezifizierten Pipeline-Schritt ein Python-Dictionary bereitgestellt, das alle erforderlichen Parameterwerte für dessen Konfiguration bündelt. Im Gegensatz zur Initialisierung werden dabei nicht nur Schemainformationen, sondern beispielsweise auch der Verarbeitungsmodus oder der benutzerdefinierte Code eines Pipelines-Schritts berücksichtigt. Das Dictionary wird anschließend als Eingabeparameter für die Funktion `run_deployment` verwendet, um abhängig vom Pipeline-Schritt-Typ eines der Flow Deployments LOADING, HARMONIZATION oder DISTILLATION auszuführen.

Backend

Das Auslösen eines Flow Deployments bewirkt die Planung und Ausführung eines neuen Flow Runs. Hierzu wird dieser in die entsprechende Work Queue eingereiht und bearbeitet, sobald der zuständige Agent verfügbar wird. In Abschnitt 6.2.2 wird erläutert, wie der Prototyp eine Daten-Pipeline auf Prefect-Elemente abbildet. Dieser Unterabschnitt legt seinen Fokus auf die logische Organisation und den Inhalt der Python-Skripte, die jeweils eine Flow-Definition beinhalten und als Flow Deployments bereitgestellt werden. Das Backend verwendet drei Flow-Definitionen, um den Ablauf der verschiedenen Pipeline-Schritt-Typen und des Initialisierungsschritts festzulegen. Für den LOADING-Schritt und den INITIALIZATION-Schritt wird jeweils eine separate Flow-Definition benötigt. Da sich die grundlegenden Abläufe eines HARMONIZATION- und DISTILLATION-Schrittes nicht voneinander unterscheiden, können Pipeline-Schritte beider Typen über die gleiche Flow-Definition abgebildet werden, welche in Listing 6.2 präsentiert wird. Allen Flow-Definitionen ist gemeinsam, dass sie die Art und Reihenfolge der auszuführenden Tasks bestimmen. Sie enthalten keine Datenverarbeitungsoperationen, sondern bestehen ausschließlich aus Funktionsaufrufen zu Python-Funktionen, die als Tasks annotiert sind. Um sicherzustellen, dass alle Flow-Definitionen auf eine zentrale Sammlung von Tasks zugreifen, sind die Tasks in einem Python-Skript gebündelt.

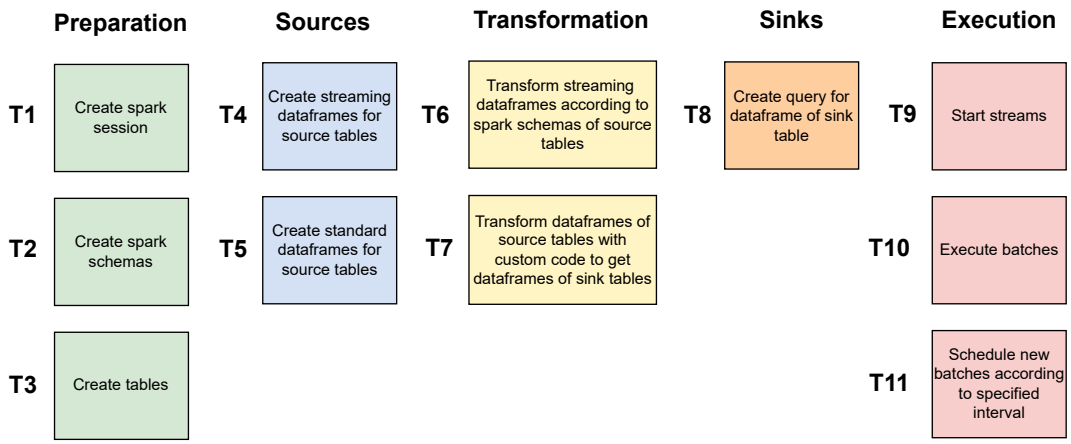

Abbildung 6.6.: Übersicht über die Prefect-Tasks, die im Backend des Prototyps mittels Python-Funktionen definiert werden. Die Gruppenzugehörigkeit eines Tasks ergibt sich aus dem Aufgabenbereich, für den er innerhalb eines Flows zuständig ist.

Jeder Task wird durch eine Python-Funktion repräsentiert, deren zugrundeliegende Logik in PySpark implementiert ist. PySpark ist die Python-API von Apache Spark, weshalb das Backend des Prototyps auf die Verwendung des Prozessierungsframeworks Apache Spark beschränkt ist. An dieser Stelle könnte man ansetzen, um den Prototyp mit anderen Prozessierungsframeworks zu erweitern. Abbildung 6.6 bietet einen Überblick über die verschiedenen Tasks, die nach ihrem Aufgabenbereich im Kontext eines Flows gruppiert sind. Die erste Gruppe namens *Preparation* enthält Tasks, die lediglich Vorbereitungen treffen und keine Datenverarbeitung durchführen. Über die beiden Gruppen *Sources* und *Sinks* werden Tasks zusammengefasst, die auf die Erstellung und Konfiguration von Spark-Dataframes ausgerichtet sind. Diese Dataframes bilden die Source- und Sink-Tabellen innerhalb der Spark-Umgebung ab. Falls erforderlich, können sowohl die Struktur als auch der Inhalt der Dataframes im Rahmen eines Pipeline-Schritts modifiziert werden, was durch Tasks der Gruppe *Transformation* umgesetzt wird. Schließlich sind Tasks der Gruppe *Execution* für den Start der Ausführung eines Pipeline-Schritts verantwortlich. Es ist zu beachten, dass die Pluralformen *Dataframes* und *Tables* zur Beschreibung der in Abbildung 6.6 dargestellten Tasks verwendet werden. Der Grund dafür ist, dass die entsprechende Python-Funktion über mehrere Source- oder Sink-Tabellen iteriert. Auf diese Weise kann ein Task auch mit Pipeline-Schritten umgehen, die eine n:m-Abbildung zwischen Source- und Sink-Tabellen vorsehen. In der nachfolgenden Liste werden die Aufgaben der einzelnen Tasks beschrieben.

Listing 6.1 Task-Definition von Task T4 in Python.

```python
@task
def create_streaming_source_dfs(cfg_sources, spark):
  global allowed_keys
  source_dfs = {}

  for idx, _ in enumerate(cfg_sources):
    table_name = cfg_sources[idx]["tableName"]
    table_format = cfg_sources[idx]["format"]
    data_path = ""

    if cfg_sources[idx]["zone"] == "landing":
      cfg_sources[idx]["kafka.bootstrap.servers"] = cfg_sources[idx]["broker"]
    else:
      data_path = cfg_sources[idx]["dataPath"]

    cfg = {k: v for k, v in cfg_sources[idx].items() if k in allowed_keys}
    df = spark.readStream.format(table_format).options(**cfg)

    if table_format == "delta":
      df = df.load(f"{data_path}//{table_name}")
    else:
      df = df.load()
    source_dfs[table_name] = df

  return source_dfs

```

T1: Um Apache Spark als Prozessierungsframework einsetzen zu können, ist eine Spark-Session notwendig. Dieser Task umfasst das Erstellen und Konfigurieren einer Spark-Session basierend auf den übergebenen Parameterwerten. Insbesondere werden dabei Pakete eingebunden, welche die Nutzung eines bestimmten Lakehouse-Frameworks ermöglichen.

T2: Dieser Task erhält Schemainformationen als Eingabe, bestehend aus Spaltennamen und den zugehörigen Datentypen, und wandelt sie in ein Spark-Schema um. Hierzu werden die Datentypen in Darstellungen übersetzt, die von Apache Spark verwendet werden können.

T3: Im Rahmen dieses Tasks wird eine Lakehouse-Tabelle gemäß den Spezifikationen eines in T2 generierten Spark-Schemas angelegt. Zu diesem Zweck erstellt der Task zunächst ein leeres Dataframe, um ihn anschließend im angeforderten Lakehouse-Format abzuspeichern. Dabei wird auch eine benutzerdefinierte Partitionierung der Tabelle berücksichtigt.

T4: Dieser Task arbeitet im Streaming-Modus und liest eingehende Streaming-Daten. Zur Darstellung des Datenstroms instanziiert der Task ein Streaming-Dataframe mithilfe von Funktionen aus Spark Structured Streaming. Um es zu konfigurieren, müssen unter anderem das Format des Lakehouse-Frameworks, sowie die Herkunft der zu lesenden Daten angegeben werden, z.B. in Form einer IP-Adresse und eines Topic-Namens bei Aufnahme von Daten aus Apache Kafka. Die Implementierung von T4 als Python-Funktion ist in Listing 6.1 dargestellt.

T5: Analog zu T4 befasst sich dieser Task ebenfalls mit der Erstellung eines Dataframes zur Anbindung von externen Daten. In diesem Fall ist der Task jedoch auf das Einlesen von Batch-Daten ausgelegt.

T6: Wenn ein Streaming-Dataframe einen Datenstrom aus Apache Kafka in das Lakehouse einspeist, enthält er zunächst die beiden Spalten *Key* und *Value* der Kafka-Nachrichten. Die Streaming-Daten verbergen sich in der *Value*-Spalte im JSON-Format, was ihre Weiterverarbeitung behindert. Dieser Task sorgt für die Abflachung der verschachtelten Daten, indem die Struktur des Dataframes an das von T2 bereitgestellte Spark-Schema angepasst wird.

T7: Dieser Task wird verwendet, um das Dataframe der unverarbeiteten Eingabedaten in ein Dataframe umzuwandeln, das die verarbeiteten Ausgabedaten darstellt. Um dies zu erreichen, wird benutzerdefinierter Code ausgeführt. Er legt fest, wie Daten der Source-Tabellen transformiert werden müssen, um Daten für die Sink-Tabellen zu erhalten. Der Task wendet diese Transformation auf das Dataframe der unverarbeiteten Eingabedaten an. Falls der Nutzer keinen Code spezifiziert hat, wird eine identische Abbildung zwischen Source- und Sink-Tabellen angenommen.

T8: Um verarbeitete Streaming-Daten zu schreiben ist im Kontext von Spark Structured Streaming eine Streaming-Abfrage erforderlich. In diesem Task wird eine Streaming-Abfrage erzeugt und entsprechend der benutzerdefinierten Parameterwerte konfiguriert, um die Speicherung der verarbeiteten Ausgabedaten zu steuern. Insbesondere werden dabei das Lakehouse-Format, der physische Ablageort der Sink-Tabellen, der Ausgabemodus und ein Trigger spezifiziert.

T9: Dieser Task dient dem Schreiben von verarbeiteten Streaming-Daten entsprechend der Konfiguration einer Streaming-Abfrage. Dabei wird die Stream-Verarbeitung im Hintergrund gestartet. Der Prozess bleibt so lange aktiv, bis die Streaming-Abfrage explizit gestoppt wird.

T10: Analog zu T9 befasst sich dieser Task mit dem Schreiben von verarbeiteten Batch-Daten im angeforderten Lakehouse-Format. Abgesehen von einem Trigger verwendet dieser Task die gleichen Parameterwerte wie T8.

T11: Um dem Orchestrierungsframework Prefect den Zeitpunkt der nächsten Ausführung einer Batch-Verarbeitung bekanntzugeben, muss ein neuer Flow Run geplant werden. Dieser Task verwendet ein benutzerdefiniertes Intervall, um die Zeitabstände zu bestimmen, in denen ein Pipeline-Schritt im Batch-Modus wiederholt werden soll.

Die Tasks fungieren als einzelne Bausteine verschiedener Flows. Wie in Abbildung 6.7 veranschaulicht, werden sie kombiniert, um die vom Backend verwendeten Flows zu erhalten. Die Abbildung visualisiert die Abfolge der Tasks in Form von Workflow-Diagrammen, was dem in Abschnitt 6.1.3 beschriebenen Verständnis von Flows aus Sicht von Prefect entspricht. Während die Flow-Definition des INITIALIZATION-Schritts nicht zwischen Pipeline-Schritten im Batch- oder Streaming-Modus unterscheiden muss, erfordert die Verarbeitung von Batch- und Streaming-Daten in den beiden anderen Flow-Definitionen größtenteils unterschiedliche Tasks. Dieser Umstand ist darauf zurückzuführen, dass die verwendeten Dataframes Instanzen unterschiedlicher Klassen

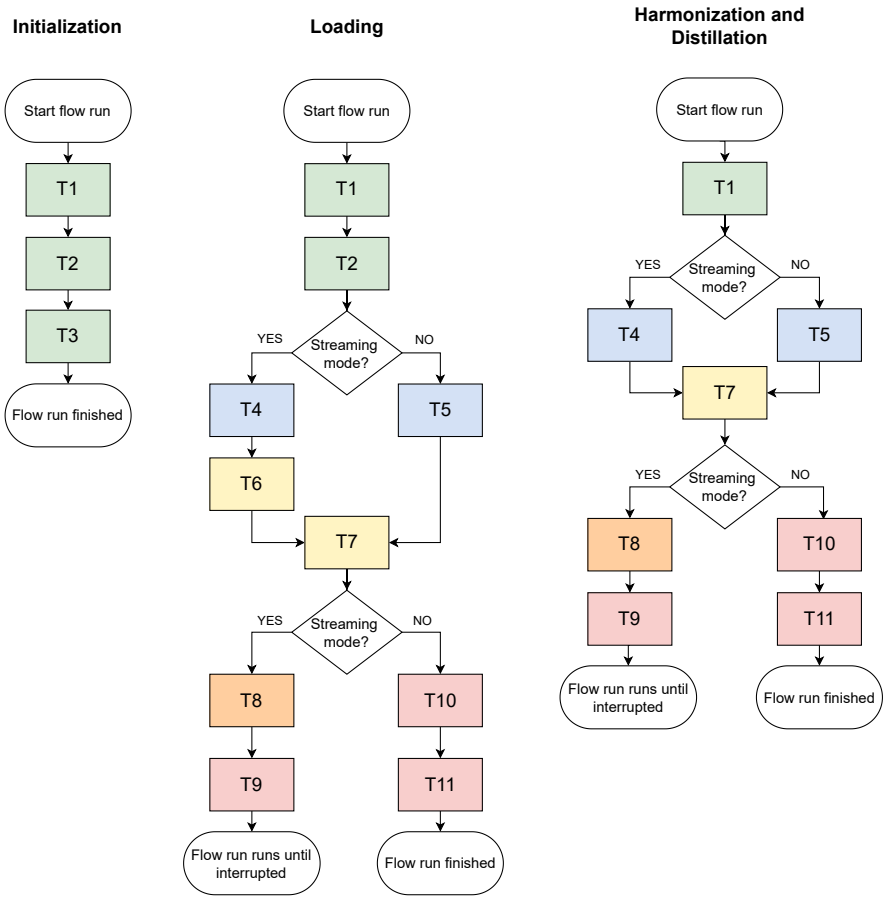

Abbildung 6.7.: Übersicht über die Workflow-Diagramme der vom Backend bereitgestellten Flow-Definitionen. Alle Flow-Definitionen greifen zur Erfüllung ihrer Aufgabe auf die gleiche Sammlung von Tasks zurück.

sind. Zur Erstellung von Streaming-Dataframes wird die `DataStreamReader`-Schnittstelle verwendet, welche Spark Structured Streaming zugeordnet ist. Im Gegensatz dazu wird für die Verarbeitung von Batch-Daten die `DataFrameReader`-Schnittstelle der Python-API von Apache Spark genutzt. Im Vergleich zur Flow-Definition für Harmonization- oder Distillation-Schritte umfasst die Flow-Definition eines Loading-Schritts die beiden zusätzlichen Aufgaben T2 und T6. Das von T2 erzeugte Spark-Schema ist erforderlich, um die Struktur externer Datenquellen bei der Aufnahme von Daten in das Lakehouse zu berücksichtigen. Im Falle der Aufnahme von Streaming-Daten aus Apache Kafka wird über T6 zudem eine Transformation zur Abflachung der Daten durchgeführt, die im Streaming-Dataframe initial im JSON-Format vorliegen. Der Grund für die Zusammenlegung von Harmonization- und Distillation-Schritten ist darin zu sehen, dass Pipeline-Schritte dieser Typen zum Lesen und Schreiben von Daten ausschließlich auf Tabellen des Lakehouses zugreifen. Listing 6.2 gibt einen Einblick, wie die Flow-Definition der beiden Pipeline-Schritt-Typen auf der Implementierungsebene umgesetzt wird. Obwohl der Unterschied zwischen den beiden Typen auf dieser Ebene nicht offensichtlich ist, sind entsprechend des verfeinerten Metamodells beide Typen erforderlich, da ihre Ausgabe in Sink-Tabellen unterschiedlicher Zonen erfolgt.

Listing 6.2 Flow-Definition von Harmonization- und Distillation-Schritten.

```python
import calendar
import time
import tasks
from prefect import flow
from pprint import pprint

@flow
def flow_harmonization(cfg: dict):

  cfg_sources = cfg["sources"]
  cfg_processing = cfg["processing"]
  cfg_sinks = cfg["sinks"]

  timestamp = calendar.timegm(time.gmtime())
  spark_app_name = f"{timestamp}-harmonization"
  table_format = cfg_sinks[0]["format"]

  spark = tasks.create_spark_session(table_format, spark_app_name, cfg_processing)

  if cfg_processing["processingType"] == "stream":
    source_dfs = tasks.create_streaming_source_dfs(cfg_sources, spark)
    sink_dfs = tasks.process_source_dfs(cfg_sources, cfg_sinks, cfg_processing, source_dfs)
    sink_queries = tasks.create_sink_queries(cfg_processing, cfg_sinks, sink_dfs)
    tasks.start_streams(cfg_processing, cfg_sinks, sink_queries, spark)

  elif cfg_processing["processingType"] == "batch":
    source_dfs = tasks.create_batch_source_dfs(cfg_sources, spark)
    sink_dfs = tasks.process_source_dfs(cfg_sources, cfg_sinks, cfg_processing, source_dfs)
    tasks.start_batches(cfg_processing, cfg_sinks, sink_dfs)

```

6.3.2. Monitoring

Sobald ein Nutzer die YAML-Definition einer Daten-Pipeline an den Webserver übergeben hat, sind keine manuellen Schritte mehr erforderlich, damit diese ausgeführt wird. Allerdings können Daten-Pipelines oft über längere Zeiträume laufen, in denen es zu einer Reihe von Problemen sowohl mit den Daten als auch mit der Verarbeitungsinfrastruktur kommen kann. Um eine echtzeitnahe Beurteilung ihres Ausführungszustandes zu erlauben und eine schnelle Fehlererkennung und -behebung zu fördern, verfügt der Prototyp über einfache Funktionen zur Überwachung von Daten-Pipelines. Insbesondere ist es möglich, den Datenflussgraphen aller bisher ausgeführten Daten-Pipelines zu visualisieren. Abbildung 6.8 gibt einen Überblick über die beteiligten Komponenten und ihre Interaktionen. Aus der Abbildung geht hervor, dass der Webserver die zur Erstellung eines Datenflussgraphen benötigten Informationen von der Prefect-API abfragt. Diese bietet umfangreiche Metainformationen zu einzelnen Flow Runs. In den folgenden Unterabschnitten wird beleuchtet, wie die Visualisierung in der Dash-Anwendung generiert wird und welche Informationen sie liefert.

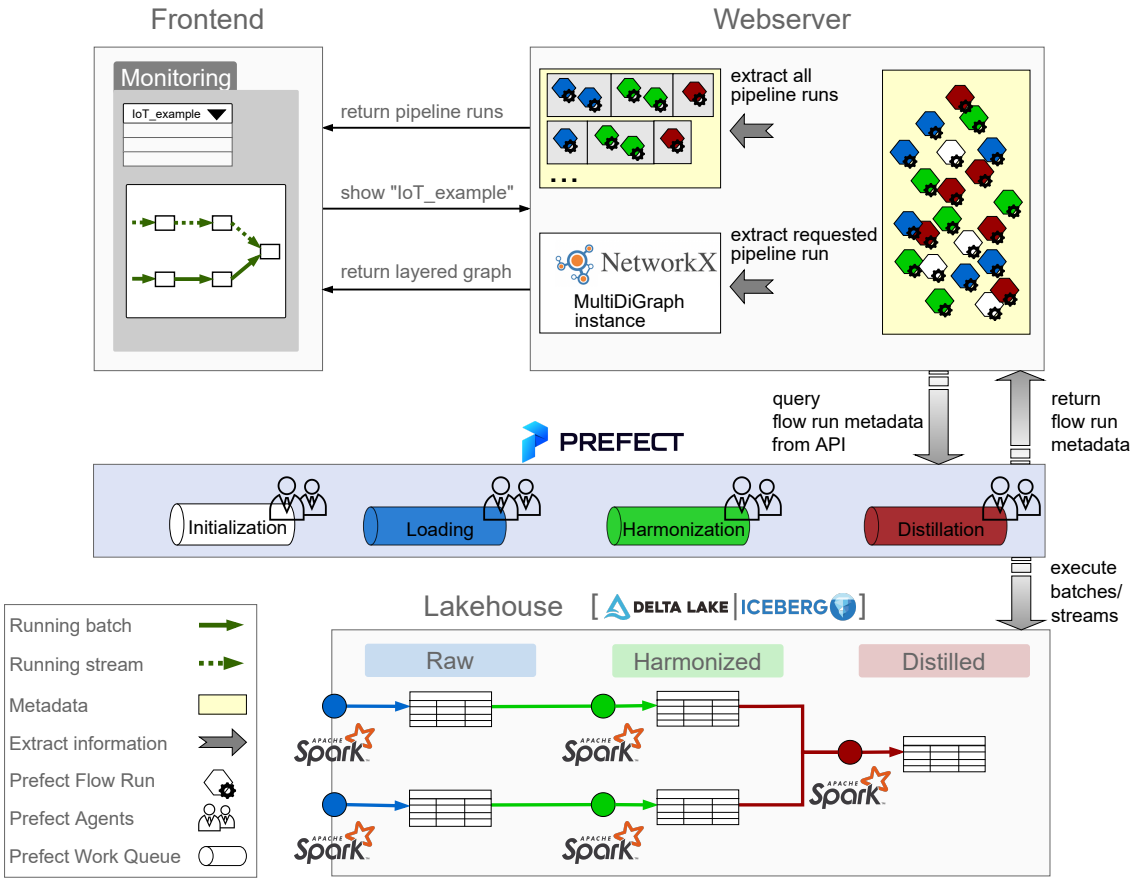

Abbildung 6.8.: Schematische Übersicht über die Architektur des Prototyps zur Überwachung einer Daten-Pipeline. Die von der Prefect-API abgefragten Metadaten eines Flow Runs werden für die Visualisierung des Datenflussgraphen im Frontend genutzt.

Frontend

Um den Ausführungszustand einer Daten-Pipeline zu beurteilen, können Nutzer den Reiter *Monitoring* in der Dash-Anwendung aufrufen. Dessen zentrales Element bildet eine *Cytoscape*-Komponente[9] der Python-Bibliothek Dash, die zur Visualisierung individuell anpassbarer und interaktiver Graphen dient. Im Prototyp wird die Cytoscape-Komponente eingesetzt, um den Datenflussgraphen von Daten-Pipelines darzustellen, die ein Nutzer zuvor definiert und ausgeführt hat. Abbildung 6.9 zeigt den Reiter *Monitoring*, dessen Cytoscape-Komponente den Datenflussgraphen einer laufenden Daten-Pipeline visualisiert. Diese Daten-Pipeline verbindet IoT-Sensordaten mit Gerätedaten, wie in Abschnitt 5.4 beschrieben wurde. In einem solchen Datenflussgraphen repräsentieren Knoten die Source- und Sink-Tabellen einer Daten-Pipeline, während Kanten den Pipeline-Schritten entsprechen, die eine oder mehrere Source-Tabellen mit einer oder mehreren Sink-Tabellen verbinden. Das Erscheinungsbild des Graphen wird durch ein hinterlegtes Stylesheet gesteuert. Dieses definiert, dass Pipeline-Schritte im Streaming-Modus als gestrichelte Linien zwischen Knoten dargestellt werden. Hingegen werden für Pipeline-Schritte im Batch-Modus durchgezogene Linien vorgeschrieben. Des Weiteren führt das Stylesheet eine Farbkodierung der Kanten ein, die vom Ausführungszustand der zugehörigen Flow Runs abhängt und eine farbliche Unterscheidung zwischen den nachfolgend aufgelisteten Ausführungszuständen ermöglicht. Deren Beschreibungen wurden aus der Dokumentation von Prefect übernommen.[10]

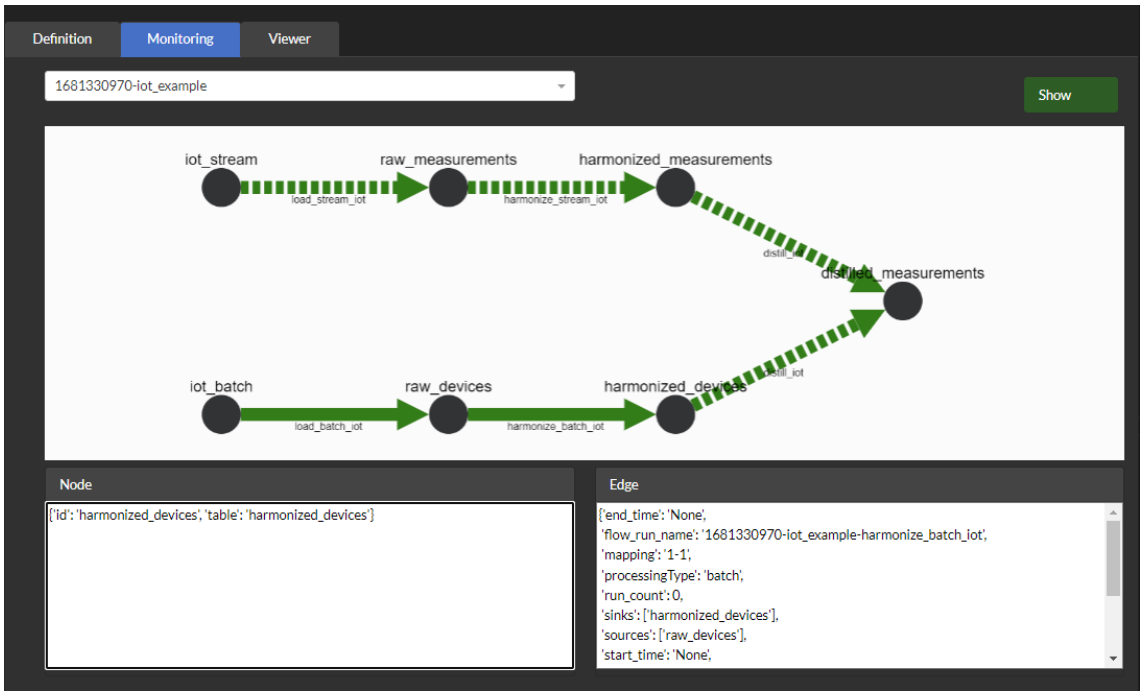

Abbildung 6.9.: Ansicht des Reiters *Monitoring*. Der dargestellte Datenflussgraph zeigt eine Daten-Pipeline, die gerade ausgeführt wird und IoT-Sensordaten mit Gerätedaten anreichert. Diese Daten-Pipeline wurde zuvor in Abschnitt 5.4 vorgestellt.

[9] https://dash.plotly.com/cytoscape
[10] https://docs.prefect.io/latest/concepts/states

Scheduled (blau): Der Flow Run wird zu einem bestimmten Zeitpunkt in der Zukunft beginnen.

Pending (grau): Der Flow Run wurde zur Ausführung freigegeben, wartet aber noch auf die Erfüllung notwendiger Voraussetzungen (z.B. verfügbarer Agent).

Running (grün): Der Flow-Run-Code wird gerade ausgeführt.

Completed (schwarz): Der Flow Run wurde erfolgreich beendet.

Failed (orange): Der Flow Run wurde wegen eines Problems im Flow-Run-Code nicht beendet.

Crashed (rot): Der Flow Run wurde wegen eines Problems in der Infrastruktur nicht beendet.

Der Ausführungszustand eines Pipeline-Schritts kann sich dynamisch ändern, weshalb ein Interval-Element von Python Dash in die Cytoscape-Komponente eingebettet wurde. Dieses sorgt dafür, dass die Anzeige der Cytoscape-Komponente in definierten Zeitabständen aktualisiert wird. Zusätzlich kann ein Nutzer durch Klicken auf einen Knoten oder eine Kante der Cytoscape-Komponente weitere Details über die dargestellten Tabellen und Pipeline-Schritte abrufen. Diese Informationen werden als Textausgabe in zwei Textfeldern im unteren Bereich des Reiters bereitgestellt. Um die Cytoscape-Komponente zu initialisieren, muss ein Nutzer über die Auswahlliste eines Dropdown-Elements die zu visualisierende Daten-Pipeline bestimmen. Zur Auswahl stehen alle Daten-Pipelines, deren Ausführung er im Vorfeld über den *Run*-Button des Reiters *Definition* angefordert hat. Sobald der Nutzer eine Daten-Pipeline festgelegt hat, kann er den Datenflussgraphen durch Klicken auf den Button *Show* anzeigen lassen. Sowohl die Ermittlung der im Dropdown-Element verfügbaren Daten-Pipelines als auch die Generierung und fortlaufende Aktualisierung des Datenflussgraphen erfordern die Kommunikation der Dash-Anwendung mit dem Webserver.

Webserver

Die HTTP-POST-Nachrichten der Dash-Anwendung werden zu Webserver-Funkionen weitergeleitet, die zur Erledigung ihrer Aufgabe einen *Prefect-Client*[11] einsetzen. Ein Prefect-Client dient als Schnittstelle zwischen Anwendungen und der Prefect-Orchestrierungsengine. Er kann auf Metainformationen aller Flow Runs zugreifen, unabhängig von deren Ausführungszustand. Somit bleiben in der Prefect-Umgebung auch historische Flow Runs erhalten. Eine Aufgabe des Prefect-Clients besteht darin, die IDs aller Daten-Pipelines für die Auswahlliste des Dropdown-Elements bereitzustellen. Dazu fragt er alle Flow Runs ab, die der Prefect-Orchestrierungsengine aktuell bekannt sind. Da ein Flow Run einem Pipeline-Schritt entspricht, der einen einzelnen Bestandteil in einer möglicherweise mehrschrittigen Daten-Pipeline darstellt, kann das Ergebnis der Abfrage nicht unmittelbar verwendet werden. Um einem Nutzer eine Übersicht über abgeschlossene Daten-Pipelines zu bieten, identifiziert die zuständige Webserver-Funktion die Flow Runs, die derselben Pipeline angehören. Diese Identifikation erfolgt anhand eines Zeitstempels, der jedem Flow Run bei seiner Erzeugung aus einem Flow Deployment zugewiesen wird. Mit dieser Methode können auch mehrere Ausführungen derselben Daten-Pipeline unterschieden und zeitlich eingeordnet werden. Als Resultat der serverseitigen Verarbeitung gehen eindeutige Pipeline-IDs hervor, die aus dem Zeitstempel und dem Namen der Daten-Pipeline bestehen.

[11] https://docs.prefect.io/latest/concepts/tasks/#python-client

Um den Datenflussgraphen zu erstellen, greift die zuständige Webserver-Funktion auf die im Dropdown-Element ausgewählte Pipeline-ID zurück. Dadurch kann sie mithilfe eines Prefect-Clients alle relevanten Metainformationen zu den beteiligten Flow Runs abfragen. Neben dem Ausführungszustand der Pipeline-Schritte umfassen diese Metainformationen auch die Start- und Endzeitpunkte ihrer Ausführung, sowie alle Parameterwerte, über die bei ihrer Erzeugung das zugrundeliegende Flow Deployment konfiguriert wurde. Auf Basis dieser Metainformationen generiert die Webserver-Funktion eine Instanz der NetworkX-Klasse MultiDiGraph, um den Datenflussgraphen einer Daten-Pipeline abzubilden. NetworkX[12] ist eine Python-Bibliothek, die sich auf die Erstellung und Bearbeitung komplexer Netzwerke spezialisiert hat. Hierfür bietet sie eine Reihe von Werkzeugen, einschließlich Algorithmen zur Berechnung von Netzwerkeigenschaften und Graph-Layouts. Bei Verwendung einer MultiDiGraph-Instanz sind Schleifen und mehrere gerichtete Kanten zwischen zwei Knoten erlaubt. Dies macht die MultiDiGraph-Klasse zu einer passenden Wahl, um beliebige über den Prototyp definierte Daten-Pipelines darstellen zu können.

Die Konstruktion des Graphen erfolgt in zwei separaten Schritten. Im ersten Schritt werden die Abfrageergebnisse des Prefect-Clients genutzt, um die Source- und Sink-Tabellen als Knoten in den noch leeren Graphen einzufügen. Dabei werden die Knoten mit verschiedenen Attributen versehen, wie beispielsweise dem Namen der entsprechenden Tabelle. Außerdem wird jeder Knoten einem Layer zugeordnet, der sich aus der Zonenzuordnung einer Tabelle ergibt. Im zweiten Schritt werden Kanten zwischen den Knoten der Source- und Sink-Tabellen erstellt. Je nach Art der Abbildung, die ein Pipeline-Schritt zwischen den Source- und Sink-Tabellen realisiert, sind hierbei zwei verschiedene Fälle zu unterscheiden. Bei 1:1-, 1:n- und n:1-Abbildungen wird keine zusätzliche Information zur Erzeugung der Kanten zwischen Source- und Sink-Tabellen benötigt, da angenommen wird, dass die Daten jeder Source-Tabelle nach ihrer Verarbeitung in allen Sink-Tabellen abgelegt werden. Hingegen muss der Nutzer bei einer n:m-Abbildung eine *Lineage-Information* in der YAML-Definition des Pipeline-Schritts angeben. Ohne diese wäre unklar, aus welchen Source-Tabellen die verarbeiteten Daten einer Sink-Tabelle stammen, weil dem benutzerdefinierten Code in dieser Hinsicht keine Einschränkungen auferlegt sind. Nachdem die Knoten und Kanten des Graphen erzeugt wurden, wird eine NetworkX-Funktion genutzt, um das Layout des Graphen in eine mehrschichtige Darstellung zu konvertieren. Dadurch werden die Knoten entlang gerader Linien positioniert, wobei Knoten, die zuvor demselben Layer zugeordnet wurden, auf derselben Linie liegen. Diese Darstellung des Datenflussgraphen bildet das Ergebnis der serverseitigen Verarbeitung und wird an die Dash-Anwendung weitergeleitet.

6.3.3. Abfrage

Der Prototyp legt die Daten eines Lakehouses in Form von Parquet-Dateien im lokalen Dateisystem ab. Da diese Dateien binäre Daten enthalten, ist es für Nutzer ohne Werkzeugunterstützung schwer festzustellen, ob die Verarbeitung durch eine Daten-Pipeline wie vorgesehen erfolgt. Deshalb bietet der Prototyp die Möglichkeit, den Inhalt einer Tabelle in einem lesbaren Format anzuzeigen. Abbildung 6.10 zeigt die daran beteiligten Komponenten und wie sich ihre Zusammenarbeit bei der Abfrage von Verarbeitungsergebnissen gestaltet. Im Gegensatz zu den zuvor vorgestellten Funktionalitäten des Prototyps ist Prefect hierbei nicht involviert, weil die Abfrage der in verschiedenen

[12] https://networkx.org/documentation/stable/index.html

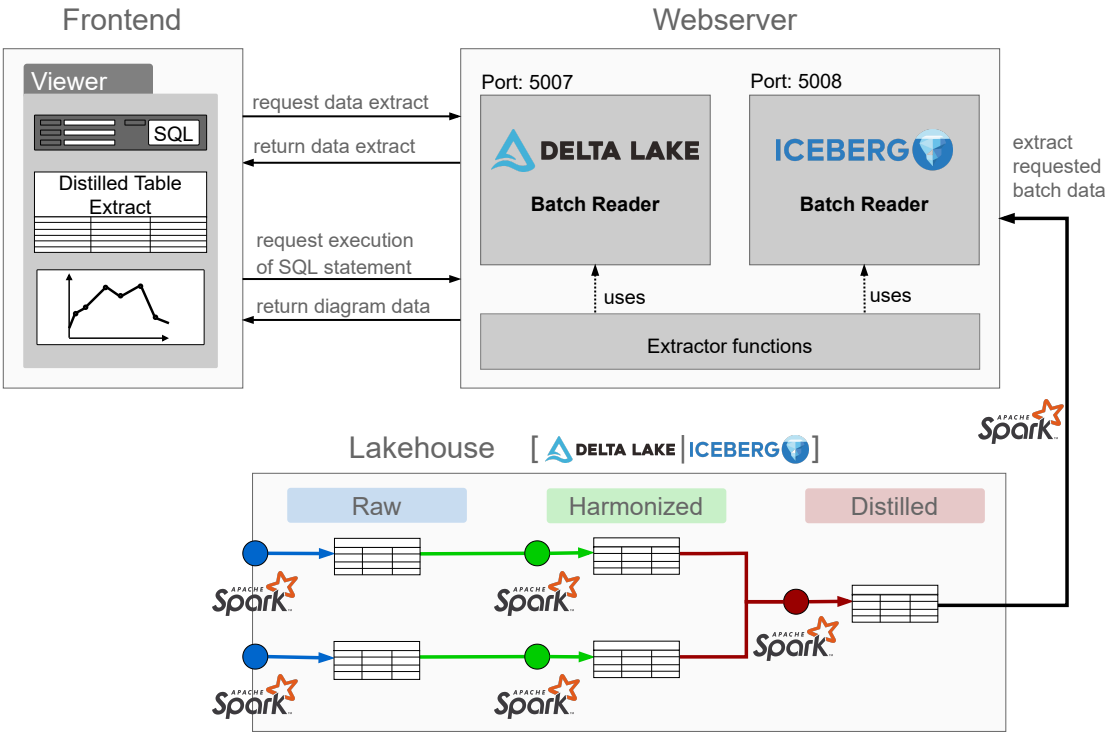

Abbildung 6.10.: Schematische Übersicht über die Architektur des Prototyps zur Abfrage von Verarbeitungsergebnissen. Der Webserver hält entsprechend konfigurierte Spark-Sessions vor, um auf Tabellen in verschiedenen Lakehouse-Formaten zuzugreifen.

Lakehouse-Formaten abgelegten Daten vollständig vom Webserver übernommen werden kann. In den nachfolgenden Abschnitten wird erläutert, welche Eingabe von einem Nutzer erwartet wird und wie der für diese Aufgabe verantwortliche Teil des Webservers implementiert ist.

Frontend

Die Dash-Anwendung stellt einem Nutzer im Reiter *Viewer* zwei unterschiedliche Methoden zur Verfügung, um Daten einer Tabelle abzufragen und die Ergebnisse der Abfrage zu visualisieren. Für beide Abfragevarianten ist es erforderlich, dass der Nutzer zunächst den Verzeichnispfad angibt, in dem sich die Parquet-Dateien einer Lakehouse-Tabelle befinden. Der Prototyp legt Tabellen auf der Grundlage ihres Lakehouse-Formats und ihrer Zonenzugehörigkeit ab, so dass eine Tabelle eindeutig identifiziert werden kann, indem ein Wurzelpfad, ein Lakehouse-Framework, eine Zone und ein Name kombiniert werden. Zusätzlich zu diesen Angaben muss der Nutzer eine Begrenzung für die Anzahl der Zeilen festlegen, die von der Abfrage zurückgegeben werden sollen. Sobald alle erforderlichen Angaben über Eingabefelder spezifiziert sind, kann der Nutzer mithilfe der ersten Abfragevariante einen Datenauszug der Tabelle anfordern. Beim Klicken auf den *Load*-Button wird eine HTTP-POST-Nachricht an den Webserver gesendet, der die angeforderten Daten abruft und

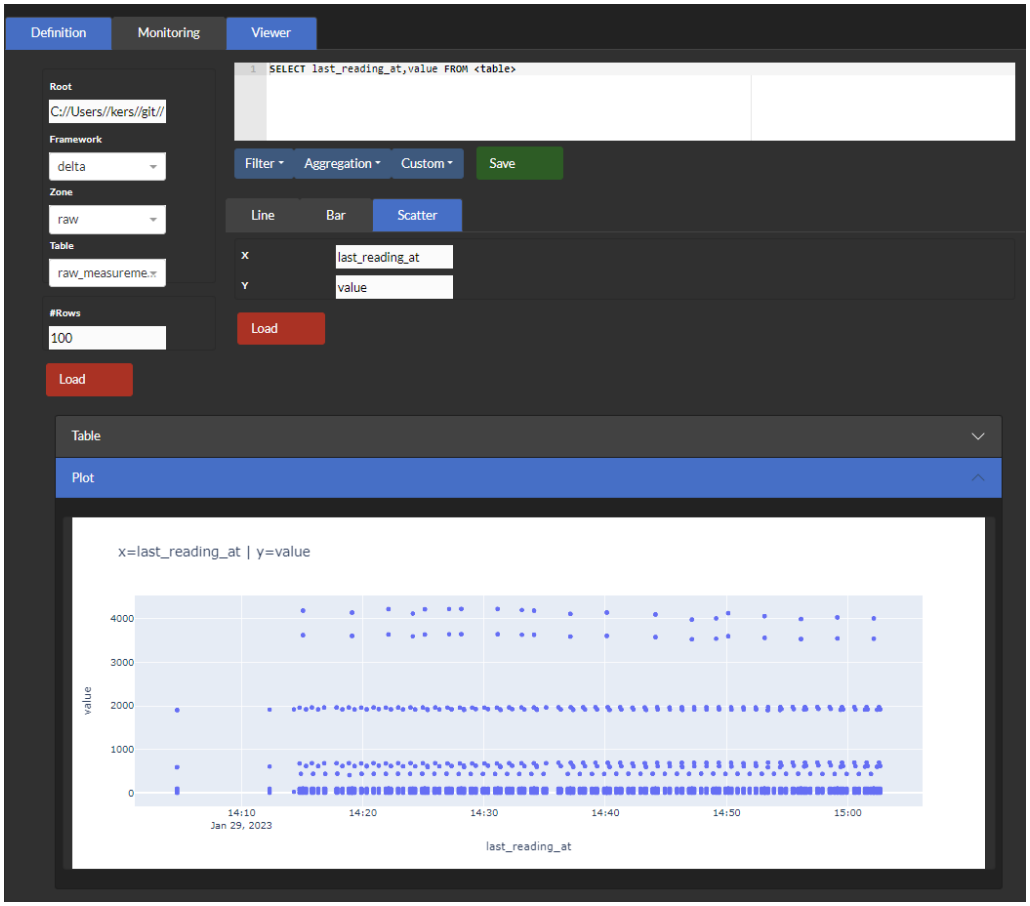

Abbildung 6.11.: Ansicht des Reiters *Viewer*. Das dargestellte Streudiagramm basiert auf den Rohdaten, die von den IoT-Sensoren der beispielhaften Daten-Pipeline aus Abschnitt 5.4 erfasst wurden.

an die Dash-Anwendung zurücksendet. Anschließend werden die Daten mithilfe der interaktiven *Datatable*-Komponente[13] visualisiert. Diese Komponente bietet nicht nur eine Anzeige, sondern auch einfache Sortier- und Filterfunktionen zur Erkundung des Datenauszugs.

Wie in Abbildung 6.11 dargestellt, bietet die zweite Abfragevariante im Reiter *Viewer* die Möglichkeit, ein Diagramm aus den Daten zweier Spalten einer Tabelle anzuzeigen. In der Abbildung ist ein Streudiagramm zu sehen, das aus unverarbeiteten IoT-Sensordaten der beispielhaften Daten-Pipeline aus Abschnitt 5.4 erzeugt wurde. Die Datenpunkte des Diagramms stammen von IoT-Sensoren mehrerer IoT-Sensorplattformen und basieren auf den Werten der Spalten `last_reading_at` und `value`. Für ein solches Diagramm muss der Nutzer eine Spark-SQL-Anfrage definieren, welche auf die Tabelle verweist, die durch die zuvor beschriebenen Benutzereingaben identifiziert wird. Die Spark-SQL-Anfrage stellt den Bezug zu einer Tabelle über den Platzhalter *<table>* her, anstatt den Speicherpfad und den Namen der Tabelle vorauszusetzen. Diese Vorgehensweise vereinfacht nicht nur die Erstellung der Anfrage, sondern auch eine Wiederverwendung derselben Anfrage für verschiedene Tabellen. Eine weitere Voraussetzung ist, dass die mithilfe der Anfrage identifizierten

[13] https://dash.plotly.com/datatable

Daten die für das Diagramm benötigten Spalten enthalten. Wenn eine solche Anfrage fertig erstellt ist, kann sie vom Nutzer zur späteren Wiederverwendung in einem lokalen Ordner gespeichert werden. Darüber hinaus muss der Nutzer für diese Abfragevariante die gewünschte Diagrammart festlegen. Er hat die Wahl zwischen Linien-, Balken oder Streudiagrammen. Für alle Diagrammarten muss zudem angegeben werden, welche Spalte als X- und welche als Y-Achse verwendet werden soll. Sobald alle erforderlichen Benutzereingaben vorliegen, kann der Nutzer die Abfrage der Diagrammdaten durch Klicken auf den *Plot*-Button initiieren, wodurch eine entsprechende Anfrage an den Webserver gesendet wird.

Webserver

In den vorherigen Abschnitten wurde der im Prototyp verwendete Webserver als zusammenhängende Einheit präsentiert, was auf logischer Ebene zutrifft. Auf der Implementierungsebene ist der Webserver jedoch in mehrere Teile unterteilt, die über verschiedene Ports zugänglich sind. Der Grund für diese Aufteilung ergibt sich daraus, dass die für den Reiter *Viewer* zuständigen Webserver-Funkionen eine speziell konfigurierte Spark-Session benötigen, um mit einem bestimmten Lakehouse-Format arbeiten zu können. Für die prototypische Implementierung erwies es sich als notwendig, jede dieser Spark-Sessions in einem eigenen Webserver vorzuhalten, da mehrere Spark-Sessions innerhalb eines Ausführungskontextes von Apache Spark nicht unterstützt werden. Außerdem führt die Einrichtung einer Spark-Session zu Verzögerungen beim Starten des Webservers und verlängert somit die Zeit, bis dieser für die Verarbeitung von Anfragen bereitsteht. Wenn ein Nutzer den Prototyp zur Definition, Ausführung oder Überwachung von Daten-Pipelines verwenden möchte, ist keine Spark-Session erforderlich. Daher ist es sinnvoll, die Webserver-Funktionen für diese Aufgaben von denen zu trennen, die datenverarbeitende Operationen beinhalten und deshalb eine Spark-Session erfordern. Je nach Lakehouse-Format sendet der Prototyp eine Anfrage an den entsprechenden Webserver, der bereits eine passend konfigurierte Spark-Session eingerichtet hat. Zur Verarbeitung von Anfragen greifen beide Webserver auf dieselben Funktionen zurück. Die Implementierung dieser Funktionen ist formatunabhängig, da ihnen das angeforderte Lakehouse-Format als Eingabeparameter übergeben wird.

Die erste vom Dashboard unterstützte Abfragevariante erfordert keine zusätzliche Verarbeitung der Daten durch den Webserver. Stattdessen werden alle Spalten der angefragten Tabelle im Batch-Modus in ein Spark-Dataframe eingelesen, wobei die Abfrage die vom Nutzer angeforderte Zeilenanzahl berücksichtigt. Anschließend wird das Ergebnis der Abfrage an die Dash-Anwendung übermittelt. Bei der zweiten Abfragevariante erfolgt die Abfrage über die benutzerdefinierte Spark-SQL-Anfrage, die zunächst keinen Bezug zu einer bestimmten Tabelle aufweist, sondern stattdessen den Platzhalter <*table*> enthält. Im Webserver wird dieser Platzhalter durch eine Zeichenkette ersetzt, welche die angeforderte Tabelle anhand ihres Speicherpfads und ihres Namens identifiziert. Hinsichtlich dieser Zeichenkette müssen die spezifischen Anforderungen der Lakehouse-Frameworks beachtet werden. Nach der Identifikation der Tabelle werden die Daten im Batch-Modus abgefragt und die Ergebnisse an die Dash-Anwendung zurückgegeben.

6.4. Eigenschaften des Prototyps

Dieser Abschnitt gibt einen Überblick über eine Reihe von Eigenschaften, die dem Prototyp zugeschrieben werden können. Für jede Eigenschaft wird beschrieben, wie und in welchem Umfang sie vom Prototyp berücksichtigt wird. Darüber hinaus lassen sich mehrere Einschränkungen ableiten, die bei einer künftigen Weiterentwicklung oder Neuimplementierung des Prototyps behoben werden sollten. Dafür werden potentielle Lösungen vorgestellt, die skizzieren, wie die Eigenschaften auf Implementierungsebene umfassender durchgesetzt werden könnten.

Plattformunabhängigkeit

Der Prototyp verwendet plattformspezifische Details der beiden Lakehouse-Frameworks Delta Lake und Apache Iceberg, sowie des Prozessierungsframeworks Apache Spark, die zur Ausführung einer Daten-Pipeline zwingend erforderlich sind. Eine Leitlinie während der Entwicklung bestand darin, die Eigenheiten dieser Frameworks von der Modellierungsfunktionalität zu trennen und sie in separaten Artefakten zu kapseln, um die Plattformunabhängigkeit der Implementierung zu gewährleisten. Deshalb sind alle vom Frontend benötigten, plattformspezifischen Parameter im JSON-Schema zusammengefasst, das in Abschnitt 6.2.1 vorgestellt wurde. Der Webserver benötigt die Klasse PIPELINE, wie in Abschnitt 6.3.1 beschrieben, um ausführungsrelevante Parameterwerte aus der YAML-Definition einer Daten-Pipeline zu extrahieren und damit Flow Deployments zu konfigurieren. Allerdings lässt sich mit dem gewählten Implementierungsansatz eine Plattformabhängigkeit des Backends nicht vermeiden. Die in PySpark verfassten Task-Definitionen sind ausschließlich für die Nutzung mit Apache Spark als Prozessierungsframework konzipiert. Da diese Tasks jedoch das angeforderte Lakehouse-Format als Eingabeparameter erhalten, werden sie für mehrere Lakehouse-Frameworks genutzt. Innerhalb der Tasks werden spezifische Eigenheiten eines Lakehouse-Frameworks durch `if-else`-Verzweigungen berücksichtigt.

Automatisierung

Über den Prototyp können Daten-Pipelines auf Basis deklarativer Beschreibungen im YAML-Format automatisiert ausgeführt werden. Sobald der Webserver eine valide YAML-Definition empfangen hat, erfolgt die Ausführung der Daten-Pipeline automatisch, ohne dass der Nutzer weitere Schritte unternehmen muss. Dieser Ansatz erlaubt es, auf die manuelle Erstellung von Pipeline-Code zu verzichten und bereits existierende YAML-Definitionen als Vorlagen für neue Daten-Pipelines zu verwenden. Allerdings bietet der entwickelte Prototyp keine automatisierte Monitoring-Funktionalität, um Ausführungsfehler einer Daten-Pipeline automatisch zu erkennen und zu beheben. Um dieses Problem zu lösen, könnte auch ein verbessertes Modellierungswerkzeug auf Informationen der Prefect-Orchestrierungsengine zurückgreifen, welche über die Prefect-API abrufbar sind. Diese Informationen enthalten beispielsweise Details zum Ausführungszustand eines Flow Runs. Anstatt wie der Prototyp eine Visualisierung zur Überwachung einer Daten-Pipeline zu generieren, könnte das Modellierungswerkzeug basierend auf diesen Informationen im Fehlerfall automatisch eine geeignete Fehlerbehebungsmaßnahme einleiten, ohne dass manuelles Eingreifen des Nutzers notwendig ist. Falls eine automatische Fehlerbehebung scheitert, könnte der Nutzer zumindest automatisch über den Fehler informiert werden.

Erweiterbarkeit

Der Prototyp kann um neue Lakehouse-Frameworks und Prozessierungsframeworks erweitert werden, indem ihre plattformspezifischen Details zu den für die Eigenschaft *Plattformunabhängigkeit* aufgeführten Artefakten hinzugefügt werden. Dies beinhaltet eine Erweiterung des JSON-Schemas um eine Reihe von JSON-Objekten, die als Vorgaben für Deployment-Abschnitte in der YAML-Definition einer Daten-Pipeline dienen und plattformspezifische Parameter zur Konfiguration eines Frameworks definieren. Sollte ein JSON-Objekt komplexe Datenstrukturen wie Listen oder zusätzliche Verschachtelungsebenen erfordern, können Änderungen an der Klasse PIPELINE notwendig sein, um festzulegen, wie die entsprechenden Deployment-Abschnitte in ein flaches Python-Dictionary abzubilden sind. Weitere Anpassungen sind im Backend notwendig, wobei der Änderungsaufwand davon abhängt, ob ein zusätzliches Lakehouse-Framework oder ein zusätzliches Prozessierungsframework unterstützt werden soll. Zur Unterstützung eines neuen Lakehouse-Frameworks können die in Abschnitt 6.3.1 behandelten Task- und Flow-Definitionen übernommen werden. Jedoch muss der Task erweitert werden, der für die Einrichtung der Spark-Session zuständig ist, um sicherzustellen, dass die Spark-Session die erforderlichen Pakete zur Arbeit mit dem vom Lakehouse-Framework definierten Tabellenformat einbezieht. Die Integration eines weiteren Prozessierungsframeworks ist aufwendiger, da die Task- und Flow-Definitionen in PySpark verfasst sind und somit vom Prozessierungsframework Apache Spark abhängen. Daher müssten die Task-Definitionen neu implementiert werden, um sie an die Anforderungen eines neuen Prozessierungsframeworks anzupassen. Ähnlich wie die Erweiterung um neue Frameworks gestaltet sich die Einführung zusätzlicher Zonen. Hierbei müssen neue Pipeline-Schritt-Typen im JSON-Schema und in der PIPELINE-Klasse bekanntgegeben werden. Wenn jedoch ein neuer Pipeline-Schritt-Typ eine Abfolge von Tasks bedingt, die einer bestehenden Flow-Definition entsprechen, fallen keine Änderungen im Backend an.

Unterstützung externer Datenquellen

Die Fähigkeit des Prototyps, externe Daten in ein Lakehouse aufzunehmen, ist eingeschränkt. Er ist dazu in der Lage, Events von Kafka-Topics im Streaming-Modus in ein Lakehouse einzuspeisen. Als externe Datenquellen für Batch-Daten akzeptiert der Prototyp einzelne Dateien, in denen die Daten im CSV- oder JSON-Format abgelegt sind. Um das Potential eines Lakehouses als integrierte Datenplattform für verschiedene Arten von Batch- und Streaming-Daten auszuschöpfen, ist es von entscheidender Bedeutung, die Unterstützung für externe Datenquellen zu erweitern. Ein Modellierungswerkzeug sollte in dieser Hinsicht keine Einschränkungen auferlegen. Eine Möglichkeit zur Erweiterung des Prototyps wäre die Unterstützung von anderen Streaming-Plattformen wie *Azure Event Hub*[14] oder *Amazon Kinesis*[15], sowie die Anbindung an relationale Datenbanken oder NoSQL-Datenbanken zur Extraktion von Batch-Daten.

[14] https://learn.microsoft.com/de-de/azure/event-hubs
[15] https://aws.amazon.com/de/kinesis

Unterstützung von benutzerdefiniertem Code

Der Prototyp ermöglicht es, benutzerdefinierten Code zur Spezifikation der Verarbeitungslogik von Pipeline-Schritten zu verwenden. Dabei kann pro Pipeline-Schritt ein Codeanhang bereitgestellt werden, der bestimmten Vorgaben unterliegt. Insbesondere muss der Code spezifizieren, wie Daten der Source-Tabellen eines Pipeline-Schritts zu transformieren sind, so dass sie in den Sink-Tabellen abgelegt werden können. Der für die Transformation zuständige Task wird in Abschnitt 6.3.1 vorgestellt. Im Backend wird der Task durch eine Python-Funktion repräsentiert, die als Eingabe ein oder mehrere Spark-Dataframes erhält, welche die Source-Tabellen darstellen. Als Ausgabe der Python-Funktion resultieren ein oder mehrere Spark-Dataframes, die den Sink-Tabellen entsprechen. Der benutzerdefinierte Code wird sowohl durch die Manipulationsmöglichkeiten eines Spark-Dataframes als auch durch die im Python-Skript verfügbaren Bibliotheken eingeschränkt. Bei einer Weiterentwicklung des Prototyps sollte darauf geachtet werden, dass der benutzerdefinierte Code nicht auf den Geltungsbereich einer Funktion und auf die Manipulation von Dataframes beschränkt ist. Ein Modellierungswerkzeug sollte eine größere Vielfalt externer Bibliotheken bereitstellen oder nachladen können, um eine flexiblere Spezifikation der Verarbeitungslogik zu ermöglichen. Auf diese Weise wäre es dann beispielsweise möglich, Machine Learning Modelle innerhalb einer Daten-Pipeline zu trainieren.

Skalierbarkeit

Die Architektur des Prototyps zeichnet sich durch ihre horizontale Skalierbarkeit aus. Diese Eigenschaft ergibt sich aus den für Webserver und Backend gewählten Technologien, sowie dem Zusammenspiel dieser beiden Komponenten bei der Verarbeitung von Benutzeranfragen. Da der Prototyp jedoch in einer lokalen Umgebung entwickelt wurde, bedarf es einer Anpassung seiner Implementierung, um eine Skalierung in einer verteilten Umgebung zu ermöglichen. Im Zuge einer Überführung des Modellierungswerkzeugs in eine skalierbare Lösung muss eine horizontale Skalierbarkeit auf drei Ebenen hergestellt werden: Das Modellierungswerkzeug muss in der Lage sein, mit einer steigenden Anzahl von Benutzeranfragen an den Webserver, mit einer steigenden Anzahl von parallel laufenden Daten-Pipelines, sowie mit einer größer werdenden Datenmenge innerhalb einer Daten-Pipeline umzugehen. Für die Bewältigung der ersten Anforderung ist der Quart-Webserver verantwortlich. Quart basiert auf dem Framework Asynchronous Server Gateway Interface (ASGI), das für die asynchrone Verarbeitung von Anfragen konzipiert ist und somit die horizontale Skalierung einer Anwendung begünstigt. Sämtliche Webserver-Funktionen, die der entwickelte Prototyp verwendet, sind asynchron implementiert und zustandslos. Demnach könnten beispielsweise mehrere Instanzen des Quart-Webservers hinter einem Load Balancer bereitgestellt werden, um sicherzustellen, dass eingehende Anfragen gleichmäßig auf die Instanzen verteilt werden. Für die Bewältigung der beiden anderen genannten Anforderungen ist hingegen die horizontale Skalierbarkeit der Backend-Technologien entscheidend. Um eine steigende Anzahl von Daten-Pipelines parallel auszuführen, kann eine horizontale Skalierung auf der Ebene des Orchestrierungsframeworks Prefect erreicht werden. Je nach Arbeitslast könnte die Anzahl der Agents angepasst werden, die für die Abarbeitung einer Work Queue zuständig sind. Prefect bietet verschiedene Agent-Typen an, darunter auch sogenannte *Kubernetes-Agents*[16], die Flow

[16] https://prefecthq.github.io/prefect-kubernetes

Runs als Kubernetes-Jobs bereitstellen können. Kubernetes ist eine Open-Source Container-Orchestrierungsplattform, die unter anderem für die Skalierung von containerisierten Anwendungen in einem Kubernetes-Cluster verwendet wird. Um mit einer größer werdenden Datenmenge innerhalb einer Daten-Pipeline umzugehen, könnte auch eine Kombination von Apache Spark mit Kubernetes in Betracht gezogen werden.[17] In diesem Fall würde Apache Spark die Rechenressourcen des Kubernetes-Clusters nutzen, um eine verteilte Datenverarbeitung zu realisieren. Aufgrund des begrenzten zeitlichen Rahmens dieser Masterarbeit wurde jedoch weder untersucht, wie die Konzepte zur Skalierung auf Orchestrierungs- und Prozessierungsebene zusammengeführt werden können, noch ob eine solche Lösung auch auf andere Prozessierungsframeworks übertragbar wäre.

[17] https://spark.apache.org/docs/latest/running-on-kubernetes

7. Evaluation

In diesem Kapitel wird evaluiert, inwiefern die in Kapitel 4 eingeführten Anforderungen an ein Metamodell zur Modellierung von Daten-Pipelines in Lakehouses im Rahmen dieser Masterarbeit erfüllt werden konnten. Zu diesem Zweck wird jede Anforderung einzeln aufgegriffen und geprüft, inwieweit sie durch den vorgeschlagenen Lösungsansatz erfüllt wird. Wo es zur Untermauerung der Begründung sinnvoll erscheint, werden Bezüge zu vorhergehenden Kapiteln hergestellt oder einfache Beispiele angeführt, anhand derer sich der Erfüllungsgrad einer Anforderung erschließen lässt. Die Ergebnisse dieser Bewertung sind in Tabelle 7.1 zusammengefasst. Sie gibt einen Überblick über die evaluierten Anforderungen einschließlich ihres Erfüllungsgrads und bietet damit eine komprimierte Darstellung der Bewertungsergebnisse, die in den folgenden Unterabschnitten begründet werden.

A1: Berücksichtigung der Delta-Architektur

Anforderung A1 setzt voraus, dass das Metamodell die Delta-Architektur berücksichtigen muss. Dadurch wird der Anwendungsbereich des Metamodells auf Daten-Pipelines in Lakehouses eingeschränkt, da aktuell nur diese Datenplattformen in der Lage sind, die Delta-Architektur umzusetzen. Um A1 zu erfüllen, muss das Metamodell die Fähigkeit besitzen, Tabellen als Datenquellen- und senken für Batch- und Stream-Verarbeitung zu modellieren. Außerdem sollte ein Nutzer auf Ebene einzelner Pipeline-Schritte bestimmen können, ob ein Verarbeitungsschritt im Batch- oder Streaming-Modus erfolgen soll. Im Metamodell erschließt sich die Berücksichtigung der Delta-Architektur ausgehend vom zentralen Entitätstyp Pipeline Step, der einen Bezug zwischen Datenverarbeitung und Datenablage einer Daten-Pipeline einführt. Für eine Entität

#	Name	Erfüllungsgrad
A1	Berücksichtigung der Delta-Architektur	●
A2	Modularer Aufbau	●
A3	Flexibilität und Änderbarkeit	●
A4	Beschreibung der Eigenschaften externer Datenquellen	◕
A5	Integration von Daten verschiedener externer Datenquellen	●
A6	Speicherung von Zwischenergebnissen	●
A7	Unterstützung komplexer Filter- und Bereinigungsoperationen	◕
A8	Unterstützung selbstanpassender Daten-Pipelines	●
A9	Bezug zu einem Zonenmodell	●
A10	Berücksichtigung von Data Governance	◐

○ nicht erfüllt ◐ zur Hälfte erfüllt ◕ größtenteils erfüllt ● vollständig erfüllt

Tabelle 7.1.: Übersicht über den Erfüllungsgrad der Anforderungen, dargestellt durch Harvey-Bälle.

dieses Typs kann spezifiziert werden, in welchem Modus er die Eingabedaten verarbeitet soll. Der Verarbeitungsmodus ist im Metamodell als Spezialisierungshierarchie abgebildet, um dem Nutzer eine Auswahl zwischen Batch- und Streaming-Modus zu gestatten. Zur Konfiguration einer Batch- oder Stream-Verarbeitung bietet das Metamodell die Möglichkeit, den Ausgabemodus sowie das zeitliche Verhalten eines Pipeline-Schritts festzulegen. Die dafür verfügbaren Optionen sind an den Funktionsumfang gängiger Prozessierungsframeworks, wie Apache Spark und Apache Flink, angelehnt. Hinsichtlich der Datenablage eines Pipeline-Schritts sieht das Metamodell die beiden Beziehungstypen EXTRACTS DATA FROM und LOADS DATA INTO zwischen Pipeline-Schritten und Tabellen vor. Aus diesen leitet sich eine Unterscheidung zwischen Source- und Sink-Tabellen eines Pipeline-Schritts ab. Entsprechend den Vorgaben der Delta-Architektur wird im Metamodell jedoch nicht zwischen Tabellen unterschieden, die Streaming-Daten speichern, und solchen, die ausschließlich Batch-Daten speichern. Deshalb ermöglicht das Metamodell beispielsweise die Verwendung derselben Tabelle als Sink-Tabelle zweier Pipeline-Schritte, die in unterschiedlichen Verarbeitungsmodi ausgeführt werden. Es erlaubt auch Daten-Pipelines, die sich ausschließlich aus Pipeline-Schritten im Streaming-Modus zusammensetzen, um echtzeitnahe Anwendungen zu unterstützen. Eine solche Anwendung könnte zusätzlich verlangen, dass die Streaming-Daten vor ihrer Bereitstellung durch historische Daten angereichert werden. Dann bietet sich eine Daten-Pipeline an, die sich sowohl aus Streaming- als auch Batch-Pipeline-Schritten aufbaut. Dass das Metamodell ein derartiges Szenario abbilden kann, belegt das Beispiel einer Daten-Pipeline, die IoT-Messwerte mit Gerätedaten anreichert, wie es in Abschnitt 5.4 vorgestellt wird. Aus dieser Betrachtung geht hervor, dass das Metamodell auf die Besonderheiten der Delta-Architektur eingeht. Daher wird der Erfüllungsgrad von A1 als vollständig bewertet.

A2: Modularer Aufbau

Anforderung A2 schreibt vor, dass das Metamodell einen modularen Aufbau von Daten-Pipelines durchsetzen muss. Deshalb sollte eine Daten-Pipeline aus mehreren Verarbeitungsschritten aufgebaut werden, die sich mit wenig Aufwand austauschen und wiederverwenden lassen. Das vorgeschlagene Metamodell abstrahiert abgeschlossene Daten-Pipelines und führt Modularität auf unterschiedlichen Abstraktionsebenen ein. Erstens legt es fest, dass eine Daten-Pipeline aus einzelnen, unabhängigen Bausteinen besteht, die als Pipeline-Schritte bezeichnet werden. Diese sind deswegen als unabhängig einzustufen, weil jeder Pipeline-Schritt einer potentiell mehrschrittigen Daten-Pipeline in sich abgeschlossen ist. Er bündelt alle Informationen, anhand derer die durch ihn bewirkte Datenverarbeitung und Datenablage beschrieben werden kann. Da ein Pipeline-Schritt keine Abhängigkeiten zu anderen Pipeline-Schritten derselben oder anderer Daten-Pipelines eingeführt, könnte jeder Pipeline-Schritt einzeln ausgeführt werden. Zweitens kann eine abgeschlossene Daten-Pipeline optional selbst in eine übergeordnete Daten-Pipeline eingegliedert werden, was den zugrundeliegenden modularen Aufbau unterstreicht. Diese Verschachtelung von Daten-Pipelines wird im Metamodell durch einen Selbstbezug des Entitätstyps DATA PIPELINE bewirkt. Dabei bewahrt das Metamodell die Modularität und die Unabhängigkeit aller untergeordneten Daten-Pipelines. Diese können trotz der Verschachtelung unabhängig von der übergeordneten Daten-Pipeline ausgeführt werden, da sie gemäß dem Metamodell selbst valide Daten-Pipelines darstellen. Insgesamt gewährleistet der durch das Metamodell vorgesehene modulare Aufbau von Daten-Pipelines, dass sowohl einzelne Pipeline-Schritte als auch untergeordnete Daten-Pipelines in anderen Daten-Pipelines wiederverwendet werden können, was sich durch den Erhalt der Unabhängigkeit der genannten Bausteine begründen lässt. Darüber hinaus stellt das Metamodell sicher, dass diese Bausteine einfach ausgetauscht werden

können. Da das Metamodell die Definition eines Bausteins vom Rest der Daten-Pipeline abgrenzt, wird deutlich herausgestellt, wo ein Austausch ansetzen kann. Diese Begründung lässt sich auf das Hinzufügen, das Entfernen und auf die Änderung der Reihenfolge von Pipeline-Schritten oder untergeordneter Daten-Pipelines übertragen. Zusammenfassend geht daraus hervor, dass Modularität in den Vorgaben des Metamodells verankert ist und sich auf verschiedenen Abstraktionsebenen einer damit beschriebenen Daten-Pipeline widerspiegelt. A2 wird daher als vollständig erfüllt erachtet.

A3: Flexibilität und Änderbarkeit

Anforderung A3 bedingt, dass das Metamodell auf Flexibilität und Änderbarkeit ausgelegt ist. Dadurch sollen Daten-Pipelines auf vielfältige Verarbeitungsaufgaben zugeschnitten und mit wenig Aufwand an geänderte Bedienungen angepasst werden können, wie zum Beispiel an eine Änderung der Eigenschaften externer Datenquellen oder der Verarbeitungsaufgabe. Im Folgenden werden mehrere Punkte beleuchtet, die darauf hinweisen, dass A3 vom Metamodell erfüllt wird. Zunächst ist festzustellen, dass der modulare Aufbau einer Daten-Pipeline, wie er von A2 gefordert wird, die Erreichung beider in A3 aufgeführten Eigenschaften begünstigt. Demnach ist eine nach den Vorgaben des Metamodells erstellte Daten-Pipeline als flexibel und änderbar zu bezeichnen, weil sie sich aus unabhängigen Bausteinen zusammensetzt. Wenn sich beispielsweise die Datenqualität einer externen Datenquelle verschlechtert, könnten zusätzliche Datenbereinigungsmaßnahmen innerhalb einer Daten-Pipeline nötig werden. Infolgedessen könnte ein Nutzer einen neuen Pipeline-Schritt definieren und ihn an passender Stelle zu den vorhandenen Pipeline-Schritten der Daten-Pipeline hinzufügen, wobei in diesem Beispiel von jeweils einer Source-Tabelle und einer Sink-Tabelle je Pipeline-Schritt ausgegangen wird. Bei der Definition des neuen Pipeline-Schritts muss der Nutzer beachten, dass dessen Source-Tabelle mit der Sink-Tabelle des vorgelagerten Pipeline-Schritts übereinstimmt und dass zur Ablage seiner Zwischenergebnisse eine neue Tabelle angelegt werden muss. Die einzige Änderung an der bestehenden Daten-Pipeline ergibt sich für die Source-Tabelle des nachgelagerten Pipeline-Schritts, da diese der neuen Tabelle entsprechen muss. Ein weiterer Punkt, der für die Erfüllung von A3 spricht, ist die Fähigkeit des Metamodells, benutzerdefinierten Code zur Spezifikation der Verarbeitungslogik eines Pipeline-Schritts zuzulassen. Ein Nutzer des Metamodells wird also in die Lage versetzt, eine Daten-Pipeline auf eine spezifische Verarbeitungsaufgabe auszurichten. Da dem benutzerdefinierten Code keine Einschränkungen auferlegt sind, ist eine flexible Nutzung des Metamodells für die Abbildung vielfältiger Daten-Pipelines möglich. Implizit wird durch die Anbindung eines Pipeline-Schritts an Source- und Sink-Tabellen lediglich gefordert, dass der Code die Eingabedaten in die vom Nutzer gewünschten Ausgabedaten transformiert und dabei die Schemata der Tabellen berücksichtigt. Auch das in Kapitel 6 vorgestellte prototypische Modellierungswerkzeug lässt benutzerdefinierten Code zu. Um die Wiederverwendbarkeit des Codes in verschiedenen Pipeline-Schritten derselben oder anderer Daten-Pipelines zu erleichtern, verhindert der Prototyp eine Abhängigkeit zwischen Code und Pipeline-Schritt, indem er Platzhalter anstelle der definierten Tabellennamen im hinterlegten Code verlangt. Schließlich kann die Erfüllung von A3 auch durch die Grobstruktur des Metamodells untermauert werden, insbesondere wenn es um die Unterstützung von Änderbarkeit geht. Die Grobstruktur berücksichtigt das Designprinzip *Separation of Concerns*, bei dem die Komponenten eines Systems auf der Grundlage ihrer jeweiligen Aufgaben aufgeteilt werden. Dadurch kann die unabhängige Änderung verschiedener Komponenten erleichtert und verhindert werden, dass sich Änderungen an einer Komponente auf andere auswirken. Das Metamodell setzt eine Trennung zwischen den Bereichen *Datenverarbeitung* und *Datenverwaltung* durch, woraus sich Vorteile bei Änderungen innerhalb eines Pipeline-Schritts ableiten. Diese

Bereiche sind durch einen nicht überlappenden Satz von Entitätstypen modelliert und werden durch eine Entität vom Typ Pipeline Step in Bezug zueinander gesetzt. Folglich könnte zum Beispiel der Name der Sink-Tabelle eines Pipelines-Schritts geändert werden, ohne dass seine Verarbeitungsweise auf die Änderung abgestimmt werden muss. Aus den erläuterten Punkten geht hervor, dass eine hohe Flexibilität und Änderbarkeit gegeben ist, weshalb A3 als vollständig erfüllt betrachtet wird.

A4: Beschreibung der Eigenschaften externer Datenquellen

Anforderung A4 liegt die Idee zugrunde, dass sich das Metamodell nicht ausschließlich auf die Bestandteile einer Daten-Pipeline konzentrieren, sondern auch die Anbindung an externe Entitäten modellieren können sollte. Insbesondere muss es eine Möglichkeit bieten, die Eigenschaften externer Datenquellen zu beschreiben, um diese beim Aufbau und bei der Wartung einer Daten-Pipeline miteinzubeziehen. Dies soll einem Nutzer ermöglichen, ein besseres Verständnis für Ein- und Ausgabedaten einer Daten-Pipeline zu entwickeln. Für diesen Zweck sieht das vorgeschlagene Metamodell den Entitätstyp External Data Source vor, dessen Attribute sich aus dem Big-Data-Kontext ergeben. Eine Entität diesen Typs charakterisiert eine externe Datenquelle über die Menge, Struktur und Qualität der bereitgestellten Daten. Außerdem wird unterschieden, ob es sich um eine externe Datenquelle für Streaming-Daten oder Batch-Daten handelt. Indem das Metamodell diese zusätzliche Beschreibungsinformation einführt, befähigt es einen Nutzer, bei der Definition einer Daten-Pipeline schnellere und zuverlässigere Entscheidungen zu treffen, was im Folgenden beispielhaft skizziert wird. Angenommen er steht vor der Aufgabe, eine neue Streaming-Datenquelle an ein Lakehouse anzubinden, die semi-strukturierte Daten niedriger Qualität bereitstellt. Aus den Eigenschaften der externen Datenquelle könnte der Nutzer ableiten, dass der für die Einspeisung zuständige Pipeline-Schritt im Streaming-Modus arbeiten und mit semi-strukturierten Daten umgehen können muss. Außerdem könnte der Nutzer aus der Struktur der Daten folgern, dass ein weiterer Pipeline-Schritt erforderlich ist, um die Datenstruktur abzuflachen und relational zu strukturieren. Um die niedrige Datenqualität zu behandeln, könnte der Nutzer zusätzliche Pipeline-Schritte hinzufügen, die passende Datenqualitätsmaßnahmen durchführen. Das Beispiel zeigt, dass sich aus den Eigenschaften externer Datenquellen hilfreiche Implikationen für Zusammensetzung einer Daten-Pipeline sowie für einzelne Pipeline-Schritte ableiten, an denen sich ein Nutzer orientieren kann. Darüber hinaus fördert die Erfüllung von A3 auch die in A2 und A3 thematisierte Wiederverwendung von Pipeline-Schritten und untergeordneten Daten-Pipelines. Die Eigenschaften externer Datenquellen liefern einen Anhaltspunkt dafür, welche bestehende Daten-Pipelines gute Vorlagen darstellen, nach der neue externe Datenquellen in ein Lakehouse eingespeist und verarbeitet werden könnten. Da die im Metamodell enthaltenen Eigenschaften dem Big-Data-Kontext entnommen sind und dem Entwurf keine umfassende Analyse zur Ermittlung der aussagekräftigsten Eigenschaften vorausging, kann sich eine Erweiterung oder Änderung dieser als erforderlich erweisen. Deshalb wird A4 als größtenteils erfüllt angesehen.

A5: Integration von Daten verschiedener externer Datenquellen

Um Anforderung A5 zu erfüllen, muss das Metamodell die Integration verschiedener externer Datenquellen unterstützen. Dahinter stehen zwei Teilanforderungen, die nachfolgend separat behandelt werden. Zum einen sollte das Metamodell auf A5 eingehen, indem es die Unterschiede zwischen verschiedenen externen Datenquellen abstrahiert und Pipeline-Schritte unabhängig von

deren Eigenschaften hält. Dass das Metamodell diese Fähigkeit besitzt, leitet sich bereits aus der Evaluation von A4 ab. Da die Eigenschaften externer Datenquellen in speziell dafür vorgesehenen Entitäten vom Typ External Data Source gekapselt sind, werden diese nicht mit den Eigenschaften der Pipeline-Schritte vermischt, die externe Daten in ein Lakehouse einspeisen und verarbeiten. Dadurch verhindert das Metamodell, dass Pipeline-Schritte konzeptionell auf bestimmte externe Datenquellen zugeschnitten sind, weshalb ein Pipeline-Schritt mit beliebigen externen Datenquellen kompatibel ist. Zusätzlich bewirkt die Kapselung, dass Änderungen externer Datenquellen, mit denen bei einer großen Anzahl zu integrierender Datenquellen regelmäßig zu rechnen ist, zwar abgebildet werden können, jedoch nicht unmittelbar in einer Änderung der zuständigen Daten-Pipelines resultieren muss. Dies räumt einem Nutzer die Freiheit ein, nach eigenem Ermessen die Auswirkung der Änderung zu beurteilen und zu entscheiden, ob sie in der Daten-Pipeline widergespiegelt werden muss. Zum anderen verbindet sich mit A5 die Forderung, dass das Metamodell auch Pipeline-Schritte abbilden sollte, die Daten aus mehr als einer Tabelle als Eingabe beziehen, um eine integrierte Ausgabe in einer anderen Tabelle bereitzustellen. Im vorgeschlagenen Metamodell erschließt sich dieser Aspekt aus den Kardinalitätsbeschränkungen, die für Beziehungen zwischen Entitäten vom Typ External Data Source und Pipeline Step sowie für Beziehungen zwischen Entitäten vom Typ Table und Pipeline Step vorgesehen sind. Aus diesen folgt, dass ein einzelner Pipeline-Schritt dazu fähig ist, Daten beliebig vieler Datenquellen zu verarbeiten und seine Zwischenergebnisse in beliebig vielen Tabellen zu persistieren. Das Metamodell unterstützt somit n:m-Abbildungen zwischen Datenquellen und Datensenken, um einer großen Bandbreite von Verarbeitungsaufgaben gerecht zu werden, einschließlich Integrationsaufgaben. Beispielweise kann ein Pipeline-Schritt zwei Source-Tabellen über einen Join zusammenführen und das Ergebnis in einer Sink-Tabelle bereitstellen. Als Fazit der Betrachtung beider Teilanforderungen ist festzustellen, dass diese vom Metamodell beachtet werden, so dass A5 ebenfalls als vollständig erfüllt gilt.

A6: Speicherung von Zwischenergebnissen

Anforderung A6 verlangt, dass das Metamodell die Speicherung von Zwischenergebnissen ermöglichen sollte, um diese nachgelagerten Pipeline-Schritten oder anderen Daten-Pipelines verfügbar zu machen und auf diese Weise eine wiederholte Berechnung gleicher Ergebnisse zu verhindern. Das Metamodell geht auf diese Anforderung ein, indem es jeden Pipeline-Schritt über den Beziehungstyp Loads data into an mindestens eine Tabelle bindet, in der seine Ausgabe persistiert wird. Ein beliebiger anderer Pipeline-Schritt kann die Tabelle als Source-Tabelle verwenden und in seine Verarbeitung einbeziehen, sofern sein Zugriff im Einklang mit den Vorgaben des verwendeten Zonenmodells steht. Dementsprechend ist die Berücksichtigung von A6 in der Struktur des Metamodells verankert. Ergänzend ist darauf hinzuweisen, dass das Metamodell dennoch Flexibilität bei der Umsetzung von A6 auf Instanzebene zulässt. Es macht keine Aussage darüber, wie häufig Zwischenergebnisse innerhalb einer Daten-Pipeline gespeichert werden müssen, und auch die Anzahl der Verarbeitungsoperationen, die ein Pipeline-Schritt in seinem Code bündelt, ist nicht festgelegt. Dadurch kann sich das Metamodell den individuellen Anforderungen eines Anwendungsfalls anpassen und verlagert die Entscheidung, welche Zwischenergebnisse persistiert werden sollen, an den Nutzer. Da die Ausgabe jedes Pipeline-Schritts zwischengespeichert wird, ist es entscheidend, wie der Nutzer eine Daten-Pipeline in einzelne Pipeline-Schritte untergliedert. Um diesbezüglich eine Entscheidung zu treffen, muss der Nutzer die Vorgaben des Zonenmodells, die erforderlichen Verarbeitungsmodi und die Abhängigkeiten zwischen Pipeline-Schritten, die aus der Wiederverwendung von Ergebnissen resultieren, einbeziehen. Zum Beispiel könnte eine

Daten-Pipeline benötigt werden, um Streaming-Daten nach ihrer Ablage im Rohformat durch mehrere Verarbeitungsoperationen für eine externe Anwendung echtzeitnah aufzubereiten. Dabei wird angenommen, dass das Zonenmodell keine persistente Zone verlangt, die gesamte Verarbeitung im Streaming-Modus erfolgt und die Daten-Pipeline keine Schleifen enthält. Dann sind mehrere Alternativen denkbar, wie der Nutzer die Daten-Pipeline in Pipeline-Schritte aufteilen und damit implizit die Speicherung von Zwischenergebnissen steuert kann. Er könnte beispielsweise für jede Verarbeitungsoperation einen separaten Pipeline-Schritt definieren oder alle Verarbeitungsoperationen im Code eines einzelnen Pipeline-Schritts zusammenfassen. Letzteres ist nicht zu empfehlen, da die Verarbeitung nach einem Fehler nicht auf Zwischenergebnissen aufbauen kann, sondern vollständig wiederholt werden muss. Dennoch demonstriert das Beispiel die Anpassungsfähigkeit des Metamodell hinsichtlich A6. Insgesamt folgt aus diesen Überlegungen, dass das Metamodell A6 berücksichtigt, weshalb die Anforderung als vollständig erfüllt einzustufen ist.

A7: Unterstützung komplexer Filter- und Bereinigungsoperationen

In Anforderung A7 wird vom Metamodell erwartet, dass es komplexe Filter- und Bereinigungsoperationen unterstützt. Um dies zu erreichen, sollte es möglich sein, einem Pipeline-Schritt benutzerdefinierten Code zur Spezifikation seiner Verarbeitungslogik zu hinterlegen. Das Metamodell setzt diese Anforderung mithilfe des Attributs Code um, welches dem Entitätstyp Processing angehört. Die Bewertung von A3 zeigt, dass benutzerdefinierter Code Flexibilität bietet, da er potentiell beliebige Verarbeitungsoperationen enthalten kann. In Bezug auf A7 kann er verwendet werden, um Datenbereinigungsmaßnahmen sowohl auf die Eigenschaften externer Datenquellen als auch auf die Anforderungen externer Datensenken zuzuschneiden. Dazu können vielfältige Bereinigungsschritte definiert werden, die sich nicht mit einer vordefinierten Palette von Filter- und Bereinigungsoperationen abbilden lassen. Um dem Nutzer trotz eines möglicherweise komplexen Codeanhangs eine schnelle Beurteilung der Verarbeitungslogik eines Pipeline-Schritts zu ermöglichen, ist der Entitätstyp Processing mit dem Entitätstyp Semantics verbunden. Über diesen können ein oder mehrere Operationen der relationalen Algebra angegeben werden, die eine Aussage über die vorgesehene Manipulation der eingegebenen Tabellendaten gestatten. Der in Kapitel 6 vorgestellte Prototyp belegt, dass es technisch machbar ist, den bei der Definition eines Pipeline-Schritts festgelegten benutzerdefinierten Code zu verwerten. Dazu reicht der Prototyp den Codeanhang an sein Backend weiter, damit dieser an passender Stelle in den eigentlichen Pipeline-Code eingebettet und ausgeführt werden kann. Wenn für eine Filter- oder Bereinigungsoperation externe Bibliotheken oder andere Ressourcen erforderlich werden, stößt die prototypische Implementierung an ihre Grenzen. Diese externen Elemente müssten im Code verfügbar gemacht werden, wofür der Prototyp derzeit keine Lösung bietet. Die Entwicklung und Umsetzung eines umfassenden Ansatzes zur Unterstützung von benutzerdefiniertem Code hätte den Rahmen dieser Masterarbeit überstiegen. Erfahrungen aus der Entwicklung des Prototyps legen nahe, das zusätzliche Vorgaben im Metamodell hilfreich sein könnten, um die Komplexität der Implementierung zu reduzieren. Beispielsweise könnte das Metamodell wichtige Abhängigkeiten und Konfigurationsparameter des Attributs Code explizit modellieren und gleichzeitig sicherstellen, dass dem Codeanhang nur minimale Beschränkungen auferlegt werden. Somit unterstützt das Metamodell komplexe Filter- und Bereinigungsoperationen, jedoch könnte die Modellierung in Bezug auf benutzerdefinierten Code umfassender gestaltet werden, um eine bessere Orientierung auf Implementierungsebene zu bieten. Aus den genannten Gründen wird A7 als größtenteils erfüllt erachtet.

A8: Unterstützung selbstanpassender Daten-Pipelines

Anforderung A8 besagt, dass das Metamodell in der Lage sein muss, selbstanpassende Daten-Pipelines zu modellieren. Um dies zu gestatten, darf das Metamodell eine Daten-Pipeline nicht als DAG abstrahieren, sondern muss auch zyklische Abhängigkeiten zwischen Pipeline-Schritten innerhalb einer Daten-Pipeline zulassen. Das vorgeschlagene Metamodell erfüllt diese Anforderung, indem es keine Restriktionen in Bezug auf die Konnektivität des zugrundeliegenden Datenflussgraphen einer Daten-Pipeline einführt. Es legt fest, dass jeder Pipeline-Schritt, sofern er sich nicht mit der Datenaufnahme externer Daten befasst, seine Eingabedaten aus einer oder mehreren Tabellen extrahiert und seine Ausgabedaten in einer oder mehreren Tabellen ablegt. Das Lakehouse gewährleistet, dass für den Zugriff auf Source- und Sink-Tabellen eines Pipeline-Schritts ACID-Eigenschaften durchgesetzt werden.

Deshalb kann ein Pipeline-Schritt ungeachtet anderer Pipeline-Schritte seiner Verarbeitungsaufgabe nachgehen, die möglicherweise auf dieselben Tabellen zugreifen. Insbesondere benötigt ein Pipeline-Schritt keine Kenntnis darüber, mit welchen Pipeline-Schritten er implizit über Source- und Sink-Tabellen verbunden ist, in welchem Verarbeitungsmodus diese arbeiten oder was deren Verarbeitungsaufgabe ist. Aus dieser Beschreibung der Vorgaben und Freiheiten, die das Metamodell für Pipeline-Schritte ausdrückt, geht hervor, dass auch selbstanpassende Daten-Pipelines unterstützt werden. Im Kontext dieser Masterarbeit verbindet sich mit dem Begriff *Selbstanpassung* die Fähigkeit eines Pipeline-Schritts, seine Verarbeitungslogik dynamisch anzupassen, indem er aktualisierte Konfigurationswerte aus einer Konfigurationstabelle bezieht. Dabei ist kein externer Eingriff erforderlich, da die Konfigurationswerte von einem Pipeline-Schritt bereitgestellt werden, der dem konfigurierten Pipeline-Schritt nachgelagert ist. Zur Veranschaulichung dieses Verfahrens wurde in Abschnitt 5.4 ein Beispiel für eine selbstanpassende Daten-Pipeline erläutert. Diese passt die Konfiguration eines Filterschritts im Streaming-Modus dynamisch an. Dazu leitet ein nachgelagerter Pipeline-Schritt aktuelle Konfigurationswerte aus historischen Daten ab und schreibt sie regelmäßig in eine Tabelle, die eine der Source-Tabellen des Filterschritts darstellt. Somit kann das Metamodell selbstanpassende Daten-Pipelines unterstützen, weshalb A8 als vollständig erfüllt betrachtet wird.

A9: Bezug zu einem Zonenmodell

Anforderung A9 gibt vor, dass das Metamodell den Bezug zu einem Zonenmodell herstellen muss, indem alle Source- oder Sink-Tabellen einer Daten-Pipeline einer Zone des Zonenmodells zugeordnet werden. Damit soll erreicht werden, dass die Datenverarbeitung innerhalb einer Daten-Pipeline konsistent zu den zugrundeliegenden Organisations- und Verwaltungsprinzipen des Lakehouses ist. Um eine Abhängigkeit des Metamodells zu einem bestimmten Zonenmodell zu vermeiden und den Nutzern Flexibilität bei individuellen Anpassungen einzuräumen, wird außerdem gefordert, dass das Zonenmodell die Anzahl und Art der Zonen nicht festschreibt. Das Metamodell berücksichtigt A9 durch die Einbettung des von Giebler et al. [GGH+20] vorgeschlagenen Metamodells für Zonen in Data Lakes. Dahinter steht die Übernahme des Entitätstyps Zone in das Metamodell für Daten-Pipelines und dessen Verbindung mit dem Entitätstyp Table. Giebler et al. [GGH+20] begründen, dass das Metamodell für Zonen durch die systematische Definition von Zonen und ihren Merkmalen als Rahmen für die Implementierung einer zonenbasierten Datenverwaltung in Data Lakes verwendet werden kann. Insbesondere schreiben sie dem Metamodell die Fähigkeit zu, dass es an verschiedene Anwendungsszenarien aus den Bereichen Reporting/OLAP und

Advanced Analytics angepasst werden kann, indem zum Beispiel in einer Implementierung nicht benötigte Zonen weggelassen werden. Wie in Abschnitt 5.1 dargestellt, wird angenommen, dass das Metamodell für Zonen in Data Lakes auch als Anleitung für die Datenverwaltung innerhalb eines Lakehouses eingesetzt werden kann. Welche Vorgaben aus einem instanziierten Zonenmodell in Bezug auf Daten-Pipelines resultieren, wird in Abschnitt 5.3 thematisiert. Dabei wird aufgezeigt, dass einer Tabelle aufgrund ihrer Zonenzuordnung bestimmte Eigenschaften zugeschrieben werden können und auch Pipeline-Schritte sich in Abhängigkeit davon kategorisieren lassen, in welcher Zone sich die Sink-Tabelle des Pipeline-Schritts befindet. Aufgrund der beschriebenen Einbettung des Metamodells für Zonen ist A9 als vollständig erfüllt einzustufen.

A10: Berücksichtigung von Data Governance

Gemäß Anforderung A10 muss das Metamodell eine Möglichkeit zur Einbindung von Data-Governance-Richtlinien vorsehen. Dazu sollte das Metamodell Daten-Pipelines annotieren können, die besonders zu schützende Daten verarbeiten, und außerdem gewährleisten, dass deren Verarbeitung unter Einhaltung vorgegebener Data-Governance-Richtlinien erfolgt. Das Metamodell für Data-Lake-Zonen schreibt für alle Zonen einen geschützten Bereich vor, in dem besonders sensible Daten zu speichern sind [GGH+20]. Durch dessen Einbettung in das vorgeschlagene Metamodell für Daten-Pipelines kann auf Tabellenebene eine Trennung zwischen sensiblen und weniger sensiblen Daten bewirkt werden. Auf Basis dieser Information lässt sich sicherstellen, dass eine Daten-Pipeline, die sensible Daten verarbeitet, Zwischenergebnisse ausschließlich in Sink-Tabellen ablegt, die dem geschützten Bereich einer Zone angehören. Eine Ausnahme gilt für Daten-Pipelines, die einen Desensibilisierungsschritt enthalten. Durch diesen sind nachgelagerte Pipeline-Schritte berechtigt, ihre Ausgabe im nicht geschützten Bereich einer Zone zu speichern. Auf andere Weise spiegelt sich Data Governance nicht im Metamodell wider. Da das Metamodell somit Data Governance ausschließlich für die Datenablage miteinbezieht und keine Aussagen darüber erlaubt, wie eine zulässige Verarbeitung sensibler Daten vorgenommen werden sollte, wird A10 lediglich als zur Hälfte erfüllt betrachtet.

8. Zusammenfassung und Ausblick

Lakehouses stellen integrierte, analytische Datenplattformen dar, über die sich die Delta-Architektur umsetzen lässt. Das Ziel der vorliegenden Masterarbeit bestand darin, ein Metamodell zu entwickeln, welches die Modellierung von Daten-Pipelines in Lakehouses gemäß der Delta-Architektur ermöglicht. Auf diese Weise sollte der Entwurf von Daten-Pipelines vereinfacht und eine Grundlage für ihre automatisierte Ausbringung und Überwachung geschaffen werden. Wie sich aus der Evaluierung in Kapitel 7 ergab, konnte die gestellte Aufgabe weitestgehend erfüllt werden.

Zusammenfassung

Das entwickelte Metamodell erlaubt die Modellierung von abgeschlossenen Daten-Pipelines, die sämtliche Verarbeitungsschritte zwischen der Datenaufnahme aus externen Batch- oder Streaming-Datenquellen und der Bereitstellung von verarbeiteten Daten für externe Datensenken abdecken. Hierfür legt das Metamodell fest, dass eine Daten-Pipeline aus unabhängigen Pipeline-Schritten besteht, die Daten im Batch- oder Streaming-Modus verarbeiten und ihre Zwischenergebnisse in einer oder mehreren Tabellen persistieren. Dabei unterstützt das Metamodell insbesondere auch selbstanpassende Daten-Pipelines und lässt zu, dass die Verarbeitungslogik eines Pipeline-Schritts durch benutzerdefinierten Code spezifiziert wird. Im Rahmen der Entwurfsarbeit wurde zudem ein Zonenreferenzmodell, welches ursprünglich für Data Lakes konzipiert wurde [GGH+20], auf Lakehouses angewendet, um das ursprüngliche Metamodell zu verfeinern. Das daraus resultierende, verfeinerte Metamodell identifiziert fünf verschiedene Typen von Pipeline-Schritten, welche bestimmten Zonen und Zonenübergängen des Zonenreferenzmodells zugeordnet werden können und eine logische Gliederung der Pipeline-Schritte einer Daten-Pipeline einführen. Auf Basis der genannten Metamodelle wurden Konzepte für die Entwicklung eines plattformunabhängigen Modellierungswerkzeugs vorgestellt, über das Daten-Pipelines deklarativ definiert und anschließend automatisiert ausgeführt werden können. Die vorgeschlagenen Konzepte sind an die Model-Driven Architecture (MDA) angelehnt, einem Ansatz zur modellbasierten Softwareentwicklung, welcher darauf abzielt, die Funktionalität eines Systems durch Modelle verschiedener Abstraktionsebenen und deren Transformationen zu beschreiben. Demnach kann das Modellierungswerkzeug ein plattformunabhängiges Daten-Pipeline-Modell durch eine definierte Transformation zwischen dem verfeinerten Metamodell und einem plattformspezifischen Metamodell automatisch auf bestimmte Plattformen anpassen und bis zu ausführbarem Pipeline-Code verfeinern.

Nach Abschluss der konzeptionellen Entwurfsphase wurde ein prototypisches Modellierungswerkzeug umgesetzt, welches sich an den vorgestellten Konzepten orientiert. Die Implementierung des Werkzeugs sieht eine dreischichtige Architektur vor, die aus einem Frontend, einem Webserver und einem Backend besteht und sich auf ein JSON-Schema stützt, das drei Pipeline-Schritt-Typen des verfeinerten Metamodells abbildet. Mittels des Frontends kann ein Nutzer des Werkzeugs Daten-Pipeline-Modelle im YAML-Format definieren und zur Ausführung an den Webserver übergeben. Dieser extrahiert relevante Parameter aus der YAML-Definition, um daraufhin Flow

Deployments zu konfigurieren und auszuführen. Ein Flow Deployment repräsentiert ein Element des Orchestrierungsframeworks Prefect und nutzt den im Backend hinterlegten Code zur Durchführung aller Aufgaben, die innerhalb eines bestimmten Pipeline-Schritt-Typs anfallen. Auf diese Weise kann ein Nutzer Daten-Pipelines definieren und in Lakehouses ausführen, die über eines der beiden Lakehouse-Frameworks Delta Lake oder Apache Iceberg realisiert sind. Als einziges Prozessierungsframework wird Apache Spark unterstützt, jedoch wurde die Implementierung auf Erweiterbarkeit ausgelegt. So können zusätzliche Prozessierungsframeworks hinzugefügt werden, ohne dass dies grundlegende Änderungen an der Funktionsweise des Prototyps erfordert. Neben der Ausführung von Daten-Pipelines ermöglicht der Prototyp auch deren Überwachung anhand der Visualisierung ihres Datenflussgraphen. Aus diesem lassen sich die Ausführungszustände der einzelnen Pipeline-Schritte ablesen. Außerdem besteht die Möglichkeit, Daten einer Tabelle abzufragen, um sicherzustellen, dass eine Daten-Pipeline die erwarteten Ergebnisse erzeugt.

Ausblick

Ansatzpunkte für die Verbesserung und Weiterentwicklung der Konzepte, die im Rahmen zukünftiger Arbeiten in Betracht gezogen werden können, erschließen sich aus der Evaluation des Metamodells in Kapitel 7. Dabei wurde festgestellt, dass das vorgeschlagene Metamodell nicht alle vorab definierten Anforderungen vollständig erfüllt. Insbesondere im Kontext von Data Governance bietet das vorgeschlagene Metamodell nur begrenzte Möglichkeiten. Es beschränkt sich auf die Vorgabe, dass sensible Daten im geschützten Teil einer Zone abzulegen sind, wie es vom eingebetteten Metamodell für Zonen in Data Lakes vorgesehen ist [GGH+20]. Eine Erweiterung des Metamodells für Daten-Pipelines wäre daher wünschenswert, um umfassendere Konzepte zur Handhabung sensibler Daten in einem Lakehouse bereitzustellen, welche über die Datenablage hinausgehen. Diese Konzepte sollten dafür sorgen, dass sensible Daten von der Einspeisung bis hin zur ihrer Bereitstellung an externe Datensenken gemäß den Data-Governance-Richtlinien des Unternehmens geschützt sind, auch während der Verarbeitung durch eine Daten-Pipeline. Eine weitere Verbesserungsmöglichkeit des Metamodells betrifft die Modellierung von benutzerdefiniertem Code. Obwohl das Metamodell benutzerdefinierten Code für Pipeline-Schritte zulässt, stellte sich bei der Implementierung des Prototyps heraus, dass hierbei zahlreiche Herausforderungen zu bewältigen sind. Um diese Herausforderungen zu adressieren, könnte das Metamodell präziser darauf eingehen, wie der benutzerdefinierte Code an einen Pipeline-Schritt anknüpft, wie seine Ein- und Ausgabeschnittstellen spezifiziert werden müssen und wie eine Anbindung an externe Bibliotheken und Ressourcen erfolgen muss. Schließlich bieten sich für das Metamodell erweiterte Beschreibungsmöglichkeiten externer Datenquellen an, von denen angenommen wird, dass sie eine Hilfestellung beim Aufbau und bei der Wartung von Daten-Pipelines darstellen. Die Eigenschaften sollten sorgfältig ausgewählt werden, um fundierte Rückschlüsse auf die erforderliche Verarbeitung externer Daten innerhalb einer Daten-Pipeline ziehen zu können.

In Bezug auf das prototypische Modellierungswerkzeug könnte eine Neuimplementierung unter Verwendung eines Implementierungsansatzes in Erwägung gezogen werden, der die an MDA angelehnten Konzepte mit dem angewendeten Implementierungsansatz in Einklang bringt. Ziel einer solchen Neuimplementierung könnte es insbesondere sein, die Funktionalität zu erweitern, so dass benutzerdefinierte, plattformunabhängige Daten-Pipeline-Modelle automatisch in plattformspezifische Daten-Pipeline-Modelle transformiert und diese bis hin zu ausführbarem Pipeline-Code

verfeinert werden können. Um die Plattformunabhängigkeit des Modellierungswerkzeugs zu gewährleisten und gegenüber dem entwickelten Prototyp auszubauen, sollte die Spezifikation der Transformation plattformspezifische Details verschiedener Lakehouse-Frameworks und Prozessierungsframeworks bereitstellen. Der automatisch generierte Pipeline-Code könnte als Flow Run über die Prefect-Orchestrierungsengine ausgeführt werden und die vom Prototyp verwendeten, parametrisierten Flow Deployments ersetzen, die für ihre Ausführung durch benutzerdefinierte Parameterwerte konfiguriert werden müssen. Dabei ist die vollständige Abbildung einer abgeschlossenen Daten-Pipeline auf Prefect-Elemente entscheidend. Dies würde sicherstellen, dass die Daten-Pipeline als Ganzes in der Prefect-Umgebung vorliegt und zur Ausführung nicht in einzelne Pipeline-Schritte aufgeteilt werden muss. Darüber hinaus sollte das weiterentwickelte Modellierungswerkzeug nicht nur die Ausführung der Daten-Pipeline automatisieren, sondern auch automatisierte Überwachungsfunktionen zur Verfügung stellen, wie zum Beispiel die automatische Erkennung und Behebung von Fehlern. Abgesehen von den funktionalen Eigenschaften des Modellierungswerkzeugs bietet auch seine Ausbringung Potential für Verbesserungen. Aufgrund der lokalen Entwicklung wurde lediglich bei der Auswahl der Technologien auf deren Fähigkeit zur horizontalen Skalierbarkeit geachtet. Um das Modellierungswerkzeug auch in skalierbaren Umgebungen einsetzen zu können, sollte seine Implementierung entsprechend angepasst werden.

Literaturverzeichnis

[ADS+20] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak, M. Świtakowski, M. Szafrański, X. Li, T. Ueshin, M. Mokhtar, P. Boncz, A. Ghodsi, S. Paranjpye, P. Senster, R. Xin, M. Zaharia. „Delta lake“. In: *Proceedings of the VLDB Endowment* 13.12 (Aug. 2020), S. 3411–3424. DOI: 10.14778/3415478.3415560 (zitiert auf S. 18, 23, 24).

[AGXZ21] M. Armbrust, A. Ghodsi, R. Xin, M. Zaharia. „Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics“. In: *Proceedings of CIDR*. 2021 (zitiert auf S. 15, 17, 18, 23–27, 37, 72).

[ASB19] R. Abraham, J. Schneider, J. vom Brocke. „Data governance: A conceptual framework, structured review, and research agenda“. In: *International Journal of Information Management* 49 (Dez. 2019), S. 424–438. DOI: 10.1016/j.ijinfomgt.2019.07.008 (zitiert auf S. 40).

[BCW17] M. Brambilla, J. Cabot, M. Wimmer. *Model-Driven Software Engineering in Practice*. Springer International Publishing, 2017. DOI: 10.1007/978-3-031-02549-5 (zitiert auf S. 59, 61–63).

[BGK21] E. Begoli, I. Goethert, K. Knight. „A Lakehouse Architecture for the Management and Analysis of Heterogeneous Data for Biomedical Research and Mega-biobanks“. In: *2021 IEEE International Conference on Big Data (Big Data)*. IEEE, Dez. 2021. DOI: 10.1109/bigdata52589.2021.9671534 (zitiert auf S. 26, 32).

[CY14] R. Casado, M. Younas. „Emerging trends and technologies in big data processing“. In: *Concurrency and Computation: Practice and Experience* 27.8 (Okt. 2014), S. 2078–2091. DOI: 10.1002/cpe.3398 (zitiert auf S. 22).

[CZ15] L. Cai, Y. Zhu. „The Challenges of Data Quality and Data Quality Assessment in the Big Data Era“. In: *Data Science Journal* 14.0 (Mai 2015), S. 2. DOI: 10.5334/dsj-2015-002 (zitiert auf S. 45).

[Dat] Databricks. *Medallion Architecture*. URL: https://www.databricks.com/glossary/medallion-architecture (zitiert auf S. 28).

[DS13] X. L. Dong, D. Srivastava. „Big data integration“. In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, Apr. 2013. DOI: 10.1109/icde.2013.6544914 (zitiert auf S. 38).

[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*. Springer International Publishing, 2020, S. 73–88. DOI: 10.1007/978-3-030-59065-9_7 (zitiert auf S. 19).

[EQS17] H. Eichelberger, C. Qin, K. Schmid. „Experiences with the model-based generation of Big Data pipelines“. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017) (zitiert auf S. 16).

[EQSS16] H. Eichelberger, C. Qin, R. Sizonenko, K. Schmid. „Using IVML to model the topology of big data processing pipelines“. In: *Proceedings of the 20th International Systems and Software Product Line Conference*. ACM, Sep. 2016. DOI: 10.1145/2934466.2934476 (zitiert auf S. 30, 34).

[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)*. IEEE, Juni 2015. DOI: 10.1109/cyber.2015.7288049 (zitiert auf S. 18).

[FPKJ10] A. Fouad, K. Phalp, J. M. Kanyaru, S. Jeary. „Embedding requirements within Model-Driven Architecture“. In: *Software Quality Journal* 19.2 (Dez. 2010), S. 411–430. DOI: 10.1007/s11219-010-9122-7 (zitiert auf S. 62, 63).

[Gel21] A. Geller. *You No Longer Need Two Separate Systems for Batch Processing and Streaming*. 2021. URL: https://www.prefect.io/guide/blog/you-no-longer-need-two-separate-systems-for-batch-processing-and-streaming (zitiert auf S. 70).

[Gel22] A. Geller. *How to Make Your Data Pipelines More Dynamic Using Parameters in Prefect: How to pass runtime-specific parameter values to your data pipelines*. 2022. URL: https://www.prefect.io/guide/blog/how-to-make-your-data-pipelines-more-dynamic-using-parameters-in-prefect (zitiert auf S. 70).

[GGH+19] C. Giebler, C. Gröger, E. Hoos, H. Schwarz, B. Mitschang. „Leveraging the Data Lake: Current State and Challenges“. In: *Big Data Analytics and Knowledge Discovery*. Springer International Publishing, 2019, S. 179–188. DOI: 10.1007/978-3-030-27520-4_13 (zitiert auf S. 18, 19).

[GGH+20] C. Giebler, C. Groger, E. Hoos, H. Schwarz, B. Mitschang. „A Zone Reference Model for Enterprise-Grade Data Lake Management“. In: *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, Okt. 2020. DOI: 10.1109/edoc49727.2020.00017 (zitiert auf S. 19–21, 38, 40, 42, 53, 54, 105–108).

[GSS21] S. Grafberger, J. Stoyanovich, S. Schelter. „Lightweight inspection of data preprocessing in native machine learning pipelines“. In: *Conference on Innovative Data Systems Research (CIDR)*. 2021 (zitiert auf S. 46).

[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*. SCITEPRESS - Science und Technology Publications, 2018. DOI: 10.5220/0006861802940301 (zitiert auf S. 21–23, 39).

[Han21] J. Hansen. *Selling the Data Lakehouse*. 2021. URL: https://medium.com/snowflake/selling-the-data-lakehouse-a9f25f67c906adfa (zitiert auf S. 24).

[HH10] M. Herschel, M. A. Hernández. „Explaining missing answers to SPJUA queries“. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), S. 185–196 (zitiert auf S. 48).

[HL19] B. Heintz, D. Lee. *Productionizing Machine Learning with Delta Lake*. 2019. URL: https://www.databricks.com/blog/2019/08/14/productionizing-machine-learning-with-delta-lake.html (zitiert auf S. 28).

[HQJ21] R. Hai, C. Quix, M. Jarke. „Data lake concept and systems: a survey“. In: (2021). DOI: 10.48550/ARXIV.2106.09592 (zitiert auf S. 18).

[HTG+22] S. Hambardzumyan, A. Tuli, L. Ghukasyan, F. Rahman, H. Topchyan, D. Isayan, M. McQuade, M. Harutyunyan, T. Hakobyan, I. Stranic, D. Buniatyan. *Deep Lake: a Lakehouse for Deep Learning*. 2022. DOI: 10.48550/ARXIV.2209.10785 (zitiert auf S. 31, 34).

[Hug] J. Hughes. *Apache Iceberg: An Architectural Look Under the Covers*. URL: https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers (zitiert auf S. 73).

[HZ22] A. A. Harby, F. Zulkernine. „From Data Warehouse to Lakehouse: A Comparative Review“. In: *2022 IEEE International Conference on Big Data (Big Data)*. IEEE, Dez. 2022. DOI: 10.1109/bigdata55660.2022.10020719 (zitiert auf S. 23).

[ISL21] B. Inmon, R. Srivastava, M. Levins. *Building the Data Lakehouse*. Technics Publications, LLC, 2021. ISBN: 9781634629669 (zitiert auf S. 17, 24–26).

[JKP+23] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, M. Zaharia. „Analyzing and Comparing Lakehouse Storage Systems“. In: (2023) (zitiert auf S. 64).

[KM21] D. Kraetz, M. Morawski. „Architecture Patterns—Batch and Real-Time Capabilities“. In: *The Digital Journey of Banking and Insurance, Volume III*. Springer International Publishing, 2021, S. 89–104. DOI: 10.1007/978-3-030-78821-6_6 (zitiert auf S. 16, 23, 26).

[Kre14] J. Kreps. *Questioning the Lambda Architecture*. 2014. URL: https://www.oreilly.com/radar/questioning-the-lambda-architecture (zitiert auf S. 23).

[KW18] P. P. Khine, Z. S. Wang. „Data lake: a new ideology in big data era“. In: *ITM Web of Conferences* 17 (2018). Hrsg. von K. Eguchi, T. Chen, S. 03025. DOI: 10.1051/itmconf/20181703025 (zitiert auf S. 17–19, 37).

[Lea20] H. Leano. *Delta vs. Lambda: Why Simplicity Trumps Complexity for Data Pipelines*. 2020. URL: https://www.databricks.com/blog/2020/11/20/delta-vs-lambda-why-simplicity-trumps-complexity-for-data-pipelines.html (zitiert auf S. 27, 28).

[LEs22] R. L'Esteve. „The Data Lakehouse Paradigm“. In: *The Azure Data Lakehouse Toolkit*. Apress, 2022, S. 3–41. DOI: 10.1007/978-1-4842-8233-5_1 (zitiert auf S. 28).

[LS14] L. Lyadova, A. Sukhov. „Visual Models Transformation in MetaLanguage System“. en. In: (2014). DOI: 10.13140/2.1.2349.7282 (zitiert auf S. 60).

[LSB+17] Z. Lv, H. Song, P. Basanta-Val, A. Steed, M. Jo. „Next-Generation Big Data Analytics: State of the Art, Challenges, and Future Research Topics“. In: *IEEE Transactions on Industrial Informatics* 13.4 (Aug. 2017), S. 1891–1899. DOI: 10.1109/tii.2017.2650204 (zitiert auf S. 15).

[LTS+18] X. Liu, S. Tamminen, X. Su, P. Siirtola, J. Roning, J. Riekki, J. Kiljander, J.-P. Soininen. „Enhancing Veracity of IoT Generated Big Data in Decision Making“. In: *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, März 2018. DOI: 10.1109/percomw.2018.8480371 (zitiert auf S. 39).

[Mat17] C. Mathis. „Data Lakes“. In: *Datenbank-Spektrum* 17.3 (Okt. 2017), S. 289–293. DOI: 10.1007/s13222-017-0272-7 (zitiert auf S. 18, 19).

[MBO20] A. R. Munappy, J. Bosch, H. H. Olsson. „Data pipeline management in practice: Challenges and opportunities“. In: *International Conference on Product-Focused Software Process Improvement*. Springer, 2020, S. 168–184 (zitiert auf S. 36).

[NM22] A. Nambiar, D. Mundra. „An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management“. In: *Big Data and Cognitive Computing* 6.4 (Nov. 2022), S. 132. DOI: 10.3390/bdcc6040132 (zitiert auf S. 17).

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

[ODw21] J. O'Dwyer. *How Incremental ETL Makes Life Simpler With Data Lakes*. 2021. URL: https://www.databricks.com/blog/2021/08/30/how-incremental-etl-makes-life-simpler-with-data-lakes.html (zitiert auf S. 27).

[OH21] D. Orescanin, T. Hlupic. „Data Lakehouse - a Novel Step in Analytics Architecture“. In: *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, Sep. 2021. DOI: 10.23919/mipro52101.2021.9597091 (zitiert auf S. 26, 31).

[PNM22] T. Priebe, S. Neumaier, S. Markus. „Von Data Warehouse bis Data Mesh: Ein Wegweiser durch den Dschungel analytischer Datenarchitekturen“. In: (2022). DOI: 10.48550/ARXIV.2212.03612 (zitiert auf S. 15, 26).

[Poe06] I. Poernomo. „The meta-object facility typed“. In: *Proceedings of the 2006 ACM symposium on Applied computing*. 2006, S. 1845–1849 (zitiert auf S. 60, 61).

[PVA19] F. Pervaiz, A. Vashistha, R. Anderson. „Examining the challenges in development data pipeline“. In: *Proceedings of the 2nd ACM SIGCAS Conference on Computing and Sustainable Societies*. ACM, Juli 2019. DOI: 10.1145/3314344.3332496 (zitiert auf S. 39).

[RBOW20] A. Raj, J. Bosch, H. H. Olsson, T. J. Wang. „Modelling Data Pipelines“. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Aug. 2020. DOI: 10.1109/seaa51224.2020.00014 (zitiert auf S. 29, 34).

[RK21] V. Raina, S. Krishnamurthy. „Natural Language Processing“. In: *Building an Effective Data Science Practice*. Apress, Dez. 2021, S. 63–73. DOI: 10.1007/978-1-4842-7419-4_6 (zitiert auf S. 24).

[RSGJ13] K. Raman, A. Swaminathan, J. Gehrke, T. Joachims. „Beyond myopic inference in big data pipelines“. In: *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, Aug. 2013. DOI: 10.1145/2487575.2487588 (zitiert auf S. 15, 36, 37).

[Run22] P. Runyan. *Workflow Orchestration without DAGs: How to Get Started with Dynamic Data Flows in Prefect 2.0*. 2022. URL: https://www.prefect.io/guide/blog/workflow-orchestration-without-dags (zitiert auf S. 70).

[RZ19] F. Ravat, Y. Zhao. „Data Lakes: Trends and Perspectives“. In: *Lecture Notes in Computer Science*. Springer International Publishing, 2019, S. 304–313. DOI: 10.1007/978-3-030-27615-7_23 (zitiert auf S. 17, 18).

[SBE+21] C. Stach, J. Bräcker, R. Eichler, C. Giebler, B. Mitschang. „Demand-Driven Data Provisioning in Data Lakes“. In: *The 23rd International Conference on Information Integration and Web Intelligence*. ACM, Nov. 2021. DOI: 10.1145/3487664.3487784 (zitiert auf S. 18).

[SGL+23] J. Schneider, C. Gröger, A. Lutsch, H. Schwarz, B. Mitschang. *Assessing the Lakehouse: Analysis, Requirements and Definition*. Techn. Ber. for. 2023 (zitiert auf S. 18, 23, 25–27, 35, 46, 72).

[TL03] J. Trujillo, S. Luján-Mora. „A UML Based Approach for Modeling ETL Processes in Data Warehouses“. In: *Conceptual Modeling - ER 2003*. Springer Berlin Heidelberg, 2003, S. 307–320. DOI: 10.1007/978-3-540-39648-2_25 (zitiert auf S. 29, 34, 39).

[WM15] J. Warren, N. Marz. *Big Data: Principles and best practices of scalable realtime data systems*. Simon und Schuster, 2015 (zitiert auf S. 22).

[WZX+16] D. Wu, L. Zhu, X. Xu, S. Sakr, D. Sun, Q. Lu. „Building Pipelines for Heterogeneous Execution Environments for Big Data Processing“. In: *IEEE Software* 33.2 (März 2016), S. 60–67. DOI: 10.1109/ms.2016.35 (zitiert auf S. 30, 34).

[YAH19] B. Yavuz, M. Armbrust, B. Heintz. *Diving Into Delta Lake: Unpacking The Transaction Log*. 2019. URL: https://www.databricks.com/blog/2019/08/21/diving-into-delta-lake-unpacking-the-transaction-log.html (zitiert auf S. 73).

[Yao22] A. Yao. „Interactive Query Explanations Using Fine Grained Provenance“. In: *Proceedings of the 2022 International Conference on Management of Data*. ACM, Juni 2022. DOI: 10.1145/3514221.3520251 (zitiert auf S. 48).

Alle URLs wurden zuletzt am 15. 04. 2023 geprüft.

A. Anhang

A.1. ER-Diagramm des Metamodells für Daten-Pipelines in Lakehouses

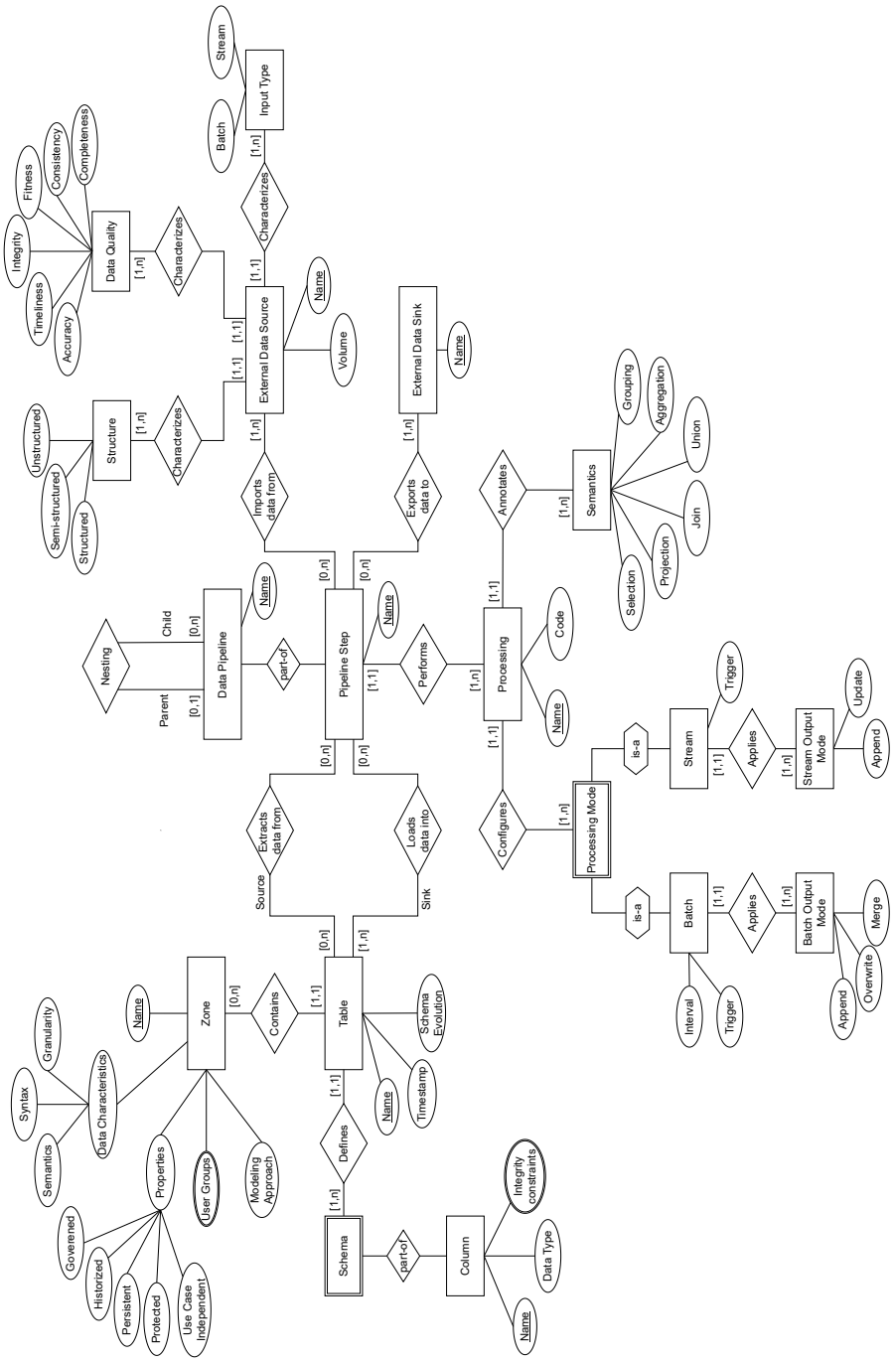

Abbildung A.1.: Entity-Relationship-Diagramm des entworfenen Metamodells für Daten-Pipelines in Lakehouses.

A.2. Beispielhafte YAML-Definition einer Daten-Pipeline

Listing A.1 Beispielhafte YAML-Definition einer Daten-Pipeline, die einem Ausschnitt des Daten-Pipeline-Modells entspricht, das in Abschnitt 5.4 beschrieben wurde.

```yaml
pipeline:
  name: harmonize_stream
  pipelineSteps:
  - harmonization:
      name: harmonize_stream
      sources:
      - table:
          name: raw_measurements
          zone: raw
          deployment:
            delta:
              dataPath: C://Users//kers//git//lakehouse-pipeline-composer//data//delta//raw//tables
              withEventTimeOrder: true
      processing:
        name: harmonize_stream_sck
        semantics: aggregation
        mapping: 1-1
        processingType:
          stream:
            outputMode: complete
        deployment:
          spark:
            schedulerPool: FAIR
            code: "source1 = source1.select(col('device_id'), col('name').alias('sensor_name'),\
              \ col('description').alias('sensor_description'), col('last_reading_at_at'),\
              \ col('id').alias('sensor_id'), col('unit'), col('raw_value'), col('value'))\r\
              \ngroup_cols = ['device_id', 'sensor_id']\r\nsource1 = \r\
              \source1.groupBy(window(source1.last_reading_at,'2')\,
              \source1.device_id, source1.sensor_id).agg(first('sensor_name').alias('sensor_name'),\r\
              \n first('sensor_description').alias('sensor_description'),\
              \ \\\r\n first('unit').alias('unit'), \\\r\n\t\t first('last_reading_at_at').alias('first_reading'),\
              \ \\\r\n\t\t last('last_reading_at_at').alias('last_reading_at'), \\\r\n\t\
              \t avg('raw_value').alias('raw_value'), \\\r\n\t\t avg('value').alias('value')\
              \ \\\r\n)\r\nsource1 = source1.drop(source1.window)\r\nsink1 =\
              \ source1"
            triggerType: processingTime
            triggerValue: 15 seconds
            mode: FAILFAST
      sinks:
      - table:
          name: harmonized_measurements
          zone: harmonized
          schemaEvolution: false
          schema:
          - field:
              name: device_id
              datatype: integer
          - field:
              name: first_reading
              datatype: timestamp
          - field:
              name: last_reading_at
              datatype: timestamp
          - field:
              name: sensor_name
              datatype: string
          - field:
              name: sensor_id
              datatype: integer
          - field:
              name: unit
              datatype: string
          - field:
              name: value
              datatype: double
          deployment:
            delta:
              dataPath: C://Users//kers//git//prototype//data//delta//harmonized//tables
              checkpointPath: C://Users//kers//git//prototype//data//delta//harmonized//checkpoints
```

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