LEUPHANA
UNIVERSITÄT LÜNEBURG

Fakultät Management und Technologie
Institut für Wirtschaftsinformatik

Bachelorarbeit

Modernisierung der bestehenden Datenarchitektur eines führenden deutschen Augenheilkundenetzwerks durch ein Data Lakehouse: Implementierung eines Prototypen

Modernization of the Existing Data Architecture of a Leading German Ophthalmology Network Through a Data Lakehouse: Implementation of a Prototype

Betreuer: Prof. Dr. Paul Drews
Erstgutachter: Prof. Dr. Paul Drews
Zweitgutachter: Jan Maschewski

Vorgelegt von:
Ioannis Leimonis, 3042822
Hammer Landstraße 236, 20537 Hamburg
io.leimonis@gmail.com
Wirtschaftsinformatik
Leuphana Universität Lüneburg

Abgabe: 26.03.2025
Hamburg, Datum

Sperrvermerk

Die vorliegende Bachelorarbeit mit dem Titel "Modernisierung der bestehenden Datenarchitektur eines führenden deutschen Augenheilkundenetzwerks durch ein Data Lakehouse: Implementierung eines Prototypen"beinhaltet interne und vertrauliche Informationen des Unternehmens Ober Scharrer Gruppe. Eine Einsicht Dritter ist nicht gestattet. Ausgenommen davon sind die Gutachter der Arbeit sowie die Mitglieder des Prüfungsausschusses. Eine Veröffentlichung und Vervielfältigung der Bachelorarbeit - auch in Auszügen – ist ohne die Genehmigung des Unternehmens Ober Scharrer Gruppe nicht gestattet.

Zusammenfassung

Die Verwaltung und Analyse großer Datenmengen stellt vielen Unternehmen vor Herausforderungen. Diese Arbeit untersucht die Modernisierung einer bestehenden Datenarchitektur in einem führenden deutschen Augenheilkundenetzwerk, durch die Umsetzung eines Data Lakehouse. Ziel ist es, die Vorteile von Data Warehouses und Data Lakes zu kombinieren, um eine skalierbare, wartbare und erweiterbare Lösung zu konzipieren.

Im Rahmen dieser Arbeit wurde ein prototypischer Data-Lakehouse-Ansatz entwickelt und implementiert. Hierzu wurden verschiedene Open-Source-Technologien eingesetzt, darunter Apache Spark für die verteilte Datenverarbeitung, Trino als SQL-Query-Engine, MinIO als Objektspeicher und Delta Lake zur Sicherstellung von ACID-Transaktionen. Der Hive Metastore dient als zentrales Metadatenmanagementsystem. Die Datenverarbeitung erfolgt nach der Medallion-Architektur, die in Bronze-, Silber- und Gold-Schichten unterteilt ist, um eine klare Trennung zwischen Rohdaten, bereinigten Daten und analytisch aufbereiteten Daten zu gewährleisten.

Der Prototyp wurde anhand eines simulierten Szenarios getestet, in dem Daten aus mehreren Datenbanken integriert und verarbeitet wurden. Zur Validierung wurde überprüft, ob Trino, MinIO, Delta Lake und der Hive Metastore korrekt zusammenarbeiten und eine zuverlässige Datenverarbeitung ermöglichen. Die abschließende Evaluation zeigt, dass die entwickelte Data-Lakehouse-Architektur im Vergleich zur bestehenden Lösung eine höhere Skalierbarkeit, Flexibilität und Wartbarkeit bietet.

Stichwörter: *Data Lakehouse, Data Warehouse, Data Lake, Medallion-Architektur, Apache Spark, Delta Lake, Trino, MinIO, Hive Metastore, Datenintegration, Skalierbarkeit, ACID-Transaktionen, Open-Source-Technologien*

Abstract

Managing and analyzing large volumes of data presents challenges for many companies. This paper examines the modernization of an existing data architecture in a leading German ophthalmology network through the implementation of a Data Lakehouse. The goal is to combine the advantages of Data Warehouses and Data Lakes in order to design a scalable, maintainable, and extensible solution.

As part of this work, a prototypical Data Lakehouse approach was developed and implemented. Various open-source technologies were used for this purpose, including Apache Spark for distributed data processing, Trino as a SQL query engine, MinIO as object storage, and Delta Lake to ensure ACID transactions. The Hive Metastore serves as the central metadata management system. Data processing follows the Medallion architecture, which is divided into bronze, silver, and gold layers to ensure a clear separation between raw data, cleaned data, and analytically prepared data.

The prototype was tested using a simulated scenario in which data from several databases was integrated and processed. For validation, it was checked whether Trino, MinIO, Delta Lake, and the Hive Metastore work together correctly and enable reliable data processing. The final evaluation shows that the developed Data Lakehouse architecture offers greater scalability, flexibility, and maintainability compared to the existing solution.

Keywords: *Data Lakehouse, Data Warehouse, Data Lake, Medallion Architecture, Apache Spark, Delta Lake, Trino, MinIO, Hive Metastore, Data Integration, Scalability, ACID Transactions, Open-Source Technologies*

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACID Atomicity, Consistency, Isolation, Durability. 1, 7, 8, 10, 31, II, III

API Application Programming Interface. 9, 12, 16

ELT Extract, Load, Transform. 4, 5

ETL Extract, Transform, Load. 1, 3, 4, 6, 22

KPI Key-Performance-Indicator. 23, 26, 34, V, XXI

OLAP Online Analytical Processing. 1, 4

ORC Optimized Row Columnar. 8

SQL Structured Query Language. 9, 10, 13, 14, 22, 23, 31, 33, 34, 36, XVII

1 Einleitung

Die moderne Datenverarbeitung steht vor der Herausforderung, große Datenmengen effizient zu verarbeiten und für unterschiedliche Anwendungen bereitzustellen. Im Gesundheitswesen, wie bei der Ober Scharrer Gruppe (OSG), werden täglich große Mengen an strukturierten und unstrukturierten Daten generiert, die für Analyse- und Verwaltungszwecke verarbeitet werden müssen. Traditionelle Datenplattformen wie Data Warehouses und Data Lakes wurden in der Vergangenheit genutzt, um diese Anforderungen zu erfüllen, sind jedoch jeweils auf spezifische Anwendungsbereiche beschränkt. Ein Data Warehouse ist für analytische Abfragen und Reporting optimiert, während ein Data Lake flexibel Rohdaten speichert und insbesondere für Machine-Learning-Anwendungen geeignet ist. Beide Ansätze haben jedoch Schwächen, wenn sie isoliert betrachtet werden. Während Data Lakes häufig Probleme mit der Datenkonsistenz und Abfrageeffizienz aufweisen, sind Data Warehouses für unstrukturierte Daten oft ungeeignet und teuer in der Skalierung (Armbrust et al, 2021; Harby und Zulkernine, 2022, S. 1–3, S. 4–5).

Die Data Lakehouse-Architektur wurde entwickelt, um diese Schwächen zu überwinden, indem sie die Stärken von Data Lakes und Data Warehouses kombiniert. Sie ermöglicht die Verarbeitung strukturierter und unstrukturierter Daten auf einer einzigen Plattform und unterstützt gleichzeitig die Anforderungen von OLAP (Online Analytical Processing) und Advanced Analytics. Dank Open-Source-Speicherformaten wie Apache Parquet und Delta Lake sowie integrierten Funktionen wie ACID-Transaktionen bietet ein Data Lakehouse eine konsistente und effiziente Datenverarbeitung. Dies reduziert den Verwaltungsaufwand und vermeidet redundante Datenkopien, die bei der parallelen Nutzung von Data Lakes und Data Warehouses häufig auftreten (Armbrust et al, 2021).

Vor diesem Hintergrund stellen sich die zentralen Forschungsfragen dieser Arbeit:

1. Wie kann eine Data Lakehouse Architektur zur Optimierung bestehender Datenverarbeitungsprozesse entwickelt und implementiert werden?

2. Welche Vorteile und Herausforderungen ergeben sich aus der Implementierung eines Data Lakehouse im Vergleich zu bestehenden Datenarchitekturen?

Die Wahl eines Data Lakehouses als Thema dieser Arbeit basiert auf seiner Relevanz für die aktuellen Herausforderungen der OSG. Die bestehende Datenarchitektur der OSG, die auf einer Kombination aus FIDUS-Arztpraxensoftware, ETL-Systemen und HANA-Datenbanken basiert, ist wenig flexibel und schwer wartbar. Neue Datenquellen oder Anpassungen erfordern umfangreiche Eingriffe, was die Einführung moderner datengetriebe-

ner Anwendungen erschwert. Ein Data Lakehouse könnte eine skalierbare, wartbare und zukunftssichere Lösung darstellen, die sowohl die Datenintegration vereinfacht als auch die Grundlage für Machine-Learning Use-Cases bietet.

Das Ziel dieser Arbeit besteht darin, die prototypische Implementierung einer Data Lakehouse-Architektur für die spezifischen Anforderungen der OSG zu untersuchen. Dabei wird analysiert, wie ein solches System die aktuellen Schwachstellen der Dateninfrastruktur adressieren kann. Durch die Konzeption und Evaluierung eines Prototyps sollen die Potenziale und Grenzen eines Data Lakehouses im Vergleich zur bestehenden Architektur aufgezeigt werden. Diese Analyse wird durch eine qualitative Bewertung der Architektur hinsichtlich Skalierbarkeit, Erweiterbarkeit und Wartbarkeit unterstützt.

Zur Beantwortung der Forschungsfragen folgt diese Arbeit dem Design Science Research Ansatz (3.1). Dabei werden die bestehende Architektur analysiert, ein Prototyp entwickelt und anhand definierter Kriterien evaluiert. Die Evaluierung erfolgt qualitativ durch eine Bewertung der Wartbarkeit, Skalierbarkeit und Erweiterbarkeit.

Die Arbeit ist wie folgt gegliedert: In Kapitel 2 werden die theoretischen Grundlagen von Data Warehouses, Data Lakes und Data Lakehouses erläutert, einschließlich relevanter Technologien. Kapitel 3 beschreibt die methodische Vorgehensweise der Arbeit. Kapitel 4 analysiert die bestehende Dateninfrastruktur der OSG, identifiziert deren Herausforderungen und legt die Grundlage für die neue Architektur. In Kapitel 5 wird die Konzeption der Data-Lakehouse-Architektur detailliert beschrieben, einschließlich der gewählten Technologien. Die prototypische Implementierung wird in Kapitel 6 dokumentiert, wobei der technische Aufbau, die Datenpipeline und die eingesetzten Werkzeuge erläutert werden. Kapitel 7 enthält eine Evaluation des Prototyps anhand qualitativer Kriterien mit der bestehenden Architektur. In Kapitel 8 erfolgt eine Diskussion der Ergebnisse, einschließlich der Limitationen der Arbeit und möglicher zukünftiger Erweiterungen.

Thematische Abgrenzung. Die Bachelorarbeit konzentriert sich, innerhalb der Infrastruktur der OSG, ausschließlich auf die technische Sicht der Datenarchitektur rund um die Arztpraxisoftware FIDUS. Ziel ist es, die technischen Herausforderungen der bestehenden Architektur zu analysieren und eine optimierte Lösung zu entwickeln, die insbesondere Aspekte wie Datenintegration, Skalierbarkeit und Wartbarkeit adressiert.

Andere Systeme und Datenquellen werden nur insofern betrachtet, als sie für das Verständnis der übergeordneten Datenarchitektur relevant sind. Die Arbeit beschränkt sich dabei bewusst auf die technischen Aspekte und schließt wirtschaftliche oder betriebswirtschaftliche Analysen vollständig aus. Ebenso ist die Untersuchung auf die theoretische Analyse und einen prototypischen Entwurf begrenzt; eine vollständige Implementierung

der vorgeschlagenen Architektur ist nicht Bestandteil dieser Arbeit.

2 Theoretische Grundlagen

Um die Implementierung eines Data-Lakehouse-Prototyps fundiert zu begründen, werden in diesem Kapitel die theoretischen Grundlagen beschrieben, die für das Verständnis der Architektur und ihrer Kernkomponenten notwendig sind. Zunächst wird das Konzept klassischer Data Warehouses beschrieben, das traditionell für analytische Datenverarbeitung genutzt wird. Anschließend werden Data Lakes als Speicherlösungen für große, heterogene Datenmengen vorgestellt. Darauf aufbauend wird das Konzept des Data Lakehouse beschrieben, das die Stärken beider Ansätze kombiniert, indem es die strukturierte Verwaltung eines Data Warehouses mit der Flexibilität eines Data Lakes verbindet.

Neben den Architekturansätzen werden auch die relevanten Technologien betrachtet, die sowohl für die Umsetzung eines Data Lakehouse als auch im Kontext der internen OSG-Architektur von Bedeutung sind.

2.1 Data Warehouse

Unter den Begriff des Data Warehouse (Abbildung 1) wird ein System verstanden, das strukturierte Daten aus verschiedenen Quellen integriert, speichert und für Analysezwecke bereitstellt. Es wurde speziell entwickelt, um Unternehmen bei der Unterstützung von Geschäftsentscheidungen zu helfen, indem historische Daten gesammelt und für komplexe Abfragen sowie Berichte verfügbar gemacht werden. Die Architektur eines Data Warehouses basiert auf mehreren Schichten, die gemeinsam eine effiziente Datenverarbeitung und -speicherung ermöglichen (Vaisman und Zimányi, 2022, S. 4).

Die Datenverarbeitung beginnt mit dem, in der Back-End-Schicht angesiedelt, ETL-Prozess (Extract, Transform, Load). Zunächst werden Daten aus verschiedenen Quellen wie operativen Datenbanken oder externen Systemen extrahiert. Anschließend werden diese Daten transformiert, indem Fehler bereinigt, Formate vereinheitlicht und die Daten für das Data Warehouse aggregiert werden. Schließlich werden die transformierten Daten in die Datenbank geladen, wobei regelmäßige Aktualisierungen sicherstellen, dass die gespeicherten Informationen aktuell bleiben (Vaisman und Zimányi, 2022, S. 68-69).

Das Herzstück eines Data Warehouse Architektur bildet die Data-Warehouse-Schicht, die aus einem Enterprise Data Warehouse besteht. Dieses deckt die gesamte Organisation ab

und wird häufig durch spezialisierte Datenmarts ergänzt, die spezifische Anforderungen einzelner Abteilungen bedienen. Ein Metadata Repository speichert zusätzliche Informationen über die Struktur, Herkunft und Transformation der Daten, was die Nachvollziehbarkeit und Konsistenz der Daten gewährleistet (Vaisman und Zimányi, 2022, S. 69).

Die Daten werden schließlich über die Front-End-Schicht für Nutzer zugänglich gemacht. In dieser Schicht kommen Reporting- und OLAP-Tools (Online Analytical Processing) zum Einsatz, die multidimensionale Analysen ermöglichen. Diese Analysen erlauben es beispielsweise, Verkaufsdaten nach Zeit, Region oder Produktkategorien zu untersuchen. Vordefinierte Berichte und Dashboards unterstützen Unternehmen bei der regelmäßigen Überwachung und Analyse ihrer Geschäftsprozesse (Vaisman und Zimányi, 2022, S. 70).

Data Warehouses sind für traditionelle analytische Abfragen ausgelegt, jedoch weniger geeignet für moderne Anwendungsfälle wie maschinelles Lernen oder datengetriebene Vorhersagen. Diese Anwendungen erfordern oft den Zugriff auf große, unstrukturierte Datensätze und nicht-SQL-basierte Workloads, die ineffizient über klassische Warehouse-Schnittstellen wie JSON(JavaScript Object Notation) oder JDBC(Java Database Connectivity). verarbeitet werden können (Armbrust et al, 2021, S. 3).

Darüber hinaus stellt die Sicherstellung der Datenqualität und Konsistenz eine der größten Herausforderungen in traditionellen Data-Warehouse-Architekturen dar. Da Daten aus unterschiedlichen Quellen extrahiert und transformiert werden, können Abweichungen in unterstützten Datentypen, SQL-Dialekten oder Schemaformaten auftreten. Diese Unterschiede führen oft dazu, dass Fehler bei der Verarbeitung auftreten oder Daten fehlerhaft integriert werden. Zusätzlich erhöht die Vielzahl an ETL- und ELT-Prozessen die Wahrscheinlichkeit von Fehlern und Inkonsistenzen, die die Zuverlässigkeit der Daten nachhaltig beeinträchtigen können (Armbrust et al, 2021, S. 2).

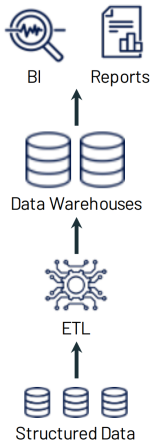


Abbildung 1: Platform der ersten Generation(Armbrust et al, 2021, S. 2)

2.2 Data Lake

Ein Data Lake ist ein moderner Ansatz zur Speicherung von Daten, der alle Arten von Daten, strukturiert, semi-strukturiert und unstrukturiert, in ihrer ursprünglichen Form in einer zentralen Plattform speichert. Diese Plattform wird als “flache Architektur” beschrieben, bei der die Daten mit Metadaten versehen und nach Bedarf abgerufen und verarbeitet werden. Die grundlegende Idee des Data Lakes ist es, große Mengen heterogener Daten an einem Ort zu sammeln und sie erst zum Zeitpunkt der Abfrage zu transformieren (Khine und Wang, 2018, S. 2).

Die Daten in einem Data Lake werden in Open-Source Dateiformaten wie Apache Parquet oder Apache ORC gespeichert und häufig in Cloud Object Stores oder verteilten Dateisystemen abgelegt (Armbrust et al, 2021, S. 15). Anders als beim Data Warehouse wird im Data Lake kein festes Schema verwendet, in das die Daten beim Laden transformiert werden müssen. Stattdessen werden die Daten in ihrer ursprünglichen Form gespeichert und erst bei der Analyse für spezifische Anwendungen transformiert. Dieser Ansatz, bekannt als Schema-on-Read (Khine und Wang, 2018, S. 3-4).

Ein zentraler Prozess im Data Lake ist der ELT-Prozess (Extract, Load, Transform). Daten werden zunächst in ihrer unverarbeiteten Form in den Data Lake geladen und nachträglich transformiert. Dies macht den Data Lake besonders geeignet für Anwendungen wie Machine Learning, bei denen Rohdaten benötigt werden, um Modelle zu trainieren oder explorative Analysen durchzuführen (Khine und Wang, 2018, S. 4).

Trotz ihrer Flexibilität und Kosteneffizienz bergen Data Lakes auch kritische Punkte. Eine der größten ist die Gefahr von Inkonsistenzen und Unübersichtlichkeit. Da Daten in ihrer unverarbeiteten Form gespeichert werden und keine einheitliche Struktur vorliegt, können leicht unorganisierte oder ungenaue Datenbestände entstehen. Ohne eine robuste Metadatenverwaltung besteht das Risiko, dass der Data Lake in einen sogenannten Data Swamp umschlägt. In einem Data Swamp ist es schwierig, relevante Daten zu finden oder sie effizient zu nutzen, da die Metadaten fehlen oder inkonsistent sind (Sawadogo und Darmont, 2021, S. 115).

Eine weitere Punkt ist der Mangel an Standardisierungen. Während der Begriff „Data Lake“ in der Literatur weit verbreitet ist, variieren die tatsächlichen Implementierungen erheblich. Es gibt keine einheitlichen Standards, wie ein Data Lake strukturiert oder verwaltet werden sollte, was zu Verwirrung und unterschiedlichen Ansätzen in der Praxis führen kann. Dies erschwert es Unternehmen, bewährte Verfahren zu etablieren und den vollen Nutzen aus der Technologie zu ziehen (Sawadogo und Darmont, 2021, S. 115).

Zusätzlich ist die Nutzung von Data Lakes oft mit einem hohen Bedarf an technischem Fachwissen verbunden. Im Gegensatz zu Data Warehouses, die auch von nicht-technischen Nutzern wie Business-Analysten verwendet werden können, sind Data Lakes komplexer in der Handhabung (Sawadogo und Darmont, 2021, S. 115).

2.3 Two-Tier Architektur

Die von Armbrust et al (2021) genannte Two-Tier-Architektur etablierte sich als Antwort auf die zunehmenden Herausforderungen tradionelle Data-Warehouse-Systeme im Umgang mit großen, vielfältigen und unstrukturierten Datenmengen. Während Data Warehouses ursprünglich für strukturierte, relationale Daten aus operativen Systemen konzipiert wurden, stiegen mit der wachsenden Anforderungen an Flexibilität, Skalierbarkeit und Kosteneffizienz erheblich. Unternehmen sahen sich zunehmend mit der Notwendigkeit konfrontiert, große Volumina an Rohdaten kostengünstig zu speichern und gleichzeitig leistungsstarke analytische Auswertungen durchführen zu können. Um diesen Ansprüchen gerecht zu werden, entwickelte sich die Two-Tier-Architektur als ein hybrides Modell, das die Vorteile von Data Lakes und Data Warehouses miteinander kombiniert (Armbrust et al, 2021, S. 1-2).

In diesem Modell (Abbildung 2) werden zunächst sämtliche Rohdaten, einschließlich strukturierter, semi-strukturierter und unstrukturierter Daten, in einem Data Lake gespeichert. Ein ausgewählter Teil dieser Daten wird anschließend über ETL-Prozesse in ein nachgelagertes Data Warehouse überführt. Dieses dient vor allem der Unterstützung von Business-Intelligence-Anwendungen und entscheidungsunterstützenden Systemen. In dieser Architektur wird somit der Data Lake als Rohdatenspeicherung genutzt, während das Data Warehouse die auswertbare Sicht auf die relevanten Daten bereitstellt.

Obwohl dieses Architekturmodell einige Probleme adressiert, identifiziert Armbrust et al (2021)(S. 1-2) auch neue Herausforderungen, die mit einer Two-Tier Architektur einhergehen:

Zuverlässigkeit. Die Aufrechterhaltung der Konsistenz zwischen Data Lake und Data Warehouse ist komplex und mit hohen Kosten verbunden, da kontinuierliche ETL-Prozesse notwendig sind. Dabei besteht das Risiko, dass Fehler oder Inkonsistenzen entstehen, die die Datenqualität beeinträchtigen.

Veraltete Daten. Im Data Warehouse stehen aktuelle Daten häufig erst mit erheblicher Verzögerung zur Verfügung, was dazu führt, dass Analysen oft auf einem veralteten Stand

basieren. Verglichen mit früheren Analysesystemen, bei denen operative Daten unmittelbar verfügbar waren, stellt dies einen deutlichen Rückschritt dar. Einer Umfrage zufolge nutzen 86% der Analysten nicht mehr aktuelle Daten, und 62% sind regelmäßig auf die Unterstützung technischer Teams angewiesen, um auf aktuelle Informationen zugreifen zu können.

Eingeschränkte Unterstützung fortgeschrittener Analytik. Viele Unternehmen verfolgen das Ziel, ihre im Data Warehouse gespeicherten Daten für prädiktive Analysen zu nutzen. Allerdings stoßen verbreitete Machine-Learning-Frameworks wie TensorFlow, PyTorch oder XGBoost dabei an Grenzen, da sie auf große Datenmengen und komplexe, nicht-SQL-basierte Verarbeitung angewiesen sind, was sich über herkömmliche Schnittstellen wie ODBC oder JDBC nur unzureichend abbilden lässt. Zwar lässt sich dieses Problem durch einen Export der Daten in "files"teilweise umgehen, jedoch führt dies zu zusätzlichem Aufwand und verzögert den Zugriff, während bei einer direkten Nutzung von Data-Lake-Daten zentrale Funktionen wie ACID-Unterstützung und Versionierung entfallen.

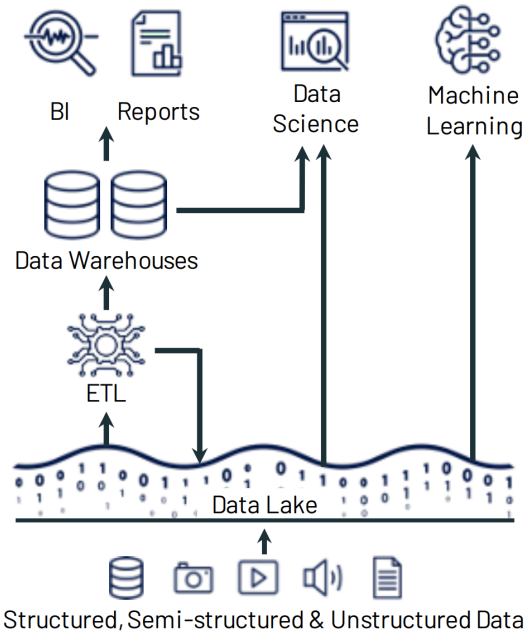


Abbildung 2: Aktuelle Two-Tier Architektur (Armbrust et al, 2021, S. 2)

2.4 Data Lakehouse

Das Data Lakehouse (Abbildung 3) stellt eine hybride Datenarchitektur dar, die entwickelt wurde, um die Stärken von Data Warehouses und Data Lakes zu kombinieren und

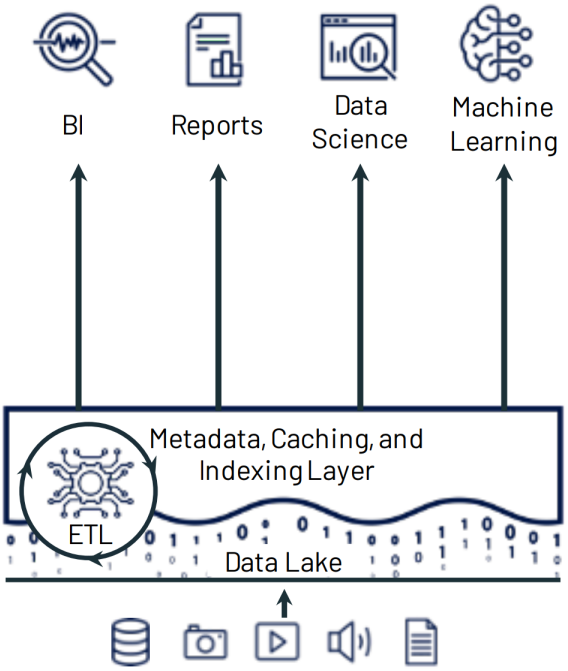


Abbildung 3: Lakehouse Platform (Armbrust et al, 2021, S. 2)

deren Schwächen auszugleichen. Es wurde speziell für die Anforderungen moderner datengetriebener Anwendungen konzipiert, die sowohl strukturierte als auch unstrukturierte Daten in einem zentralen System verarbeiten und analysieren möchten. Das Data Lakehouse integriert dabei die strukturierten Eigenschaften eines Data Warehouses mit der Flexibilität und Skalierbarkeit eines Data Lakes, wodurch eine leistungsstarke Plattform für unterschiedlichste Workloads geschaffen wird (Armbrust et al, 2021, S. 1).

Data Lakehouse zeichnet sich durch die Nutzung moderner Speicherformate wie Apache Parquet oder ORC aus. Diese Formate ermöglichen eine effiziente Speicherung unterschiedlichster Datentypen, einschließlich strukturierter und unstrukturierter Daten, und gewährleisten dabei eine hohe Interoperabilität zwischen verschiedenen Plattformen und Tools. Zu den zentralen Innovationen des Data Lakehouse Ansatzes zählt die möglichkeit, Transaktionen nach dem ACID-Prinzip (Atomicity, Consistency, Isolation, Durability) zu unterstützen. Diese Prinzipien stellen sicher, dass Datenbanktransaktionen zuverlässig und fehlerfrei ablaufen. Atomarität garantiert, dass eine Transaktion entweder vollständig ausgeführt wird oder gar nicht, während Konsistenz dafür sorgt, dass nach Abschluss einer Transaktion alle definierten Datenintegritätsregeln eingehalten werden. Isolation verhindert, dass parallele Transaktionen sich gegenseitig beeinflussen, und Dauerhaftigkeit sichert zu, dass einmal abgeschlossene Transaktionen auch bei Systemausfällen erhalten bleiben. Diese Eigenschaften sind entscheidend, um in verteilten Umgebungen, wie sie Delta Lake(2.5.2) nutzt, eine stabile und konsistente Datenverarbeitung zu gewährleis-

ten (Medjahed et al., 2009). Diese Kombination aus Speicher und Transaktionssicherheit macht das Data Lakehouse zu einer robusten Lösung für analytische Anwendungen, die auf konsistente und qualitativ hochwertige Daten angewiesen sind (Armbrust et al, 2021, S. 2-3).

Die Architektur des Data Lakehouse vereint Speicher- und Verarbeitungsschichten in einer einzigen Plattform, die durch Open-Source Standards ergänzt wird. Diese Integration ermöglicht es, Daten aus verschiedenen Quellen effizient zu speichern und zu verarbeiten, ohne dass komplexe Datentransformationen erforderlich sind. Ein vesehntlicher Bestandteil der Architektur ist das integrierte Metadatenmanagement. Dieses System sorgt dafür, dass sämtliche Daten unabhängig von ihrem Format oder ihrer Struktur organisiert und leicht zugänglich bleiben. Metadaten liefern Informationen über die gespeicherten Daten, wie deren Schema oder Eigenschaften, und erleichtern somit nicht nur die Verwaltung, sondern auch die Analyse und Abfrage großer Datenmengen.

Zu den weiteren Komponenten eines Data Lakehouse zählen Open-Source Speichertechnologien und einheitliche Programmierschnittstellen (APIs). Open-Source Speichertechnologien gewährleisten, dass Daten plattformunabhängig genutzt werden können, während die APIs sowohl SQL-basierte Abfragen als auch komplexere Workloads wie maschinelles Lernen oder Echtzeit-Analysen unterstützen. Dies erlaubt es, eine Vielzahl von Anwendungsfällen abzudecken, von klassischen Business-Intelligence-Fragen bis hin zu fortgeschrittenen datenwissenschaftlichen Modellen. Gleichzeitig sorgt das Metadatenmanagement für die Nachvollziehbarkeit und Konsistenz der Daten, was eine schnelle und präzise Datenverarbeitung ermöglicht (Armbrust et al, 2021, S. 4).

In Abbildung 4 wird ein grafischer Verlauf dargestellt, der veranschaulicht, wie die Datalakehouse-Architektur gemäß Serra (2024) aussehen könnte. „Store“ und „Transform“ können als Data-Warehouse innerhalb des Data Lakes betrachtet werden. (Serra, 2024, S. 22)

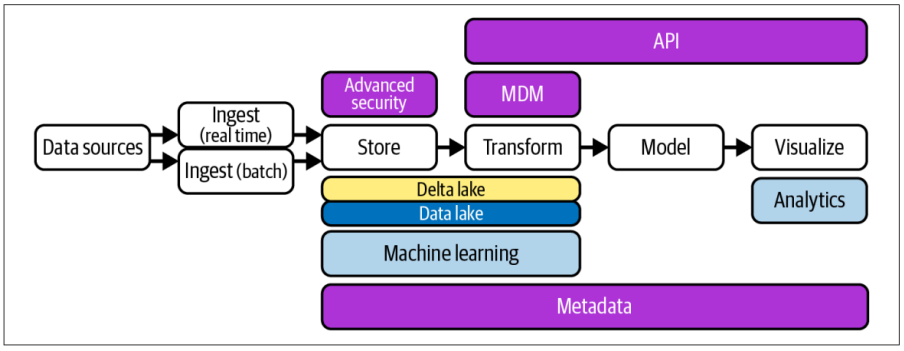


Abbildung 4: Data Lakehouse (Serra, 2024, S. 22)

2.5 Relevante Technologien

2.5.1 Apache Parquet

Apache Parquet verwendet eine spaltenorientierte Datenorganisation, bei der die Daten für jede Spalte separat gespeichert werden. Diese Struktur unterscheidet sich von zeilenorientierten Formaten, bei denen alle Werte einer Zeile zusammen abgespeichert werden. Die spaltenbasierte Speicherung hat den Vorteil, dass nur die benötigten Spalten für eine Abfrage gelesen werden müssen, was die Eeingabe/Ausgabe-Operationen reduziert und die Abfrageleistung signifikant verbessert. Dadurch eignet sich Parquet besonders gut für analytische Workloads, bei denen häufig aggregierte oder spezifische Spalten benötigt werden (Apache Parquet, 2025).

Ein entscheidender Punkt ist, dass Parquet als Standardformat verwendet wird, um Daten in einer Weise zu speichern, die nicht nur kosteneffizient ist, sondern auch eine einfache Integration mit einer Vielzahl von Tools ermöglicht, darunter SQL-Engines und Machine-Learning-Systeme. Durch die Open-Source Natur von Parquet können Daten direkt verarbeitet werden, ohne dass sie in proprietäre Formate umgewandelt werden müssen, was sowohl die Komplexität reduziert als auch die Flexibilität erhöht (Armbrust et al, 2021, S. 1-3)

Das Metadatenmanagement von Apache Parquet ist ein entscheidender Faktor, der schnelle Abfragen und eine reibungslose Verarbeitung der Daten ermöglicht. Jede Parquet-Datei enthält Metadaten auf verschiedenen Ebenen, darunter Dateimetadaten, Blockmetadaten und Spaltenstatistiken. Diese Metadaten beschreiben die Struktur und Eigenschaften der gespeicherten Daten, wie das Schema, die Datentypen, die Anzahl der Zeilen und Statistiken wie Min-, Max- und Durchschnittswerte für jede Spalte. Durch diese Informationen können Datenabfragen optimiert werden, da das System bereits vor dem Laden der Daten weiß, welche Blöcke oder Spalten für eine Abfrage relevant sind. (Apache Parquet, 2025)

2.5.2 Delta Lake

Um die zentralen Herausforderungen herkömmlicher Data-Lake-Architekturen zu bewältigen, wurde Delta Lake von Databricks als Open-Source-Speicherschicht entwickelt, die inzwischen in Apache Spark integriert ist. Es bietet ACID-Transaktionen sowie ein effizientes Metadatenmanagement und optimierte Zugriffsmethoden für tabellarische Daten, die in Cloud-Objektspeichern wie Amazon S3 oder Azure Blob Storage gespeichert werden. Diese Eigenschaften ermöglichen es, eine zuverlässige und konsistente Datenverarbeitung

in modernen verteilten Umgebungen sicherzustellen (Armbrust et al., 2020, S.3411-3412).

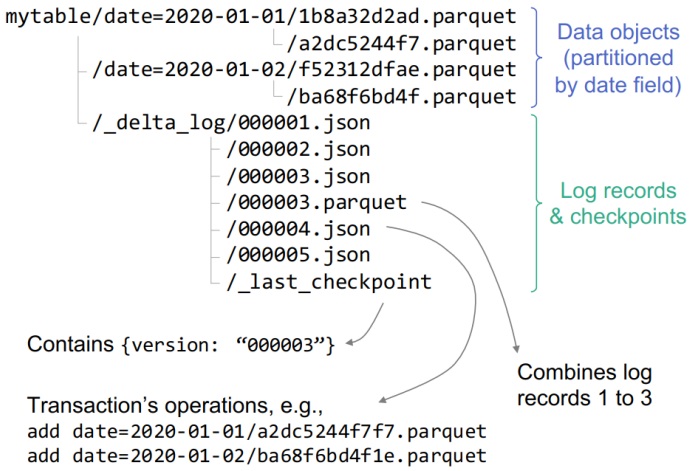


Abbildung 5: Beispiel eines Delta-Table-Logs (Armbrust et al., 2020, S. 3414)

Eine der zentralen Entscheidungen beim Design von Delta Lake ist die Verwendung des Speicherformats Apache Parquet, das eine effiziente, spaltenorientierte Ablage der Daten erlaubt. Diese Strukturierung ermöglicht es Query-Engines, gezielt nur die relevanten Spalten zu laden, wodurch die Ein- und Ausgabeoperationen reduziert und die Analysegeschwindigkeit erhöht werden. Da Parquet bereits in vielen modernen Datenverarbeitungssystemen weit verbreitet ist, profitieren bestehende Tools unmittelbar von dieser Kompatibilität. Dies erlaubt unter anderem die nahtlose Anbindung von Frameworks wie Apache Spark, Presto oder Hive, die mit Parquet-Dateien direkt arbeiten können.

Die zugrundeliegende Architektur von Delta Lake basiert auf einem unveränderbaren Transaktionsprotokoll, das sämtliche Änderungen an der Tabelle dokumentiert. Dieses sogenannte Delta-Log(Abbildung 5) wird in Form fortlaufend nummerierter JSON-Dateien gespeichert, wobei jede Datei eine Version des Tabellenzustands repräsentiert. Die Versionierung in Delta Lake ermöglicht es, frühere Zustände der Daten jederzeit wiederherzustellen. Um die Effizienz bei der Wiederherstellung früherer Versionen zu erhöhen, werden in regelmäßigen Abständen sogenannte Checkpoints erstellt. Dabei handelt es sich um Parquet-Dateien, die eine komprimierte Darstellung aller bis dahin gültigen Aktionen enthalten und somit eine schnellere Wiederherstellung erlauben(Armbrust et al., 2020, S. 3414-3415).

Beim Zugriff auf eine Tabelle prüft das System zunächst das Transaktionslog, um festzustellen, welche Datenobjekte aktuell zur Tabelle gehören. Hierbei wird auch auf Metadaten wie Spaltenstatistiken zurückgegriffen, etwa auf Minimal- und Maximalwerte einzelner Spalten. Diese Informationen erlauben ein sogenanntes Data Skipping, bei dem

irrelevante Dateien basierend auf Abfragebedingungen vorab ausgeschlossen werden. Eine zusätzliche Steigerung der Effizienz wird durch Partitionierung erreicht, bei der Daten nach bestimmten Attributen wie Zeitstempeln organisiert werden. Dadurch müssen bei selektiven Abfragen nur die relevanten Partitionen gelesen werden(Armbrust et al., 2020, S. 3421).

Darüber hinaus unterstützt Delta Lake auch Schema Evolution, sodass alte Parquet-Dateien gelesen werden können, ohne sie überschreiben zu müssen, wenn sich das Tabellenschema geändert hat. Zudem ermöglicht es UPSERT-, DELETE- und MERGE-Operationen, bei denen die relevanten Objekte effizient umgeschrieben werden, um Aktualisierungen archivierter Daten sowie die Umsetzung von Compliance-Workflows zu unterstützen(Armbrust et al., 2020, S. 3417-3419).

2.5.3 MinIO

MinIO ist eine Open-Source Object Storage-Lösung, die eine S3-kompatible Schnittstelle bietet. Es wurde entwickelt, um großen Speicherbedarf effizient und skalierbar zu bedienen(MinIO Inc., 2025). Minio ermöglicht es, Daten in Form von Objekten zu speichern. Dabei wird jede Dateneinheit als eigenständiges Paket abgelegt, das neben den eigentlichen Daten auch zugehörige Metadaten und einen eindeutigen Bezeichner enthält. Dies ist insbesondere in Cloud-Umgebungen von Vorteil, da es die Integration mit verschiedenen Analyse- und Verarbeitungswerkzeugen erleichtert (Gadban und Kunkel, 2021, S. 1-2).

Durch die Unterstützung der S3-API können bestehende Anwendungen, die für Amazon S3(Simple Storage Service) entwickelt wurden, ohne größere Anpassungen mit Minio betrieben werden. Dies erleichtert die Migration von Daten in private oder hybride Cloud-Umgebungen, in denen Kosten, Flexibilität und Kontrolle über die Daten im Vordergrund stehen. In einem Data Lakehouse dient Minio als zentraler Datenspeicher, in dem Rohdaten sowie verarbeitete Daten (zum Beispiel in Form von Delta Lake Tabellen) abgelegt werden. Die Open-Source Natur von Minio sowie seine einfache Konfiguration und Verwaltung machen es zu einer attraktiven Alternative zu nicht frei zugänglichen Cloud-Speicherlösungen (Gadban und Kunkel, 2021, S. 1-2).

2.5.4 Apache Spark

Apache Spark ist ein Open-Source-Framework zur verteilten Verarbeitung großer Datenmengen, das ursprünglich am Algorithms, Machines and People Lab der University of

California, Berkeley entwickelt wurde. Es wurde speziell für die Anforderungen moderner datenintensiver Anwendungen konzipiert und ermöglicht die parallele Datenverarbeitung in Cluster-Umgebungen. Spark unterstützt mehrere Programmiersprachen, darunter Java, Scala, Python und R, und bietet eine leistungsoptimierte Ausführungsengine auf Basis von allgemeinen Ausführungsgraphen (Directed Acyclic Graphs, DAGs)(Guo et al., 2018, S.2).

Ein zentrales Konzept von Spark ist das sogenannte Resilient Distributed Dataset (RDD), eine fehlertolerante, schreibgeschützte Sammlung von Objekten, die über verschiedene Knoten im Cluster partitioniert ist. RDDs können im Arbeitsspeicher gehalten werden, was eine sehr schnelle Verarbeitung ermöglicht. Bei Speicherengpässen werden sie jedoch automatisch auf die Festplatte ausgelagert. Die Wiederherstellung im Fehlerfall erfolgt über sogenannte Lineage-Informationen, die es ermöglichen, verlorene Partitionen auf Grundlage ihrer Ursprungsdaten neu zu berechnen. Spark unterscheidet dabei zwischen zwei Operationstypen: Transformations erzeugen neue RDDs, während Actions entweder ein Ergebnis zurückliefern oder Daten persistieren. Transformations sind „lazy“, das heißt, sie werden erst bei Ausführung einer Action tatsächlich berechnet. Je nach Transformationsart wird zwischen schmalen (narrow) und breiten (wide) Abhängigkeiten unterschieden, was Auswirkungen auf die Parallelisierbarkeit und Performance hat(Guo et al., 2018; Apache Software Foundation, 2025a, S.2).

Spark-Anwendungen laufen als eigenständige Prozesse auf einem Cluster. Der sogenannte SparkContext übernimmt dabei die Koordination zwischen dem Treiberprogramm und den Ausführungseinheiten, den sogenannten Executors. Die Anwendung kann entweder im Cluster-Modus oder im Client-Modus betrieben werden – je nachdem, ob der Treiberprozess direkt im Cluster oder auf der lokalen Maschine ausgeführt wird. Spark verwendet zur Ressourcenzuweisung einen Cluster Manager, der entweder Spark's interner Manager, Apache Mesos oder Hadoop YARN sein kann. Aufgaben (Tasks) werden vom Treiber an die Executor-Prozesse verteilt, welche die Verarbeitung übernehmen und Ergebnisse zurückliefern(Guo et al., 2018, S.2).

2.5.5 Trino

Trino, ehemals unter dem Namen PrestoSQL bekannt, ist eine Open-Source, verteilte SQL-Query-Engine, die speziell dafür entwickelt wurde, interaktive Abfragen über sehr große Datenmengen aus heterogenen Datenquellen hinweg zu ermöglichen. Im Kern basiert Trino auf einer Architektur, in der ein zentraler Koordinator Abfragen entgegennimmt, optimiert und in viele Teilaufgaben zerlegt, die dann parallel auf mehreren Worker-Knoten

ausgeführt werden. Diese verteilte Verarbeitung ermöglicht es, komplexe Analysen in Echtzeit durchzuführen, selbst wenn die Daten in unterschiedlichen Systemen wie HDFS, S3 oder relationalen Datenbanken gespeichert sind (Sethi et al., 2019).

Ein wesentlicher Vorteil von Trino liegt in seiner Fähigkeit, über einen einheitlichen SQL-Dialekt auf verschiedenste Datenquellen zuzugreifen. Diese Flexibilität erleichtert die Integration in bestehende Datenlandschaften erheblich, da Anwender ihre vertrauten SQL-Operationen beibehalten können. Darüber hinaus unterstützt Trino dynamisches Skalieren: Bei steigendem Abfragevolumen oder wachsender Datenmenge können zusätzliche Worker-Knoten hinzugefügt werden, um die Last zu verteilen und die Antwortzeiten kurz zu halten (Trino, 2025).

In Data-Lakehouse-Architekturen spielt Trino eine Rolle, indem es als Abfrageschicht fungiert, die Daten direkt aus einem Data Lake abruft, ohne dass diese Daten zuvor in ein klassisches Data Warehouse übertragen werden müssen. Die enge Zusammenarbeit mit Komponenten wie dem Hive Metastore sorgt zudem dafür, dass alle Anwendungen auf eine einheitliche, zentral verwaltete Metadaten zugreifen können. Somit wird eine konsistente Datenbasis sichergestellt, was insbesondere in unternehmenskritischen Anwendungen wichtig ist (Schneider et al., 2023, S. 47).

2.5.6 Apache Hive Metastore

Der Apache Hive Metastore stellt das zentrale Element zur Verwaltung von Metadaten in modernen Data-Lakehouse-Architekturen dar (Zhang et al., 2023, S. 3998). Er dient als zentraler Speicherort für alle relevanten Informationen zu Datenobjekten wie Tabellen, Schemata und Partitionen. Dadurch wird gewährleistet, dass verschiedene Datenverarbeitungs- und Analyse-Engines – wie Apache Spark oder Trino – auf eine einheitliche und konsistente Sicht der Daten zugreifen können (Apache Hive, 2025). Der Metastore operiert typischerweise als eigenständiger Dienst und speichert seine Daten in einer relationalen Datenbank wie PostgreSQL oder MySQL. Über das Thrift-Protokoll können Clients dynamisch auf diese Metadaten zugreifen, ohne dass sie in den einzelnen Anwendungen separat verwaltet werden müssen(Apache Hive, 2025).

Diese Verwaltung ist nicht nur für die Optimierung von Abfragen von Bedeutung, da sie es ermöglicht, nur die relevanten Partitionen zu laden und so die Abfrageleistung zu verbessern, sondern sie stellt auch sicher, dass Änderungen am Schema oder an den Partitionierungsstrategien zentral gesteuert und konsistent umgesetzt werden. Für das Datenmanagement ist der Hive Metastore besonders wertvoll, da er als gemeinsame In-

formationsquelle zwischen operativen Systemen und analytischen Anwendungen dient. So können Berichte, Dashboards und Ad-hoc-Abfragen auf einer einheitlichen, zentral verwalteten Datenbasis aufbauen, was die Datenqualität und -konsistenz erheblich verbessert. Insgesamt bildet der Apache Hive Metastore somit einen wichten Baustein, der durch seine zentrale Rolle in der Metadatenverwaltung maßgeblich zur Flexibilität und Skalierbarkeit moderner Datenarchitekturen beiträgt (Camacho-Rodríguez et al., 2019, 1775-1776).

2.5.7 PostgreSQL

PostgreSQL, häufig als „Postgres“ bezeichnet, ist ein relationales Datenbankmanagementsystem, das als Open-Source-Software zur Verfügung steht PostgreSQL Global Development Group (2024).

In modernen Datenarchitekturen, etwa in Data Lakehouse-Lösungen, spielt PostgreSQL eine Rolle im Metadatenmanagement. Häufig wird PostgreSQL als Backend für den Hive Metastore eingesetzt, wo es strukturierte Metadaten wie Tabellen-, Schema- und Partitionsinformationen speichert (Satyamurty und Murthy, 2018; Shi et al., 2022, S. 21, S. 3). Dank seiner robusten Transaktionsmechanismen können diese Metadaten konsistent und zuverlässig verwaltet werden, was entscheidend für die effiziente Zusammenarbeit verschiedener Datenverarbeitungskomponenten wie Apache Spark ist Databricks (2025b).

2.5.8 Docker

Docker ist eine Open-Source-Plattform zur Containerisierung, die es ermöglicht, Anwendungen und deren Abhängigkeiten in isolierten Umgebungen – sogenannten Containern – auszuführen. Container sind dabei wesentlich leichter als herkömmliche virtuelle Maschinen, da sie das Betriebssystem des Hostsystems gemeinsam nutzen und nur die für die Anwendung erforderlichen Bibliotheken und Konfigurationen enthalten. Dies sorgt dafür, dass Anwendungen plattformübergreifend konsistent laufen, sei es lokal, in der Cloud oder in Rechenzentren. Zu den Hauptfunktionen von Docker gehört die Möglichkeit, Software in eigenständigen Containern zu verpacken, was eine hohe Portabilität und Flexibilität bietet. Gleichzeitig trägt die gemeinsame Nutzung des Host-Betriebssystems zu einer effizienten Ressourcennutzung bei, da Docker-Container ressourcenschonender arbeiten als vollständige virtuelle Maschinen. Darüber hinaus ermöglicht Docker eine einfache Skalierung, indem Container-Instanzen je nach Bedarf schnell gestartet oder gestoppt werden können, und unterstützt die Versionierung von Anwendungen durch Docker-Images, die verschiedene Softwareversionen enthalten und eine reproduzierbare Umgebung garantie-

ren. Die Isolation der Container verbessert zudem die Sicherheit, da potenzielle Konflikte zwischen unterschiedlichen Anwendungen minimiert werden. Insgesamt vereinfacht Docker den Entwicklungs-, Bereitstellungs- und Verwaltungsprozess von Software erheblich(Docker Inc., 2025; Mindtwo GmbH, 2025).

2.5.9 FIDUS

FIDUS ist eine spezialisierte Praxissoftware, die seit über 30 Jahren exklusiv für Augenärzte entwickelt wird. Sie unterstützt sowohl Einzelpraxen als auch Gemeinschaftspraxen, Medizinische Versorgungszentren (MVZ) und Kliniken bei der effizienten Organisation ihrer Arbeitsabläufe. Die Software bietet verschiedene Kernfunktionen, darunter die Patientenverwaltung, die eine strukturierte Organisation von Patientendaten und -terminen ermöglicht, sowie einen integrierten Terminplaner, der ein übersichtliches Zeitmanagement und die Option zur Online-Terminbuchung umfasst (FIDUS GmbH, 2025).

2.5.10 Lobster_data

Lobster_data ist eine No-Code-basierte Software für Datenintegration innerhalb und zwischen Unternehmen. Sie unterstützt verschiedene Integrationsszenarien wie EAI (Enterprise Application Integration), EDI (Electronic Data Interchange), API-Management, Big Data, Data Fabric und IoT. Die Plattform ermöglicht die Verarbeitung aller gängigen Datenformate und -protokolle und bietet Funktionen wie Drag-and-Drop-Konfiguration, Überwachung und Echtzeit-Datenmanagement. Dies ermöglicht Unternehmen die Konsolidierung und den Austausch von Daten über unterschiedliche Systeme hinweg (Lobster GmbH, 2025).

2.5.11 SAP HANA

SAP HANA (High-performance Analytic Appliance) ist eine Multi-Model-Datenbank, die darauf ausgelegt ist, Daten direkt im Arbeitsspeicher (RAM) zu speichern und zu verarbeiten, anstatt auf herkömmliche Festplatten zurückzugreifen. Durch die spaltenorientierte Speicherung und die In-Memory-Technologie ermöglicht SAP HANA eine parallele Verarbeitung von komplexen Analysen und Transaktionen in einem einzigen System. Dies erlaubt Unternehmen, große Datenmengen nahezu in Echtzeit zu verarbeiten, Datenabfragen unmittelbar auszuführen und so datengetriebene Entscheidungen zu treffen.(SAP SE, 2025c).

Eine In-Memory-Datenbank (IMDB) speichert Daten vollständig im Arbeitsspeicher eines Computers und nicht auf herkömmlichen Festplatten oder Solid-State-Laufwerken (SSD). Während viele Datenbanken heutzutage In-Memory-Funktionen anbieten, basieren sie dennoch primär auf Festplattenspeicherung. SAP HANA hingegen wurde speziell entwickelt, um den Arbeitsspeicher als primäre Speicherebene zu nutzen, mit der Möglichkeit, bei Bedarf auf andere Speichermechanismen zuzugreifen. Dies ermöglicht eine ideale Balance zwischen Leistung und Kosten. Der direkte Zugriff auf Daten im RAM ist erheblich schneller als von Festplatten oder SSDs und führt zu Reaktionszeiten im Millisekundenbereich (Preuss, 2017, S. 63-65).

2.5.12 SAP BW/4 HANA

SAP BW/4HANA ist eine Data-Warehouse-Lösung, die speziell für die In-Memory-Datenbank SAP HANA entwickelt wurde. Sie kombiniert die Leistungsfähigkeit von SAP HANA mit einem vereinfachten Datenmodell und flexiblen Reporting-Tools, um Unternehmen die Verarbeitung und Analyse großer Datenmengen zu ermöglichen. Im Gegensatz zu traditionellen Data-Warehouse-Systemen bietet SAP BW/4HANA erweiterte Funktionen für die Datenmodellierung, Integration und Analyse, die den Anforderungen datengetriebener Unternehmen gerecht werden soll (SAP SE, 2025b).

2.5.13 SAP Analysis for Microsoft

SAP Analysis for Microsoft Office ist ein Add-In für Microsoft Excel und PowerPoint, das es Benutzern ermöglicht, multidimensionale Datenanalysen direkt in der vertrauten Office-Umgebung durchzuführen. Die Lösung integriert Daten aus verschiedenen Quellen, darunter SAP BW, SAP BW/4HANA, SAP HANA, SAP Analytics Cloud und SAP S/4HANA Cloud, und unterstützt Unternehmen dabei, Geschäftsentscheidungen zu treffen. Eine der Aufgaben von SAP Analysis for Microsoft Office ist die möglichkeit, komplexe Datenstrukturen mit OLAP-Funktionalitäten zu analysieren (SAP SE, 2025a).

3 Methodik

3.1 Design Science Research-Ansatz

Zur Umsetzung des Bachelorarbeit wird der Mixed-Method **Design Science Research (DSR)-Ansatz** verwendet, eine in der Wirtschaftsinformatik etablierte Forschungsmethode (Vom Brocke, 2020, S. 1-2), die sich auf die Entwicklung und Evaluation innovativer Artefakte zur Lösung praktischer Probleme konzentriert. Der Forschungsprozess folgt dem Modell von (Peffers et al., 2007, S. 57-60), welches in sechs aufeinanderfolgende Phasen unterteilt ist.

Die erste Phase des Forschungsprozesses umfasst die **Problemidentifikation und Zieldefinition**, in der die bestehende Dateninfrastruktur der OSG analysiert wird. Ziel ist es, Herausforderungen wie mangelnde Flexibilität und eingeschränkte Skalierbarkeit zu identifizieren. Im Anschluss werden konkrete Ziele für die neue Architektur definiert, darunter die Verbesserung der Datenverfügbarkeit, die Unterstützung analytischer Workloads und die Vereinfachung der Integrationsprozesse.

Darauf folgt die **Anforderungsanalyse**, bei der die funktionalen und nicht-funktionalen Anforderungen an das Data Lakehouse in enger Zusammenarbeit mit der OSG erfasst werden. Hierbei werden relevante Technologien wie Delta Lake, Apache Spark und Azure Databricks untersucht, um sicherzustellen, dass die definierten Anforderungen umgesetzt werden können.

In der Phase des **Designs und der Entwicklung** wird ein Architekturentwurf erstellt, der die wichtigsten Komponenten des Data Lakehouses umfasst. Dazu gehören unter anderem die Bereiche Data Ingestion, Storage, Processing und Visulization. Anschließend erfolgt die prototypische Implementierung einer minimal funktionsfähigen Version (Minimum Viable Product, MVP), um die grundlegenden Funktionen zu validieren.

Nach der Entwicklung des Prototyps wird in der **Demonstrationsphase** der praktische Nutzen anhand realer Daten aus den Praxis- und Verwaltungssystemen der OSG getestet. Hierbei werden verschiedene Testszenarien durchgeführt, um die Datenverarbeitung und die Workflows zu überprüfen.

Im Anschluss erfolgt die **Evaluation**, in der der entwickelte Prototyp anhand definierter Kriterien bewertet wird. Dazu gehören Aspekte wie Skalierbarkeit, Wartbarkeit und Erweiterbarkeit. Außerdem wird die neue Architektur mit der bestehenden Infrastruktur verglichen, um Verbesserungen und mögliche Schwachstellen zu identifizieren. Weil die

komplexen Strukturen der OSG einen fairen Vergleich erschweren, wurde der Performance-Vergleich bewusst ausgelassen, da keine gleiche Grundlage für beide Tests geschaffen werden konnte.

Abschließend folgt die Phase der **Kommunikation**, in der die Ergebnisse dokumentiert und Handlungsempfehlungen für eine mögliche produktive Implementierung bereitgestellt werden. Die Erkenntnisse werden den relevanten Stakeholdern präsentiert, um fundierte Entscheidungen über die weitere Umsetzung treffen zu können. Diese Phase entspricht diese vollständige Arbeit.

Wahl der Methode. Durch diesen strukturierten Ansatz stellt der Design Science Research-Ansatz sicher, dass die Implementierung des Data Lakehouse-Systems praxisnah, zielgerichtet und iterativ erfolgt. Die methodische Vorgehensweise ermöglicht es, innovative Lösungen zu entwickeln und gleichzeitig einen wissenschaftlichen Mehrwert zu generieren.

3.2 Leitfadengestüzte Experteninterview

3.2.1 Auswahl der Interviewpartner

Für diese Bachelorarbeit, in der es um den Vergleich verschiedener Datenarchitekturen geht, werden Interviewpartner ausgewählt, die über praktisches Wissen im Bereich Datenverarbeitung und -integration verfügen. Es kommt dabei nicht ausschließlich auf einen akademischen Hintergrund oder eine Führungsposition an, sondern vor allem auf die direkte Erfahrung im Umgang mit Datenpipelines und -systemen. Ziel ist es, unterschiedliche Perspektiven zu erfassen, indem Experten aus den Bereichen Datenarchitektur, Data Engineering sowie strategischer Datensteuerung und -governance befragt werden. So können sowohl operative als auch strategische Aspekte der aktuellen Datenarchitektur beleuchtet werden. Darüber hinaus dienen die Interviews der Problemidentifikation und Zieldefinition, indem zentrale Herausforderungen bestehender Datenarchitekturen ermittelt werden.

3.2.2 Erstellung des Interviewleitfadens

Der Interviewleitfaden wurde unter Berücksichtigung anerkannter Methoden der qualitativen Forschung erstellt. Als Grundlage dient beispielsweise die Methodik von Gläser und Laudel, die den Nutzen strukturierter, aber gleichzeitig offener Interviews zur Erfassung von Spezialwissen hervorhebt. Die Gestaltung eines Interviewleitfadens spielt eine zentrale

Rolle in der qualitativen Forschung, da er dazu beiträgt, eine gewisse Strukturierung in die Untersuchung und das dazugehörige Themenfeld zu bringen und als konkretes Hilfsmittel in der Erhebungssituation dient (Bogner, 2014, S. 17). Grundlage für die Gestaltung des Leitfadens für diese Arbeit ist zudem das methodische Vorgehen von (Berger-Grabner, 2022, S. 151-152). Sie betont, dass bei der Erstellung eines Leitfadens die Fragen typischerweise in mehrere Hauptthemenblöcke unterteilt werden, die jeweils aus ein bis drei Hauptfragen bestehen. In diesem Zusammenhang gliedert sich der vorliegende Leitfaden in folgende Themenblöcke:

Zunächst werden in der Einführung und im Abschnitt zur persönlichen Rolle Fragen gestellt, die die berufliche Position, die Hauptaufgaben und den persönlichen Bezug zur aktuellen Datenarchitektur erfassen.

Anschließend folgt der Block zum aktuellen Ist-Zustand der Datenarchitektur, in dem der Aufbau der bestehenden Architektur sowie die verwendeten Systeme und Technologien thematisiert werden.

Daraufhin werden im Block „Details zur Datapipeline“ Fragen zur Funktionsweise der Datapipeline, zu den eingesetzten Tools und zu den Prozessen der Datenaufnahme, -transformation und -speicherung gestellt.

Im darauffolgenden Themenblock „Herausforderungen und Optimierungsbedarf“ wird erfragt, welche Probleme in der aktuellen Pipeline bestehen und welche Anforderungen an einen Prototypen als Mindestanforderung gestellt werden.

Abschließend werden im Block „Ausblick und strategische Perspektiven“ Fragen zu geplanten Weiterentwicklungen und erwarteten Verbesserungen der Architektur gestellt.

Der Leitfaden bietet somit einen strukturierten Rahmen, der alle wichtigen Aspekte der Datenarchitektur abdeckt und zugleich Raum für individuelle Nachfragen lässt.

3.2.3 Durchführung der Experteninterviews

Die Interviews werden vorwiegend als persönliche Gespräche geführt, um einen lebendigen Austausch zu ermöglichen und detaillierte Einblicke in die Erfahrungen der Experten zu erhalten. Vor jedem Interview werden die Teilnehmer über den Zweck, den Ablauf sowie den Umgang mit Datenschutz und Anonymität informiert. Falls es organisatorisch notwendig ist, kann auch ein schriftlicher Fragebogen per E-Mail eingesetzt werden. Der Leitfaden dient dabei als Orientierungshilfe, die es erlaubt, flexibel auf den Gesprächsver-

lauf zu reagieren und bei Bedarf vertiefende Fragen zu stellen.

4 Analyse der bestehenden Architektur

In diesem Kapitel werden die Ergebnisse der Experteninterviews zusammengefasst, die einen Einblick in die bestehende Datenarchitektur der Ober Scharrer Gruppe (OSG) liefern. Zunächst wird ein Einblicke über die gesamte Infrastruktur gegeben, in der die zentralen Elemente und deren Zusammenspiel dargestellt werden. Während die grobe Architektur einen Überblick über alle beteiligten Komponenten bietet, wird im weiteren Verlauf des Kapitels insbesondere die FIDUS Pipeline untersucht. Die Erkenntnisse aus den Interviews mit Kramer (2025) und Müller (2025) bilden dabei die Grundlage für die Analyse, in der sowohl die Funktionsweise als auch die identifizierten Problembereiche der Pipeline erörtert werden.

Hintergrund des Unternehmens. Die Ober Scharrer Gruppe (OSG) betreibt über 140 Standorte und verarbeitet eine Vielzahl von Daten aus verschiedenen Praxis- und Verwaltungssystemen. Um diese große Menge an Informationen zu verwalten, setzt das Unternehmen auf eine Dateninfrastruktur, die eine Vielzahl an Datenpipelines umfasst. Diese Datenströme vereinheitlichen Informationen aus unterschiedlichen Quellen und Formaten, um eine konsistente und verlässliche Datenbasis für Analysen, Berichterstattung und Verwaltungszwecke zu schaffen. Die Architektur der OSG besteht aus verschiedenen Enterprise Resource Planning- und Business-Warehouse-Systemen sowie Cloud-Integrationen.

4.1 Hauptkomponenten der gesamten Architektur

Datenaufnahme: Die Daten werden aus über 140 verschiedenen Arztpraxen, Systemen und Standorten erfasst und abgerufen.

ERP-Systeme/SAP S/4HANA: Sie sind operative Systeme, die geschäftsrelevante Prozesse wie Rechnungswesen, Einkauf und Controlling steuern. Zur Unterstützung verschiedener Anforderungen gibt es mehrere SAP S/4HANA-Instanzen: Das Produktivsystem (S4P) gewährleistet den laufenden Geschäftsbetrieb, das Qualitätssicherungssystem (S4Q) dient zur Überprüfung von Daten und Prozessen, und das Entwicklungssystem (S4E) ermöglicht die Implementierung neuer Funktionalitäten. Diese Systemlandschaft stellt eine Weiterentwicklung der Geschäftsprozesse sicher (Müller, 2025).

Business Warehouse (BW)-Systeme(2.5.12): Sie dienen der Aggregation und Auf-

bereitung von Daten aus ERP-Systemen für analytische Zwecke. Es gibt verschiedene BW-Instanzen: BW-P für den produktiven Betrieb, BW-Q für Qualitätssicherung und BW-E für die Entwicklung. Diese Systeme ermöglichen Berichts- und Analysefunktionen zur Unterstützung von Unternehmensentscheidungen.

HANA-Datenbanken(2.5.11): Sie bilden die Grundlage für Business Warehouse (BW)-Umgebungen und ermöglichen eine Datenverarbeitung. Mit ihrer Echt-zeitverarbeitungs- und Analysefunktion unterstützen sie Unternehmen dabei, Berichte zu erstellen. Die In-Memory-Technologie sorgt für einen schnellen Datenzugriff und erlaubt eine unmittelbare Analyse.

Middleware-Systeme - Lobster(2.5.10): spielen eine Rolle bei der Integration unterschiedlicher Datenquellen, indem sie als Schnittstellen zwischen operativen und analytischen Systemen fungieren. Sie ermöglichen den Datentransfer und unterstützen die Kommunikation zwischen den Systemen, wodurch Daten in Echtzeit für weitere Analysen und Geschäftsprozesse bereitgestellt werden können.

Analysis for Office: Ist ein Add-In für Microsoft Excel und PowerPoint, mit dem Anwender direkt auf SAP BW- oder SAP-HANA-Daten zugreifen können. Auf diese Weise lassen sich Datenabfragen ausführen, Berichte erstellen und Analysen in der vertrauten Office-Umgebung durchführen.

4.2 FIDUS Data Pipeline

FIDUS(2.5.9) ist eines der größten Systeme, das von zahlreichen Augenarztpraxen genutzt wird, wobei alle Standorte aufgrund des einheitlichen Datenbankschemas identische Datenstrukturen verwenden. Dadurch können die an unterschiedlichen Standorten erfassten Daten ohne aufwendige transformationen zusammengeführt werden. Bei der Ober Scharrer Gruppe (OSG) bildet die FIDUS Data Pipeline einen zentralen Bestandteil der Datenaufnahme und dient als Grundlage für den prototypischen Aufbau einer Data-Lakehouse-Architektur im Rahmen dieser Bachelorarbeit.

Der Datenfluss (Abbildung 6) beginnt damit, dass das ETL-Tool Lobster mittels einer SQL-Abfrage Daten aus der FIDUS-Datenbank extrahiert. Bereits während des Abfragevorgangs werden erste Transformationen durchgeführt, um die Rohdaten in ein standardisiertes Format zu überführen. Anschließend erfolgt eine weitere Aufbereitung in Lobster, bevor die Daten nach dem Push-Prinzip aktiv an die HANA-Datenbank übertragen werden, ohne dass diese die Daten selbst anfordern muss.

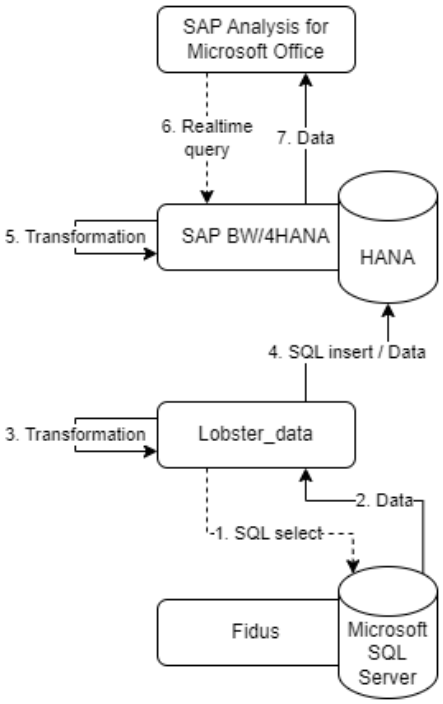


Abbildung 6: FIDUS Pipeline, Darstellung von Kramer (2025)

In der HANA-Datenbank werden die empfangenen Daten einer weiterführenden Transformation unterzogen, die darauf abzielt, verschiedene KPI(Key Performance Indicator)-Tabellen zu erstellen. So wird beispielsweise die KPI-10-Tabelle generiert durch ein SQL Query(Anhang B.1), die die Anzahl der Patienten pro Arzt ermittelt – ein entscheidender Indikator, um die Auslastung einzelner Praxen oder Ärzte zu überwachen. Ergänzend dazu werden weitere KPI-Tabellen erzeugt, darunter KPI-20, welche die Anzahl neuer Patienten darstellt, KPI-50, die verschiedene OP-Zahlen erfasst (unterteilt in Kategorien wie KAT-OPs, IVOMS, PPV und andere), sowie KPI-51, die die Anzahl der Zuweiser abbildet. Diese Kennzahlen liefern wesentliche Informationen für das Controlling und die Ressourcenplanung, da sie eine präzise Analyse der Prozessleistung ermöglichen.

Der nächste Schritt in der HANA Umgebung ist die zusätzliche Transformation, die in SAP BW erfolgt. Hier werden die in HANA erzeugten Tabellen durch externe Transformationen ergänzt, beispielsweise durch die Integration von Ladezeiten und Ladedatum in die KPI-Tabellen, sodass die Datenherkunft im Frontend nachvollziehbar wird und zeitbasierte Filterungen möglich sind. Zudem existieren in SAP BW unterschiedliche Layer(Sie Abbildung 9 und Abbildung 8), wobei jeder Layer seine eigene Transformation und presistente Speicherung durchläuft. In diesen Layern werden Daten aus verschiedensten Quellen, etwa von DrLib, diversen OSG Dashboards und weiteren internen Systemen, verarbeitet und in speziell dafür vorgesehene Zwischentabellen überführt. Vor der abschließenden

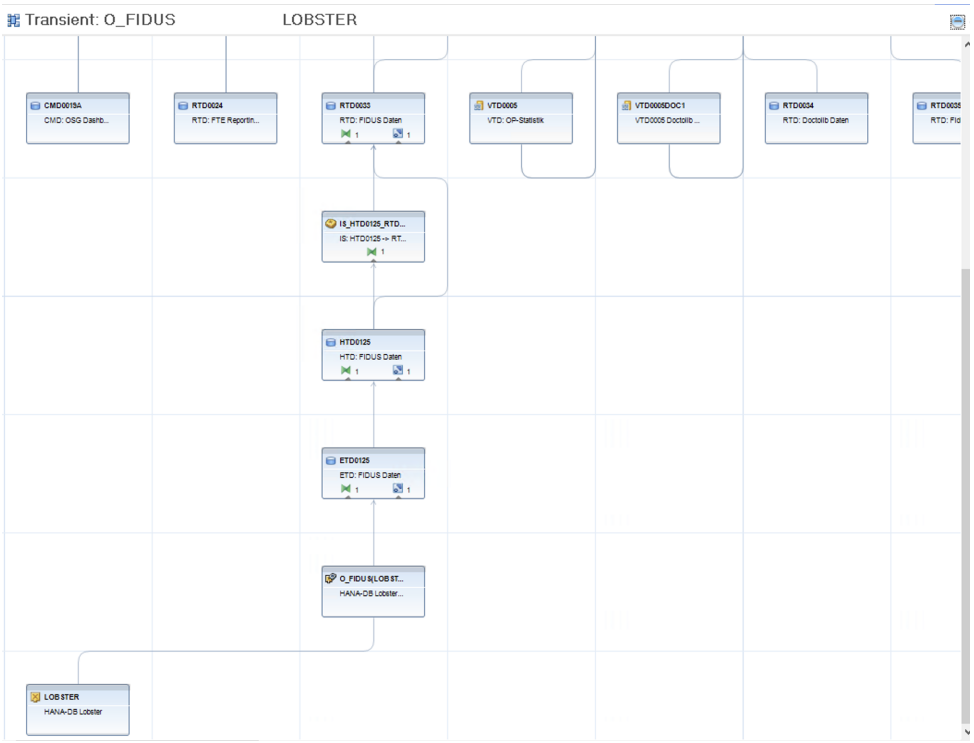


Abbildung 7: Screenshot der HANA BW Umgebung Part 1

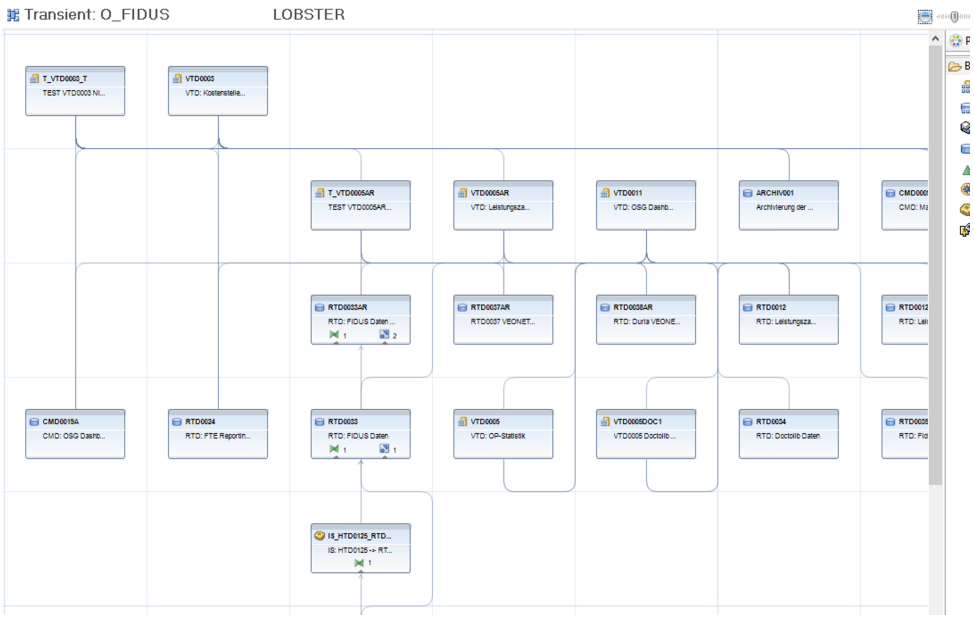


Abbildung 8: Screenshot der HANA BW Umgebung Part 2

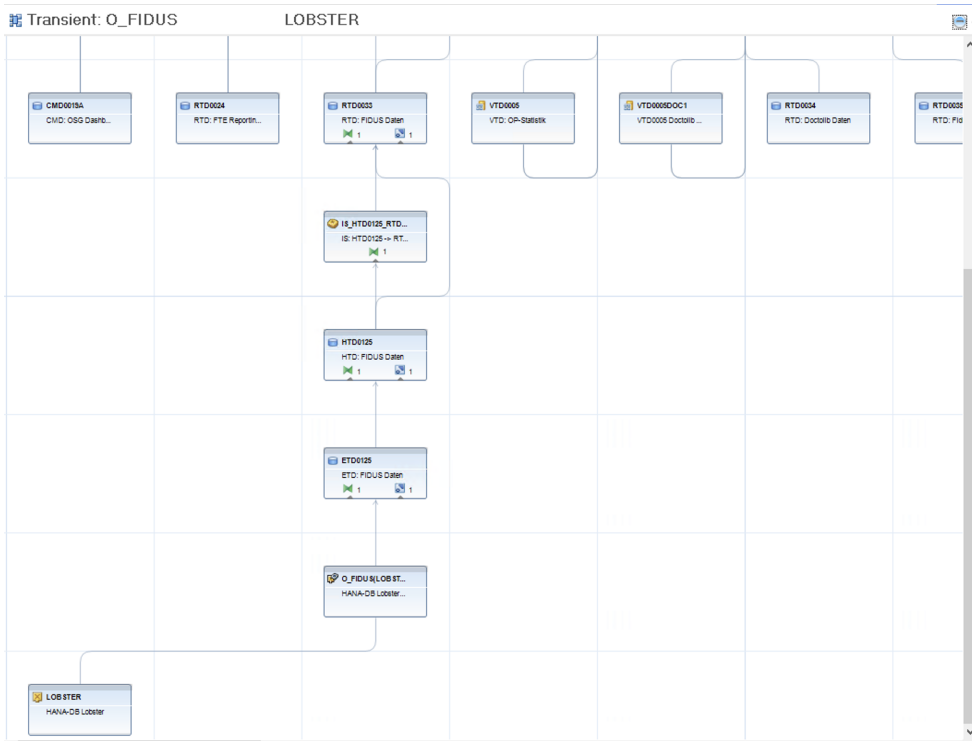


Abbildung 9: Screenshot der HANA BW Umgebung Part 1

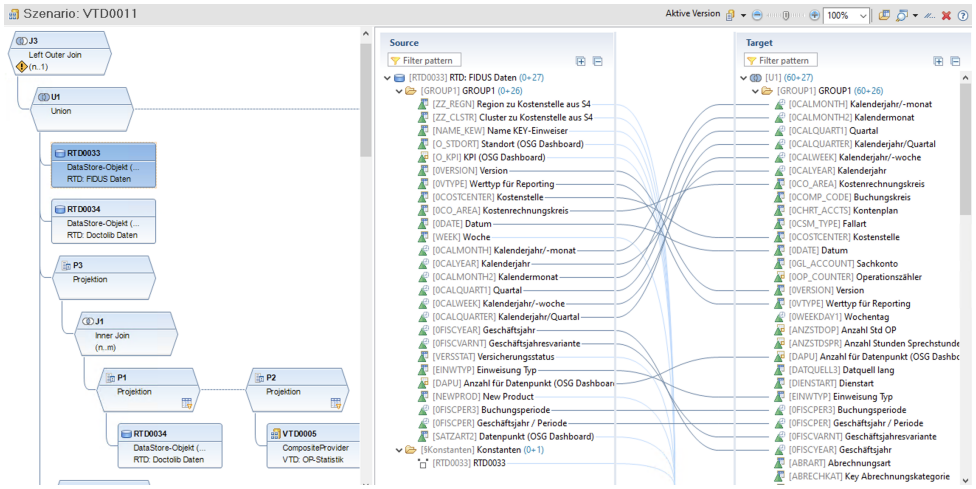


Abbildung 10: Screenshot der abschließender Union Operation

Join und Union Operation(Abbildung 10), die dazu dient, die finalen KPI-Tabellen zu erstellen, werden diese zahlreichen, teils sehr heterogenen und komplex miteinander verbundenen Tabellen weiter transformiert. In der Endphase fasst die Union Operation mehr als 20 Tabellen zusammen, darunter sowohl konsistente als auch virtuelle Tabellen, um dem Endnutzer ein umfassendes, integriertes Datenbild bereitzustellen.

Abschließend werden die Daten über SAP Analysis for Office den Endnutzern zur Visualisierung und Analyse zur Verfügung gestellt. Auf diese Weise können Berichte, Dashboards und andere Visualisierungen erstellt werden.

4.3 Problembereiche

Die aktuelle Architektur weist laut Müller (2025) und Kramer (2025) mehrere Herausforderungen auf, die sich auf die Effizienz und Flexibilität der Prozesse auswirken. Ein Problem ist die manuelle Natur einiger Prozesse. So erfolgt das Mapping der Daten über Excel-Dateien, die manuell in das System Lobster hochgeladen werden müssen. Dieser Ansatz erfordert nicht nur einen hohen personellen Aufwand, sondern birgt auch Fehlerpotenziale und erschwert eine Automatisierung der Abläufe. Allerdings wird dieser Aspekt im Prototypen nicht berücksichtigt, da es sich um einen externen Faktor handelt, auf den die OSG keinen Einfluss hat.

Ein weiteres wesentliches Hindernis ist die hohe Komplexität der Integration. Änderungen an bestehenden Datenquellen oder die Einführung neuer Datenquellen erfordern umfangreiche Anpassungen, Tests und Bugfixings entlang der gesamten Datenpipeline. Dies führt zu erhöhtem Aufwand und längeren Implementierungszeiten, was die Reaktionsfähigkeit der Architektur auf geschäftliche Anforderungen einschränkt.

Auch die Skalierbarkeit und Flexibilität der aktuellen Architektur stellen eine Herausforderung dar. Die Architektur ist nicht ausreichend anpassungsfähig, um mit wachsenden Datenmengen oder neuen Anforderungen Schritt zu halten. Dies führt dazu, dass das System schnell an seine Leistungsgrenzen stößt und die langfristige Nachhaltigkeit der Lösung infrage gestellt wird.

Zudem besteht eine erhebliche Abhängigkeit von Lobster und SAP BW/4HANA, die als primäre Transformationssysteme eine kritische Rolle in der Architektur spielen. Die Funktionsfähigkeit und Leistungsfähigkeit dieser Systeme beeinflussen direkt die gesamte Datenpipeline. Insbesondere die Performance von SAP BW/4HANA hat einen direkten Einfluss auf die Nutzererfahrung im Frontend, was sich in längeren Ladezeiten und einer

eingeschränkten Benutzerfreundlichkeit äußert.

Zusammenfassend lässt sich sagen, dass die derzeitige Architektur durch eine teils manuelle Abhängigkeit, eingeschränkte Flexibilität und Skalierbarkeit sowie durch kritische Systemabhängigkeiten gekennzeichnet ist. Eine Modernisierung und Automatisierung der Prozesse, durch ein Data Lakehouse Architektur(2.4) könnte dazu beitragen, diese Herausforderungen zu bewältigen und die Effizienz sowie die Anpassungsfähigkeit des Systems zu verbessern.

5 Konzeption der Data Lakehouse Architktur

5.1 Zielsetzung und Anforderungen

Im Rahmen dieser Bachelorarbeit wird ein Prototyp zur Umsetzung einer Data Lakehouse Architektur konzipiert, der einen Datenpipeline Durchlauf demonstrieren soll. Das zentrale Ziel besteht darin, durch die Implementierung der Kernprozesse, nämlich Datenaufnahme, Transformation, Speicherung und Visualisierung, einen funktionierenden, wenn auch minimalen, Datenverarbeitungsprozess zu realisieren. Dabei wird bewusst ein Minimal Viable Product (MVP) angestrebt, bei dem zunächst ausschließlich die essenziellen Funktionen umgesetzt werden Stevenson et al. (2024). Aspekte wie Compliance und Sicherheitsanforderungen werden aus Umfangsgründen vorerst außer Acht gelassen, um den Fokus auf die grundlegenden Abläufe zu legen.

Die Basis für den erfolgreichen Prototypenaufbau bilden die Anforderungen, die bereits im Rahmen von Experteninterviews (3.2) und der Analyse bestehender Datenarchitekturen(4) erhoben wurden. Anstelle einer umfassenden Anforderungsanalyse konzentriert sich diese Arbeit darauf, die aus den Experteninterviews gewonnenen Kernaussagen darzustellen. So konnten zentrale Anforderungen, wie beispielsweise die Anbindung an verschiedene Datenquellen, eine effiziente Datenaufbereitung sowie die strukturierte Speicherung der transformierten Daten, klar identifiziert und priorisiert werden. Zusätzlich war ein weiteres wichtiges Kriterium, dass die eingesetzten Produkte skalierbar, wartbar und erweiterbar sind – während die Performance zunächst weniger im Vordergrund steht – und dass die Datenaufnahme möglichst einfach gestaltet ist.

Die Zielsetzung des Prototyps basiert auf der Annahme, dass ein reibungsloser Datenfluss die Grundlage für weiterführende Analyse- und Reporting-Funktionen bildet. Um diesen Grundsatz umzusetzen, werden zunächst alle relevanten Prozesse erfasst, die für einen

erfolgreichen Durchlauf der Data-Pipeline notwendig sind. Hierzu zählen die Extraktion von Rohdaten aus unterschiedlichen Quellen, deren Aufbereitung und Transformation in ein nutzbares Format sowie die anschließende Speicherung in einer Umgebung, die eine effiziente Abfrage und Analyse ermöglicht.

Die Entscheidung, sich im Prototyp zunächst auf ein Minimum an Funktionalitäten zu konzentrieren, beruht auf dem MVP-Konzept Stevenson et al. (2024). Dieses Konzept zielt darauf ab, möglichst früh ein funktionsfähiges Produkt zu präsentieren, das die Kernanforderungen erfüllt, ohne durch zusätzliche, komplexe Features den Entwicklungsprozess zu verzögern. Dadurch wird sichergestellt, dass die grundlegenden Prozesse der Datenaufnahme, Transformation, Speicherung und Visualisierung im Mittelpunkt stehen, während spätere Erweiterungen flexibel integriert werden können. Für eine zielgerichtete Umsetzung wurden die Anforderungen dabei in zentrale Kategorien wie Datenintegration, Datenqualität und Performanz unterteilt, wobei in diesem Prototyp vor allem die Wartbarkeit, Skalierbarkeit und Erweiterbarkeit im Fokus stehen.

Ein weiterer wichtiger Aspekt der Zielsetzung ist die iterative Verfeinerung des Prototyps. Bereits in frühen Entwicklungsphasen werden erste Versionen erstellt, getestet und auf Basis des Nutzerfeedbacks sowie weiterer Erkenntnisse optimiert. Dieses iterative Vorgehen entspricht modernen agilen Entwicklungsmethoden, die sich als besonders effektiv bei der Entwicklung komplexer Systeme erwiesen haben (Marchesi et al., 2018, S. 2). Durch diesen Prozess wird gewährleistet, dass der Prototyp nicht nur die initial definierten Anforderungen erfüllt, sondern auch flexibel an neue Erkenntnisse und zukünftige Erweiterungen angepasst werden kann.

5.2 Architekturkonzept

Das in dieser Arbeit vorgestellte Architekturkonzept orientiert sich an dem in Deciphering Data Architecture beschriebenen Modell von Serra (2024) (Abbildung 11), das zugleich wesentliche Ideen von Armbrust et al. (2021) aufgreift. Dieses Modell verdeutlicht, wie Daten aus unterschiedlichsten Quellen, von unstrukturierten Formaten wie Bildern, Audiodateien und Freitexten über semistrukturierte Datensätze bis hin zu relationalen Datenbanken und Echtzeit-Streams, in einer einzigen Plattform zusammengeführt werden können. Dabei werden die Daten in mehreren Schritten verarbeitet, wobei zunächst eine „Raw Zone“ für die Ablage der Rohdaten in ihrem ursprünglichen Format sorgt, bevor eine schrittweise Bereinigung und Anreicherung in einer „Cleaned Zone“ erfolgt. In einer nachfolgenden „Presentation Zone“ werden die Daten schließlich eingepflegt und in einer Form abgelegt, die sich für klassische Reporting- und Analysezwecke eignet. Diese mehrstufige

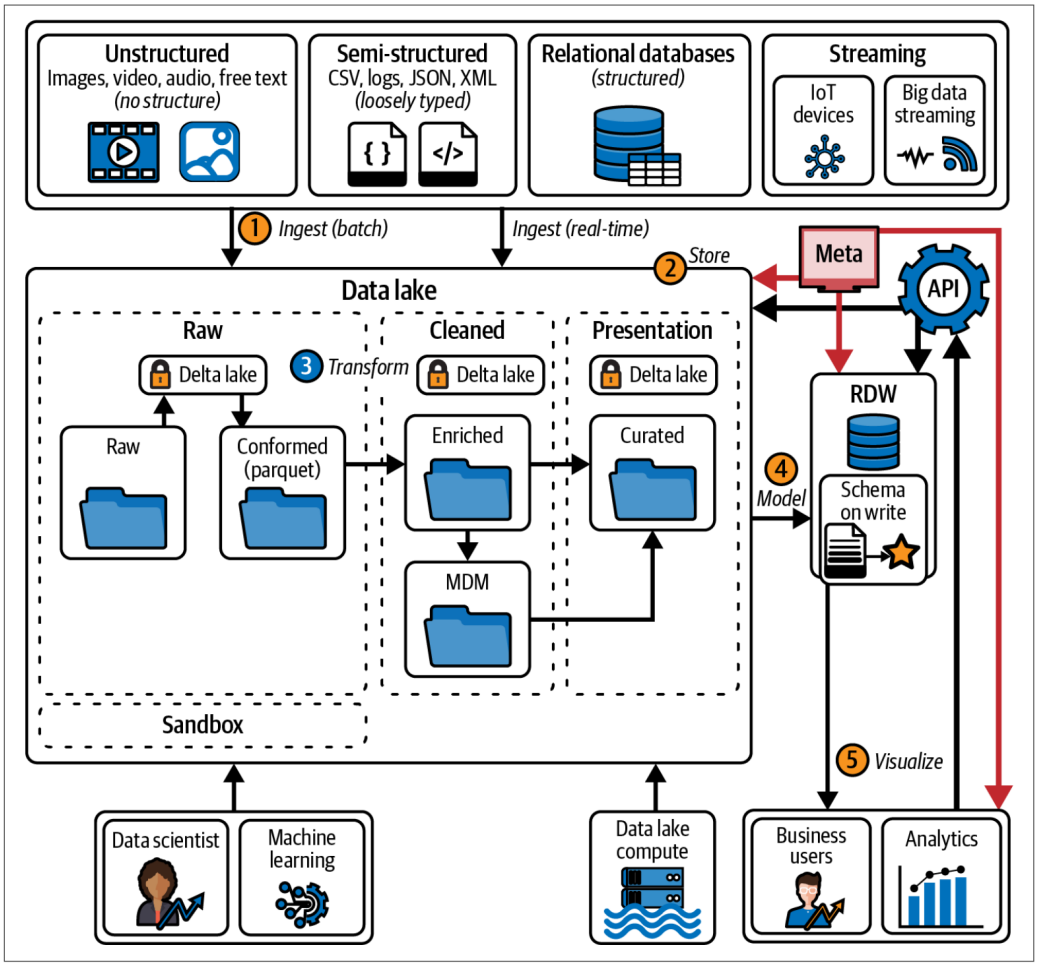


Abbildung 11: Data Lakehouse Architektur nach (Serra, 2024, S. 163)

Vorgehensweise erleichtert nicht nur die Integration verschiedener Datentypen, sondern erlaubt es auch, unterschiedliche Workloads, etwa maschinelles Lernen oder Business Intelligence, auf einer einheitlichen Plattform zu betreiben. Dieses Vorgehen, auch bekannt als Medallion-Architektur, unterteilt die Datenverarbeitung, gemäß den Empfehlungen von Databricks (2025a), in die Stufen Bronze (Rohdaten), Silver(bereinigte und angepasste Daten) und Gold(kuratierte Tabellen auf Unternehmensebene), und wird auch in der in dieser Arbeit konzipierten Lakehouse-Architektur angewendet.

Ein zentrales Element dieses Ansatzes bildet das Metadatenmanagement, das in vielen Implementierungen durch einen Metastore realisiert wird. Über diesen Metastore werden Struktur- und Schemainformationen zu allen in den verschiedenen Zonen abgelegten Daten verwaltet, was eine einheitliche Sicht auf den Datenbestand sicherstellt und Abfragen deutlich vereinfacht.

Nach Serra (2024) und Armbrust et al (2021) entsteht durch diese Kombination aus Data-Lake-Flexibilität und Data-Warehouse-Struktur eine Plattform, die sowohl für explorative

Datenanalysen als auch für klassische, strukturierte Berichte ausgelegt ist. Anders als bei einer rein zweischichtigen Architektur, bei der Data Lake und Data Warehouse strikt getrennt bleiben, ermöglicht das Lakehouse-Prinzip eine zentrale Verwaltung und Verarbeitung sämtlicher Datentypen. Dies senkt den Integrationsaufwand und reduziert die Gefahr von Inkonsistenzen, da Daten nicht mehr mehrfach in unterschiedlichen Umgebungen vorgehalten werden müssen. Stattdessen werden sie in einer einheitlichen Speicher- und Metadatenstruktur gepflegt, was die Effizienz und Nachvollziehbarkeit aller Verarbeitungsschritte erhöht.

5.3 Auswahl der Technologien

Neben den in dieser Arbeit verwendeten Open-Source-Technologien existieren auch umfassende kommerzielle Lösungen, die von Großunternehmen wie Google, Microsoft, Amazon Web Service oder Databricks angeboten werden. Diese Produkte – wie Google Cloud BigQuery, Microsoft Azure Synapse Analytics oder die Angebote von Databricks – bieten oftmals erweiterte Funktionen, hochgradige Skalierbarkeit, integrierte Sicherheitsfeatures und eine umfassende Verwaltung von Datenprozessen. Allerdings sind diese Systeme in der Regel mit erheblichen Kosten verbunden, die sich insbesondere bei großen Datenvolumen oder komplexen Workloads schnell summieren können (Fraser, 2022).

Für diese Arbeit ist der Zugang zu solchen kommerziellen Lösungen häufig aus Budget- und Zeitlichengründen nicht realisierbar. Zudem erfordern diese Plattformen häufig einen erheblichen administrativen Aufwand, weshalb sie in dieser Arbeit weniger geeignet sind. Im Rahmen des Prototyps wurde deswegen bewusst auf Open-Source-Technologien zurückgegriffen, um sowohl das begrenzte Budget zu schonen als auch eine einfache Implementierung, spätere Erweiterbarkeit sowie eine hohe Skalierbarkeit und Wartbarkeit zu gewährleisten. Die ausgewählten Produkte bieten eine breite Community-Unterstützung, umfangreiche Dokumentation und ermöglichen es, innovative Funktionen Lokal zu realisieren.

Für die flexible und skalierbare Speicherung der Daten wurde MinIO (2.5.3) eingesetzt. MinIO ist eine S3-kompatible Object-Storage-Lösung, die es ermöglicht, große Mengen an Rohdaten sowie transformierte Daten effizient abzulegen, ohne an feste Skalierungsgrenzen zu stoßen.

Das zentrale Metadatenmanagement erfolgt über den Hive Metastore (2.5.6), der als einheitliche Verwaltungseinheit für alle Informationen zu Tabellen, Schemata und Partitionen dient. Als Backend für den Hive Metastore kommt PostgreSQL (2.5.7) zum Einsatz. Ein

bewährtes, stabiles und leistungsfähiges relationales Datenbanksystem, das durch seine Open-Source-Natur und umfangreiche Community besticht.

Für die Datenverarbeitung und Transformation setzt die Architektur auf Apache Spark (2.5.4). Diese verteilte Engine ermöglicht die effiziente Verarbeitung großer Datenmengen im Batch- sowie Streaming-Modus und unterstützt komplexe Transformationsprozesse, die für die Aufbereitung der Rohdaten notwendig sind. Ergänzend dazu wird Delta Lake (2.5.2) als zusätzliche Schicht eingesetzt, um ACID-Transaktionen, Versionierung und ein optimiertes Metadatenmanagement zu gewährleisten. Delta Lake arbeitet dabei auf Basis von Apache Parquet und ermöglicht so eine zuverlässige Datenverarbeitung, indem es inkrementelle Updates, Rollbacks und Time-Travel-Funktionalitäten unterstützt.

Um den Zugriff auf die in der Storage-Schicht abgelegten Daten mittels standardisierter SQL-Abfragen zu ermöglichen, wurde Trino (2.5.5) als verteilte SQL-Query-Engine gewählt. Trino erlaubt es, heterogene Datenquellen über einen einheitlichen SQL-Dialekt abzufragen und unterstützt so sowohl interaktive Analysen als auch komplexe Reporting-Anwendungen.

Für die Visualisierung und das Reporting kommt Apache Superset zum Einsatz. Diese Open-Source-BI-Plattform ermöglicht die Erstellung interaktiver Dashboards und Berichte, die sowohl explorative Datenanalysen als auch standardisierte Reporting-Formate unterstützen – und das ohne zusätzliche Lizenzkosten (Apache Software Foundation, 2025b).

Abgerundet wird das Technologie-Setup durch Docker, ein Containerisierungstool, das die Bereitstellung und Verwaltung der einzelnen Komponenten erheblich vereinfacht. Durch Docker können alle ausgewählten Technologien in isolierten Containern betrieben werden, was zu einer hohen Portabilität, einfachen Skalierung und einer konsistenten Entwicklungsumgebung führt (Docker Inc., 2025).

Zusammengefasst basiert der Prototyp der Data Lakehouse-Architektur (Abbildung 12) auf den folgenden Open-Source-Technologien: MinIO für den Object Storage, Hive Metastore in Kombination mit PostgreSQL für das Metadatenmanagement, Apache Spark für die Datenverarbeitung, Trino für SQL-Abfragen, Apache Superset für die Visualisierung sowie Docker zur Containerisierung. Diese Auswahl ermöglicht eine kosteneffiziente, robuste und leicht erweiterbare Lösung, die sich optimal für die iterative Entwicklung und den Proof-of-Concept eines Data Lakehouse-Prototyps eignet.

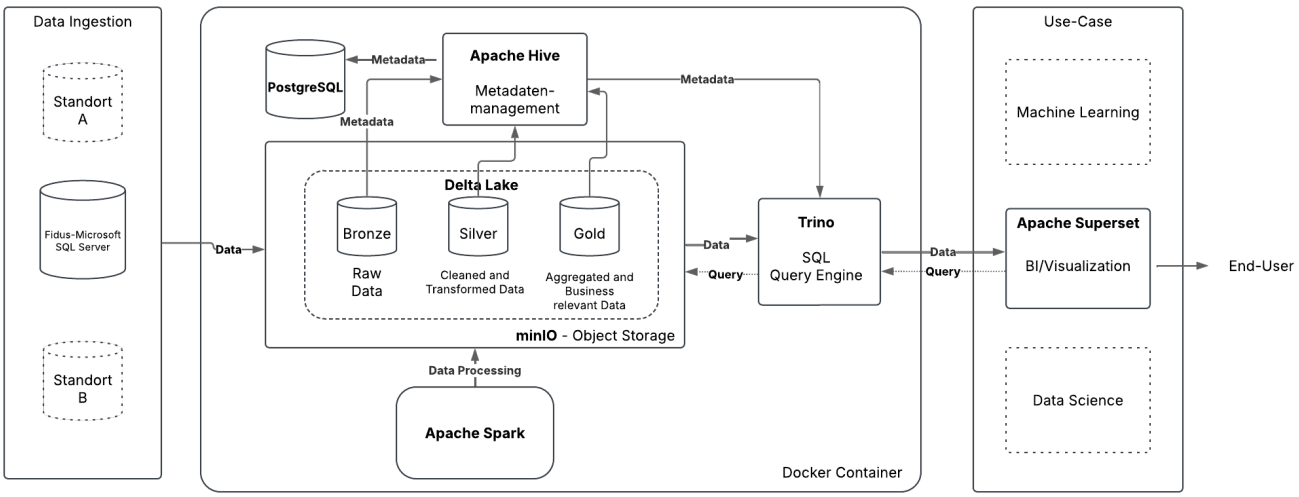


Abbildung 12: Data Lakehouse Architektur für die FIDUS Pipeline. Eigene Darstellung basierend auf (Serra, 2024)

6 Implementierung des Prototyps

In diesem Kapitel wird beschrieben, wie aus dem konzeptionellen Entwurf (Siehe Abbildung 12 und Anhang C) eine lauffähige Data-Lakehouse-Architektur entwickelt wurde. Zunächst wurde ein Docker Compose-Setup erstellt(Anhang D.1), das sämtliche ausgewählte Open-Source-Komponenten integriert. Konkret umfasst dies Container für Apache Spark, Trino, MinIO, den Hive Metastore sowie PostgreSQL. Hierfür wurden entsprechende Dockerfiles und Entrypoints für jeden Container entwickelt. Die Details und Konfigurationen, beispielsweise die Standardkonfiguration von Spark sowie die Konfigurationen von Trino und dem Hive Metastore, sind in Anhang D dokumentiert.

6.1 Hardware Umgebung

Zur Nachvollziehbarkeit der Hardware Umgebung, werden hier die Hardwarekomponenten aufgeführt, auf denen der Prototyp entwickelt und getestet wurde, da sämtliche Hardwarekonfigurationen von Spark auf diesem Gerät basieren. Dabei kam ein MacBook Pro aus dem Jahr 2021 zum Einsatz. Die nachstehende Tabelle bietet eine Übersicht über die relevanten Hardwarespezifikationen des Geräts.

Um auf dem genannten MacBook Pro optimale Spark-Einstellungen zu erzielen, empfiehlt es sich, die verfügbaren Ressourcen gleichmäßig auf zwei simulierte Worker zu verteilen, sofern mehrere Worker eingesetzt werden sollen. Jeder Worker könnte dabei etwa 5 Kerne

Komponente	Spezifikation
Modell	MacBook Pro 16"(2021)
Prozessor	Apple M1 Pro 10-Core CPU
RAM	16 GB
Speicher	512 GB
SSD-Geschwindigkeit	Bis zu 7,4 GB/s

Tabelle 1: Technische Spezifikationen des MacBook Pro 16"(2021) mit M1-Chip (Apple Support, 2025)

und 8 GB RAM erhalten, während zusätzliche Kerne und Speicherplatz für den Spark-Treiber sowie das Betriebssystem reserviert bleiben (Chao et al., 2018, S. 60). Innerhalb jedes Workers können dann mehrere Executor-Prozesse konfiguriert werden – beispielsweise zwei Executor mit jeweils 2 Kernen, was insgesamt 4 Kerne pro Worker für parallele Task-Ausführungen bereitstellt. Eine Zuweisung von rund 3,5 bis 4 GB RAM pro Executor stellt sicher, dass ausreichend Speicher für Verarbeitung, Caching und Shuffle-Operationen vorhanden ist, ohne die Gesamtressourcen zu überlasten (Chao et al., 2018, S. 60-61). Diese Konfigurationen zielen darauf ab, den Grad der Parallelität im Spark-Cluster optimal auszunutzen, während gleichzeitig ein stabiler Betrieb des Treibers und des Betriebssystems gewährleistet wird. Als Ausgangspunkt können diese Werte dienen, die je nach konkreter Workload weiter feinjustiert werden können, um die Leistung zu maximieren (Chao et al., 2018, S. 62).

6.2 Pipeline

Der nächste Schritt bestand darin, die Fertigarchitektur an die bestehende Data Pipeline anzupassen. Hierzu wurde zunächst ein Simulationsszenario implementiert, in dem Daten aus einer SQL-Datenbank gezogen werden, um den Ablauf der Pipeline nachzubauen. Die Testdaten bestehen aus 8 FIDUS-Datenbanken, die jeweils einem unterschiedlichen Standort zugeordnet sind und jeweils 10 Tabellen enthalten und der Gesamtvolumen beträgt 8 GB.

Anschließend wurde die spezifische Struktur der FIDUS-Datenbank berücksichtigt: Obwohl jeder Standort eine eigene Datenbank besitzt, verwenden alle dasselbe Schema. Dank dieser Homogenität konnten sämtliche Daten aus den verschiedenen Standorten per Ünionßusammengeführt bereits vorher, als Delta Tables im Bronze Layer gespeichert und weiterverarbeitet werden. Dieser Schritt dient der ersten Persistierung der Daten ohne weitere Verarbeitung.

Im Silber Layer erfolgt schließlich die Transformation der zusammengeführten Daten in spezifische KPI-Tabellen, beispielsweise zur Ermittlung der Anzahl von Patienten pro Arzt (KPI 10) sowie weiterer Kennzahlen wie KPI 20 (neue Patienten), KPI 50 (OP Zahlen in unterschiedlichen Kategorien) und KPI 51 (Zuweiser). Da die in der ursprünglichen FIDUS Pipeline verwendete SQL-Abfrage Microsoft SQL unterstützt und Spark diese Funktionalität nicht direkt bietet, muss sie mit Spark SQL umgeschrieben werden (Anhang D.5.2).

Im Gold Layer werden die präsentierfähigen Tabellen integriert, beispielsweise die KPI 10 Tabellen mit Ladedatum und Ladezeit. Die anderen KPI-Tabellen würden zusätzliche Transformationen erfordern, wie sie in der aktuellen FIDUS-Pipeline(4.2) beschrieben sind. Da auf diese Transformationslogiken jedoch kein Zugang besteht, wird im Rahmen dieses Minimal Viable Product (MVP) ausschließlich KPI 10 implementiert, um einen demonstrativen Durchlauf der Datenpipeline zu ermöglichen.

Zur Validierung der korrekten Integration wird überprüft, ob die bisher implementierten Komponenten, insbesondere der Hive Metastore, Delta Lake, Minio, Trino und Superset, korrekt zusammenarbeiten. Dabei sendet Superset eine SQL Abfrage an Trino, das diese Abfrage unter Zuhilfenahme der im Hive Metastore hinterlegten Metadaten an Delta Lake weiterleitet, in welchem die Daten auf Minio gespeichert sind. Da Delta Lake selbst keine SQL Schnittstelle bietet, bestätigt dieser Ablauf, dass der Hive Metastore die Metadaten ordnungsgemäß an Trino übermittelt hat, sodass Trino in der Lage ist, die auf Minio abgelegten Daten zu lesen. Letztlich werden die Daten als Tabelle in Superset angezeigt, was den erfolgreichen Durchlauf der Datenpipeline und die korrekte Transformation der Daten belegt.

7 Evaluation der Architekturen

Aufgrund der komplexen Struktur der bestehenden FIDUS-Datenpipeline, deren umfangreiche Transformationen und externen Systemabhängigkeiten nicht eins zu eins replizierbar sind, wird in dieser Arbeit eine qualitative Evaluation durchgeführt. Diese stützt sich auf bewährte Kriterien aus der Literatur zu modernen Datenarchitekturen wie Wartbarkeit, Skalierbarkeit, Erweiterbarkeit und Interoperabilität (Masak, 2010, S. 10). Im Folgenden werden diese Kriterien detailliert erörtert und auf den entwickelten Prototyp angewendet.

7.1 Wartbarkeit

Die Wartbarkeit einer Datenarchitektur beschreibt die Einfachheit, mit der das System gepflegt, Fehler identifiziert und behoben sowie Anpassungen durchgeführt werden können (Heitlager et al., 2007, S. 1). Die bestehende FIDUS-Architektur zeichnet sich durch eine historisch gewachsene Komplexität aus, in der mehrere Technologien (Lobster, SAP HANA, SAP BW) eng verzahnt miteinander arbeiten. Diese Struktur führt zu einem erheblichen Wartungsaufwand, da eine Vielzahl an Datenquellen, heterogenen Transformationen und umfangreichen Union-Operationen zu einer hohen technischen Schuld führen, was wiederum die Fehleranfälligkeit erhöht und Anpassungen erschwert.

Im Gegensatz dazu wurde der entwickelte Data-Lakehouse-Prototyp bewusst modular und klar strukturiert gestaltet. Durch den Einsatz von containerisierten Anwendungen wie Apache Spark, Trino, MinIO, Hive Metastore und PostgreSQL ist es möglich, jede Komponente individuell zu aktualisieren, zu warten oder auszutauschen, ohne das gesamte System zu beeinträchtigen. Dadurch wird nicht nur die Fehlerbehebung vereinfacht, sondern auch zukünftige Anpassungen, etwa durch geänderte Anforderungen, Aktualisierungen einzelner Komponenten, erheblich erleichtert.

7.2 Skalierbarkeit

Die Skalierbarkeit beschreibt die Fähigkeit einer Datenarchitektur, steigende Datenmengen und Nutzeranforderungen zu bewältigen, ohne dass die Performance wesentlich darunter leidet (Bondi, 2000, S. 1). Die bestehende FIDUS-Architektur weist hierbei deutliche Schwächen auf, die insbesondere durch die enge Verknüpfung der HANA-Datenbank mit SAP BW und die zahlreichen komplexen Transformationen entstehen. Diese monolithische Struktur sorgt für Engpässe, da eine horizontale Skalierung nur begrenzt möglich ist und erhöhte Anforderungen häufig teure und aufwendige Hardwareerweiterungen notwendig machen.

Der entwickelte Prototyp adressiert dieses Problem gezielt, indem er auf Technologien wie Apache Spark und Delta Lake setzt. Diese Komponenten wurden speziell für die verteilte Datenverarbeitung entwickelt und ermöglichen eine flexible horizontale Skalierung durch das einfache Hinzufügen weiterer Worker-Container (Chao et al., 2018). Dadurch können zusätzliche Kapazitäten geschaffen werden, ohne tiefgreifende Änderungen an der grundlegenden Architektur vornehmen zu müssen. Insbesondere die Nutzung von MinIO als S3-kompatiblem Object Storage unterstützt die Skalierbarkeit zusätzlich, da dieser Speicher durch seine Cloud-native Struktur von Natur aus einfach erweitert werden kann.

7.3 Erweiterbarkeit

Die Erweiterbarkeit beschreibt die Möglichkeit, neue Datenquellen, Verarbeitungsschritte oder analytische Anforderungen mit geringem Aufwand in die bestehende Architektur zu integrieren. In der bestehenden FIDUS-Architektur erfordert die Integration neuer Datenquellen oder Analyseanforderungen aufwendige Anpassungen auf mehreren Ebenen, insbesondere in den komplex verschachtelten Transformationsprozessen und den Join und Union-Operationen innerhalb von SAP BW. Der resultierende Aufwand ist sowohl zeitlich als auch finanziell erheblich, was die Anpassungsfähigkeit der Architektur reduziert.

Im Data-Lakehouse-Prototyp wurden diese Herausforderungen bewusst adressiert, indem offene Standards und modulare Komponenten verwendet wurden. Insbesondere das offene Speicherformat Delta Lake und die Verwendung standardisierter Schnittstellen wie Apache Parquet und S3 ermöglichen eine schnelle und unkomplizierte Integration neuer Datenquellen. Die Nutzung von Apache Spark erlaubt es, flexible und schnelle Transformationen durchzuführen, während der Hive Metastore für ein zentrales und konsistentes Metadatenmanagement sorgt. Dadurch entsteht eine Architektur, die es ermöglicht, neue Anforderungen schnell und mit minimalem Integrationsaufwand umzusetzen.

7.4 Offene Standards und Interoperabilität

Offene Standards fördern die Interoperabilität, d.h. die Fähigkeit eines Systems, reibungslos mit verschiedenen Technologien zusammenzuarbeiten. Die bestehende FIDUS-Architektur nutzt hauptsächlich geschlossene Technologien wie SAP HANA und SAP BW, was die Integration externer Anwendungen erschwert und häufig kostenintensive geschlossene Schnittstellen erfordert.

Die neue Data-Lakehouse-Architektur hingegen setzt bewusst auf Open-Source-Komponenten. Insbesondere die Nutzung des offenen Apache-Parquet-Formats, der S3-kompatiblen Schnittstelle von MinIO und der SQL-Query-Fähigkeit von Trino steigert die Kompatibilität zu externen Systemen deutlich. Zudem trägt die Nutzung offener Standards dazu bei, Vendor-Lock-In, also eine langfristige Abhängigkeit von einem bestimmten Anbieter aufgrund geschlossener Technologien, zu vermeiden. Dadurch wird die Flexibilität und Anpassungsfähigkeit der Architektur langfristig gewährleistet.

8 Diskussion

In dieser Arbeit wurde ein Prototyp zur Modernisierung der bestehenden Datenarchitektur eines führenden deutschen Augenheilkundenetzwerks mithilfe einer Data Lakehouse-Architektur entwickelt und implementiert. Die theoretischen Grundlagen sowie die darauf aufbauende Implementierung zeigen, dass der Einsatz moderner Open-Source-Technologien wie Apache Spark, Delta Lake, MinIO, Trino und Apache Superset viele der bestehenden Herausforderungen traditioneller Datenpipelines adressieren kann. Im Rahmen dieser Diskussion werden die Ergebnisse kritisch reflektiert und sowohl Stärken als auch potenzielle Schwachpunkte des Ansatzes beleuchtet.

Zusammenfassung der wichtigsten Erkenntnisse. Ein zentraler Vorteil der hier umgesetzten Lakehouse-Architektur liegt in der Nutzung offener Standards und der Vermeidung eines Vendor-Lock-In, der typischerweise mit geschlossenen Lösungen verbunden ist. Dies erlaubt eine flexible Integration weiterer Komponenten und reduziert Abhängigkeiten von einzelnen Anbietern. Gleichzeitig wurde durch die Nutzung von containerisierten Lösungen eine modulare Architektur geschaffen, die hinsichtlich Wartbarkeit, Erweiterbarkeit und Skalierbarkeit Vorteile gegenüber der bestehenden Struktur aufweist. Zudem konnten erste Tests zeigen, dass durch den Einsatz von Delta Lake die Datenkonsistenz verbessert und Transaktionssicherheit gewährleistet werden kann.

Vergleich mit der bestehenden Architektur. Während die neue Architektur in vielen Bereichen Vorteile bietet, gibt es auch einige Einschränkungen, die bei der Bewertung des Prototyps berücksichtigt werden müssen. So konnte aufgrund des Umfangs der Arbeit und der Komplexität der bestehenden Architektur keine vollständige funktionale Vergleichbarkeit mit der bestehenden FIDUS-Pipeline erreicht werden. Insbesondere komplexe Transformationen, die spezifisches Wissen über interne Geschäftslogiken voraussetzen, konnten im Rahmen des MVP nicht vollständig umgesetzt werden. Diese Tatsache schränkt die Aussagekraft hinsichtlich der praktischen Umsetzbarkeit in einem realen Unternehmenskontext ein.

Ein weiterer diskussionswürdiger Aspekt betrifft die Skalierbarkeit des Prototyps. Während theoretisch eine horizontale Skalierung durch Hinzufügen weiterer Worker relativ unkompliziert erfolgen kann, bleibt der praktische Nachweis dieser Skalierbarkeit innerhalb dieser Arbeit aufgrund der begrenzten Testumgebung auf ein einziges MacBook Pro limitiert. Es wäre daher ratsam, in zukünftigen Untersuchungen detaillierte Performance- und Skalierbarkeitstests unter realistischeren Bedingungen, beispielsweise in einer Cloud-basierten Umgebung, durchzuführen.

Bewertung der Forschungsfragen. Die zentrale Forschungsfrage, ob ein Data Lakehouse zur Optimierung bestehender Datenverarbeitungsprozesse implementiert werden kann, wurde in dieser Arbeit untersucht. Die Ergebnisse zeigen, dass eine Data Lakehouse-Architektur grundsätzlich eine flexible, wartbare und skalierbare Alternative zu herkömmlichen Architekturen darstellt. Dennoch wurden auch Herausforderungen identifiziert, insbesondere in Bezug auf die Komplexität der Integration und den initialen Implementierungsaufwand.

Zudem wurde die zweite Forschungsfrage, welche Vorteile und Herausforderungen sich aus der Implementierung eines Data Lakehouse ergeben, kritisch analysiert. Vorteile bestehen vor allem in der modularen Architektur, der besseren Datenverfügbarkeit und der Unabhängigkeit von geschlossenen Lösungen. Herausforderungen betreffen hingegen die Notwendigkeit spezifischer technischer Kenntnisse sowie die Evaluierung von Skalierung und Performance in produktiven Umgebungen.

Grenzen und Limitationen der Arbeit. Diese Arbeit weist mehrere Limitationen auf, die sich aus zeitlichen, ressourcenbedingten und zugangstechnischen Beschränkungen ergeben. Methodisch konnte keine umfangreiche quantitative Performance-Analyse durchgeführt werden, da der Prototyp nur in einer begrenzten Testumgebung validiert wurde. Die zur Verfügung stehenden Ressourcen erlaubten keine großflächigen Lasttests oder vergleichenden Benchmarks mit produktiven Systemen. Zudem sind die Ergebnisse auf ein spezifisches Szenario innerhalb der FIDUS-Pipeline zugeschnitten, was eine direkte Übertragbarkeit auf andere Systeme erschwert. Technische Einschränkungen bestanden darin, dass keine vollständige Orchestrierung der Datenprozesse umgesetzt wurde und einige fortgeschrittene Features der eingesetzten Technologien nicht getestet werden konnten. Darüber hinaus erschwerte der eingeschränkte Zugang zu bestehenden Transformationen der FIDUS-Architektur eine vollständige Migration aller bisherigen Prozesse.

Praktische Implikationen. Für Unternehmen, die eine Modernisierung ihrer Datenarchitektur erwägen, bietet ein Data Lakehouse eine vielversprechende Lösung. Die Nutzung von Open-Source-Technologien ermöglicht langfristig Kosteneinsparungen und mehr Flexibilität. Allerdings sollte bedacht werden, dass der Wechsel zu einer Lakehouse-Architektur initial mit einem hohen Implementierungsaufwand verbunden ist und eine gewisse technische Expertise voraussetzt. Unternehmen sollten daher sicherstellen, dass entsprechende Schulungen oder externe Beratungen in den Migrationsprozess eingeplant werden.

Darüber hinaus sollte noch berücksichtigt werden, dass bei der Größe und Komplexität des betrachteten Unternehmens zusätzlich der Einsatz eines Orchestration-Tools in Erwägung gezogen werden muss. Ein Orchestration-Tool, wie beispielsweise Apache Airflow

oder Apache NiFi, könnte erforderlich sein, um komplexe Abläufe, die aus einer Vielzahl von Datenströmen und Verarbeitungsschritten bestehen, effizient zu verwalten und automatisieren.

Theoretische Implikationen. Die Arbeit trägt zur wissenschaftlichen Diskussion über Data Lakehouses bei, indem sie eine praxisnahe Umsetzung und Evaluierung eines Prototyps liefert. Sie bestätigt bestehende Erkenntnisse über die Vorteile von Data Lakehouses, zeigt aber auch deren Herausforderungen in realen Anwendungsfällen auf. Die Ergebnisse unterstreichen die Notwendigkeit weiterführender Untersuchungen zur Integration solcher Architekturen in bestehende Unternehmensstrukturen.

Zukünftige Forschung und Weiterentwicklungen. Auf Basis dieser Arbeit ergeben sich mehrere vielversprechende Forschungsansätze für die Zukunft. Ein zentraler Aspekt ist die Durchführung umfassender Performance-Tests in einer realen Umgebung, um die tatsächliche Skalierbarkeit und Effizienz der entwickelten Architektur zu evaluieren. Darüber hinaus könnte die Architektur durch die Integration eines Orchestration-Tools erweitert werden, um die Automatisierung der Datenpipelines zu optimieren und die betriebliche Effizienz zu steigern. Ein weiterer relevanter Forschungsbereich wäre eine vergleichende Analyse verschiedener Data Lakehouse-Implementierungen, sowohl Open-Source- als auch kommerzieller Lösungen, um deren jeweilige Stärken und Schwächen herauszuarbeiten. Schließlich wäre es von Interesse, die Langzeit-Wartbarkeit eines Data Lakehouses in einem produktiven Umfeld zu untersuchen, um die Nachhaltigkeit und Anpassungsfähigkeit dieser Architektur unter realen Betriebsbedingungen zu bewerten.

9 Fazit

In dieser Arbeit wurde eine prototypische Data Lakehouse-Architektur entwickelt und implementiert, um die bestehende Dateninfrastruktur eines führenden deutschen Augenheilkundenetzwerks zu modernisieren. Ziel war es, die Vorteile von Data Lakes und Data Warehouses zu kombinieren und damit eine flexible, skalierbare und wartbare Lösung für datengetriebene Prozesse bereitzustellen. Die Evaluation der Architektur hat gezeigt, dass der Einsatz von Open-Source-Technologien wie Apache Spark, Delta Lake, MinIO und Trino wesentliche Vorteile hinsichtlich Datenintegration, Konsistenz und Skalierbarkeit bietet.

Die Ergebnisse dieser Arbeit unterstreichen die Relevanz von Data Lakehouses als moderne Datenarchitektur, insbesondere für Unternehmen, die mit großen und heterogenen

Datenmengen arbeiten. Die entwickelte Lösung zeigt, dass eine hybride Architektur auf Basis offener Standards eine vielversprechende Alternative zu geschlossenen Systemen darstellt. Unternehmen profitieren dabei von der erhöhten Flexibilität und Unabhängigkeit von einzelnen Anbietern.

Dennoch gibt es Limitationen, die bei der praktischen Umsetzung berücksichtigt werden müssen. Die in dieser Arbeit durchgeführten Tests erfolgten unter begrenzten Bedingungen, weshalb eine weiterführende Evaluation in großflächigen, produktiven Umgebungen erforderlich ist. Zudem stellt die Migration von bestehenden Datenarchitekturen auf ein Data Lakehouse eine Herausforderung dar, da sie spezifisches technisches Know-how und eine sorgfältige Planung erfordert.

Künftige Forschungsarbeiten sollten sich mit der Optimierung der Performance in realen Einsatzszenarien, der Integration von automatisierten Orchestrierungstools sowie der langfristigen Wartbarkeit und Weiterentwicklung solcher Architekturen beschäftigen. Unternehmen, die eine moderne und skalierbare Datenplattform aufbauen möchten, sollten Data Lakehouses als ernsthafte Alternative in Betracht ziehen.

Literaturverzeichnis

Apache Hive (2025): Apache Hive. `https://hive.apache.org/`. Zugriff am 20. Januar 2025.

Apache Parquet (2025): Apache Parquet Documentation. Letzter Zugriff: 06. Februar 2025.

Apache Software Foundation (2025a): Apache Spark Documentation - RDD Programming Guide. `https://spark.apache.org/docs/3.5.4/rdd-programming-guide.html`. Zugriff am 3. Februar 2025.

Apache Software Foundation (2025b): Apache Superset. Zugriff am 30. Januar 2025.

Apple Support (2025): MacBook Pro – Technische Daten. Zugriff am 3. März 2024.

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Świtakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., Xin, R., und Zaharia, M. (2020): Delta lake. *Proc. of the VLDB Endowment*, **Vol. 13** (Nr. 12), S. 3411–3424.

Armbrust et al (2021): Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics.

Berger-Grabner, D. (2022): *Wissenschaftliches Arbeiten in den Wirtschafts- und Sozialwissenschaften: hilfreiche Tipps und praktische Beispiele*. Springer eBook Collection. Springer Gabler, 4., überarbeitete und erweiterte auflage Auflage.

Bogner, A. (2014): *Interviews mit Experten: Eine praxisorientierte Einführung*. Qualitative Sozialforschung. Springer VS.

Bondi, A. B. (2000): Characteristics of scalability and their impact on performance. In: *Proceedings of the 2nd international workshop on Software and performance*, S. 195–203. ACM.

Camacho-Rodríguez, J., Chauhan, A., Gates, A., Koifman, E., O'Malley, O., Garg, V., Haindrich, Z., Shelukhin, S., Jayachandran, P., Seth, S., Jaiswal, D., Bouguerra, S., Bangarwa, N., Hariappan, S., Agarwal, A., Dere, J., Dai, D., Nair, T., Dembla, N., Vijayaraghavan, G., und Hagleitner, G. (2019): Apache Hive: From MapReduce to Enterprise-grade Big Data Warehousing. In: *Proceedings of the 2019 International Conference on Management of Data*, S. 1773–1786. ACM.

Chao, Z., Shi, S., Gao, H., Luo, J., und Wang, H. (2018): A gray-box performance model for Apache Spark. **89**, S. 58–67.

Databricks (2025a): Medallion Architecture – Bronze, Silver und Gold Layers. Zugriff am 3. März 2025.

Databricks (2025b): Was ist Apache Hive? - Glossar. Zugriff am 21. Januar 2025.

Docker Inc. (2025): Docker - Develop, Share, and Run Applications Anywhere. Zugriff am 3. Februar 2025.

FIDUS GmbH (2025): FIDUS Arztpraxissoftware GmbH - Offizielle Webseite. Letzter Zugriff: 06. Februar 2025.

Fraser, G. (2022): Cloud Data Warehouse Benchmark. Zugriff am 14. Februar 2025.

Gadban, F., und Kunkel, J. (2021): Analyzing the Performance of the S3 Object Storage API for HPC Workloads. **11** (18), S. 8540.

Guo, R., Zhao, Y., Zou, Q., Fang, X., und Peng, S. (2018): Bioinformatics applications on Apache Spark.

Harby, A. A., und Zulkernine, F. (2022): From Data Warehouse to Lakehouse: A Comparative Review. In: *2022 IEEE International Conference on Big Data (Big Data)*, S. 389–395. IEEE.

Heitlager, I., Kuipers, T., und Visser, J. (2007): A Practical Model for Measuring Maintainability. In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, S. 30–39. IEEE.

Khine, P. P., und Wang, Z. S. (2018): Data lake: a new ideology in big data era. *ITM Web of Conferences*, **17**, S. 03025.

Kramer, H. (2025): Persönliche Kommunikation mit Henryk Kramer - Data Engineer. Interview geführt am 15. Januar 2025. Nicht veröffentlichte persönliche Mitteilung.

Lobster GmbH (2025): Lobster Produkte - Übersicht der Softwarelösungen. Letzter Zugriff: 13. Januar 2025.

Marchesi, M., Marchesi, L., und Tonelli, R. (2018): An Agile Software Engineering Method to Design Blockchain Applications. In: *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia*, S. 1–8. ACM.

Masak, D. (2010): *Der Architekturreview: Vorgehensweise, Konzepte und Praktiken.* Xpert.press. Springer Berlin Heidelberg.

Medjahed, B., Ouzzani, M., und Elmagarmid, A. (2009): Generalization of ACID Properties. Cyber center publications, Purdue University. Zugriff am 21. Januar 2025.

Mindtwo GmbH (2025): Docker: Effizienz und Skalierbarkeit in der Softwareentwicklung. Zugriff am 3. Februar 2025.

MinIO Inc. (2025): Minio Inc. `https://min.io/`. Zugriff am 27. Januar 2025.

Müller, A. (2025): Persönliche Kommunikation mit Andreas Müller - Head of Data. Interview geführt am 15. Januar 2025. Nicht veröffentlichte persönliche Mitteilung.

Peffers, K., Tuunanen, T., Rothenberger, M. A., und Chatterjee, S. (2007): A Design Science Research Methodology for Information Systems Research. **24** (3), S. 45–77.

PostgreSQL Global Development Group (2024): PostgreSQL - The World's Most Advanced Open Source Relational Database. Zugriff am 25. Januar 2025.

Preuss, P. (2017): In-Memory-Datenbank SAP HANA.

SAP SE (2025a): SAP BusinessObjects Analysis for Microsoft Office - Dokumentation. Letzter Zugriff: 13. Januar 2025.

SAP SE (2025b): SAP BW/4HANA - Data Warehousing. Letzter Zugriff: 13. Januar 2025.

SAP SE (2025c): What is SAP HANA? - Official SAP Website. Letzter Zugriff: 13. Januar 2025.

Satyamurty, C. V. S., und Murthy, J. V. R. (2018): Extraction of Metadata from ETL Environment. *International Journal of Emerging Technology and Advanced Engineering*. Zugriff am 25. Januar 2025.

Sawadogo, P., und Darmont, J. (2021): On data lake architectures and metadata management. *Journal of Intelligent Information Systems*, **56** (1), S. 97–120.

Schneider, J., Gröger, C., Lutsch, A., Schwarz, H., und Mitschang, B. (2023): Assessing the Lakehouse: Analysis, Requirements and Definition:. In: *Proceedings of the 25th International Conference on Enterprise Information Systems*, S. 44–56. SCITEPRESS - Science and Technology Publications.

Serra, J. (2024): *Deciphering data architectures: choosing between a modern data warehouse, data fabric, data lakehouse, and data mesh*. O'Reilly Media, Incorporated, first edition Auflage.

Sethi, R., Traverso, M., Sundstrom, D., Phillips, D., Xie, W., Sun, Y., Yegitbasi, N., Jin, H., Hwang, E., Shingte, N., und Berner, C. (2019): Presto: SQL on Everything. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, S. 1802–1813. IEEE.

Shi, X., Nikolic, G., Fischaber, S., Black, M., Rankin, D., Epelde, G., Beristain, A., Alvarez, R., Arrue, M., Pita Costa, J., Grobelnik, M., Stopar, L., Pajula, J., Umer, A., Poliwoda, P., Wallace, J., Carlin, P., Pääkkönen, J., und De Moor, B. (2022): System Architecture of a European Platform for Health Policy Decision Making: MIDAS. *Frontiers in Public Health*, **10**.

Stevenson, R., Burnell, D., und Fisher, G. (2024): The Minimum Viable Product (MVP): Theory and Practice. **50** (8), S. 3202–3231.

Trino (2025): Trino - Distributed SQL Query Engine. `https://trino.io/`. Zugriff am 30. Januar 2025.

Vaisman, A., und Zimányi, E. (2022): *Data Warehouse Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Vom Brocke, J. (2020): Design Science Research. Cases.

Zhang, C., Shen, Q., und Tang, B. (2023): DHive: Query Execution Performance Analysis via Dataflow in Apache Hive. In: *Proceedings of the VLDB Endowment*, Band 16, S. 3998–4001.

A Experteninterview

A.1 Leitfragen

Kategorie	Fragen
Unternehmenshintergrund und Rolle der Daten	• Können Sie kurz Ihr Unternehmen und dessen Hauptaktivitäten beschreiben? • Wie wichtig sind Daten in Ihrem Geschäftsmodell und welchen Stellenwert haben sie in Ihren Prozessen? • Welche Abteilung(en) sind für die Datenverwaltung und -architektur zuständig? • Welche Rolle spielen Sie persönlich in diesem Kontext?
Aktuelle Datenarchitektur	• Wie ist Ihre derzeitige Datenarchitektur aufgebaut? • Welche Systeme (z.B. Datenbanken, Data Lakes, Data Warehouses) kommen zum Einsatz? • Wie erfolgt die Erfassung, Speicherung und Verarbeitung der Daten? • Welche Technologien und Tools werden verwendet?
Datapipeline im Detail	• Welche Datapipeline nutzen Sie aktuell? • Können Sie den Aufbau und die Architektur dieser Pipeline näher erläutern? • Welche Pipeline darf ich im Rahmen der Bachelorarbeit als Referenz oder Beispiel nutzen? • Gibt es Besonderheiten oder Spezifikationen, die für die Nutzung im Prototyp zu beachten sind?
Anforderungen an die zukünftige Architektur (Prototyp)	• Welche grundlegenden Anforderungen muss die neue Architektur erfüllen? • Auf welche Funktionalitäten legen Sie besonderen Wert? • Gibt es zusätzliche Features oder Verbesserungen, die Sie erwarten? • Welche Aspekte der aktuellen Pipeline sollen übernommen oder optimiert werden?
Identifikation von Problemen und Herausforderungen	• Welche konkreten Probleme oder Schwachstellen sehen Sie in Ihrer bestehenden Datapipeline? • Inwiefern beeinträchtigen diese Herausforderungen die Effizienz, Flexibilität und Zuverlässigkeit der Datenprozesse? • Gibt es manuelle Prozesse oder Schnittstellen, die als kritisch betrachtet werden? • Wie wirken sich diese manuellen Eingriffe auf den Betrieb aus?
Ausblick und Weiterentwicklung	• Wie sehen Ihre zukünftigen Pläne für die Weiterentwicklung der Datenarchitektur aus? • Welche Rolle spielt der geplante Prototyp in dieser Strategie? • Haben Sie zusätzliche Anmerkungen oder Empfehlungen? • Gibt es andere Ansprechpartner oder Quellen für weiterführende Informationen?

Interviewleitfaden zur Analyse der Datenarchitektur

A.2 Experteninterview mit Herr Kramer

Die Antworten des Interviews wurden gekürzt, jedoch sinngemäß wiedergegeben.

Frage: Können Sie kurz Ihr Unternehmen und dessen Hauptaktivitäten beschreiben?

Kramer: Unternehmensgruppe im Bereich der Augenheilkunde. Fokus liegt auf dem Kauf und der Optimierung von Arztpraxen und MVZs (Mergers & Acquisition).

Frage: Wie wichtig sind Daten in Ihrem Geschäftsmodell und welchen Stellenwert haben sie in Ihren Prozessen?

Kramer: Daten sind sehr wichtig: Vor dem Kauf, um potenzielle M&A-Targets ausfindig zu machen und auf Rentabilität zu prüfen. Nach dem Kauf, um zu überwachen und zu steuern, ob die Arztpraxen die Ziele erfüllen, und um Optimierungspotenzial zu entdecken. Darüber hinaus werden die Daten für das zentrale Abrechnungs- und Mahnwesen in der Zentrale benötigt.

Frage: Welche Abteilung(en) sind für die Datenverwaltung und -architektur zuständig?

Kramer: Bis Anfang 2024 externe Dienstleister. Seit 2024 neue Abteilung „Datenintegration“, die bestehende Architekturen der OSG und veonet weiterentwickelt und verbessert.

Frage: Welche Rolle spielen Sie persönlich in diesem Kontext?

Kramer: Ich bin Data Engineer und entwickle die Datenstrecken im Enterprise Service Bus Lobster_data weiter. Ich unterstütze Management, Controlling und FiBu bei Datenanforderungen und steuere externe Dienstleister bei Implementierungen in HANA- und BW/4HANA-Systemen.

Frage: Wie ist Ihre derzeitige Datenarchitektur aufgebaut?

Kramer: Etwa 10 verschiedene Praxismanagementsysteme (PMS). Ein S4/HANA-System für buchhaltungsrelevante Daten. Lobster_data als ESB und ETL-Tool für OLTP und OLAP. Zudem nutzen wir ein BW/4HANA als Data Warehouse für weitere Transformationen und Speicherung der Querys.

Frage: Wie erfolgt die Erfassung, Speicherung und Verarbeitung der Daten?

Kramer: Erfassung durch PMS in Praxen, Speicherung in HANA-Datenbank mit direkter Anbindung an das BW. Verarbeitung über SQL-Skripte, Lobster_data und BW/4HANA.

Frage: Welche Datapipeline darf ich als Referenz nutzen?

Kramer: Eine Pipeline zur Analyse von Arzt-Patienten-Kontakten (APK) aus dem PMS „FIDUS“. APK bezeichnet Treffen zwischen Arzt und Patient (keine Rezeptabholung oder Sehschule). Derzeit gibt es 8 FIDUS-Systeme für ca. 40 Standorte.

Frage: Gibt es Besonderheiten für die Nutzung im Prototyp zu beachten?

Kramer: Die Tabelle „ekktexte“ im EAV-Modell.

Frage: Welche grundlegenden Anforderungen muss der Prototyp erfüllen?

Kramer:

- Einfache Anpassungsfähigkeit
- Nachverfolgbarkeit der Transformationsschritte (Data Lineage)
- Skalierbarkeit bei neuen FIDUS-Systemen
- Performance der Querys

Frage: Gibt es zusätzliche Features oder Verbesserungen, die Sie erwarten?

Kramer: Streaming, z.B. zur direkten Darstellung der Auswirkungen von Änderungen.

Frage: Welche konkreten Probleme sehen Sie in Ihrer bestehenden Datapipeline?

Kramer: Zu viele Tools, schwierige Nachvollziehbarkeit (Low-Code/No-Code), Lobster_data ist für OLAP wenig geeignet.

Frage: Gibt es kritische manuelle Prozesse?

Kramer: Manuelle Mappings (z.B. Arztkürzel zu Kostenstelle) mit hoher Fehleranfälligkeit.

Frage: Wie sehen Ihre zukünftigen Pläne für die Weiterentwicklung der Datenarchitektur aus?

Kramer: Aktuell erwägen wir, die alte Architektur durch eine völlig neue Lösung zu ersetzen (grüne Wiese).

Frage: Welche Rolle spielt der geplante Prototyp in dieser Strategie?

Kramer: Der Prototyp ist eine mögliche Implementierung der neuen Datenarchitektur.

Frage: Haben Sie weitere Anmerkungen oder Empfehlungen?

Kramer: Nein.

Frage: Gibt es weitere Ansprechpartner oder Quellen für zusätzliche Informationen?

Kramer: Nein.

A.3 Experteninterview mit Herr Müller

Die Antworten des Interviews wurden gekürzt, jedoch sinngemäß wiedergegeben.

Frage: Können Sie kurz Ihr Unternehmen und dessen Hauptaktivitäten beschreiben?

Müller: Die Hauptaktivität der OSG liegt im Bereich Merge & Akquisition von ärztlichen Praxen und medizinischen Versorgungszentren im Bereich der Augenheilkunde (Opthomologie).

Frage: Wie wichtig sind Daten in Ihrem Geschäftsmodell und welchen Stellenwert haben sie in Ihren Prozessen?

Müller: Daten sind gerade im Bereich M&A von existenzieller Bedeutung, da sie für die Unternehmenssteuerung unabdingbar sind. Sie stellen letztlich sicher, dass kommende Investitionen getätigt, die Zukunftssicherheit der Organisation gegeben und in gewisser Weise die medizinische Versorgung im Bereich der Opthomologie über die kommenden Jahre gewährleistet werden kann.

Frage: Welche Abteilung(en) sind für die Datenverwaltung und -architektur zuständig?

Müller: Für die in den Praxen befindlichen Systeme sind die jeweiligen Softwarehersteller verantwortlich. Daten, die wir von dort und aus anderen Systemen erhalten, werden u.a. in einem zentralen Business Warehouse von SAP verarbeitet. Dieses unterliegt der Hoheit der Abteilung Business Applikationen, Datenintegration und Prozesse (kurz BDP). Es gibt auch Cloudsoftware im Sinne von Software-as-a-Service. Hier hat man als Kunde nur eingeschränkt Einfluss, da die Systeme oftmals „schlüsselfertig“ gestellt werden.

Frage: Welche Rolle spielen Sie persönlich in diesem Kontext?

Müller: Als Leiter des Bereiches BDP bin ich dafür verantwortlich, dass wir unserer Schnittstellenfunktion zu den Fachbereichen, der internen IT und externen Dienstleistern gerecht werden können. Daneben lege ich die Umfänge und Prioritäten der internen Projekte fest, die sich aus den Anforderungen unseres Managements ergeben.

Frage: Wie ist Ihre derzeitige Datenarchitektur aufgebaut?

Müller: Siehe Henryks Antwort.

Frage: Welche Systeme kommen zum Einsatz?

Müller: BW/4HANA in der OSG. Darüber hinaus siehe Henryks Antwort.

Frage: Wie erfolgt die Erfassung, Speicherung und Verarbeitung der Daten?

Müller: Die Erfassung erfolgt hauptsächlich in den Patientenmanagementsystemen der Arztpraxen. Dazu kommen Formulare, die in Lobster realisiert sind, Excel-Listen und papiergeführte Unterlagen (non-digital). Innerhalb der OSG nutzt die HR SAGE, der Einkauf comed, Kreditoren xSuite (E-Mail) sowie Excel.

Frage: Welche Technologien und Tools werden verwendet?

Müller: Siehe oben.

Frage: Welche Datapipeline nutzen Sie aktuell?

Müller: Unser Enterprise Service Bus wurde technologisch auf Basis von Lobster realisiert.

Frage: Können Sie den Aufbau und die Architektur dieser Pipeline näher erläutern?

Müller: Leistungsstarke Datenintegration & Automatisierung durch Lobster.

Frage: Welche Pipeline darf ich im Rahmen der Bachelorarbeit als Referenz nutzen?

Müller: Siehe Henryk.

Frage: Gibt es Besonderheiten oder Spezifikationen zu beachten?

Müller: Siehe Henryk.

Frage: Welche grundlegenden Anforderungen muss die neue Architektur erfüllen?

Müller: Dicht an Low-Code mit der notwendigen Flexibilität, direkt eingreifen zu können.

Frage: Auf welche Funktionalitäten legen Sie besonderen Wert?

Müller: Idealerweise eine visuelle Anzeige der Datensätze, die die Verarbeitungsknotenpunkte durchlaufen haben. Nachverfolgbarkeit ist das A und O.

Frage: Gibt es zusätzliche Features oder Verbesserungen, die Sie erwarten?

Müller: Wie oben beschrieben, visuelle Anzeige der Datensätze, die einzelne Passagen passiert haben.

Frage: Welche Aspekte der aktuellen Pipeline sollen zwingend übernommen oder optimiert werden?

Müller: Die geringe Einstiegshürde, sodass definierte Prozesse von Benutzern selbst gestartet werden können.

Frage: Welche konkreten Probleme oder Schwachstellen sehen Sie in Ihrer bestehenden Datapipeline?

Müller: Zu viele Profile durch Systemvielfalt und zu viele Modellierungsschritte in Profilen.

Frage: Wie beeinträchtigen diese Herausforderungen Ihre Datenprozesse?

Müller: Wartbarkeit wird immer schwieriger, besonders bei der Fehlersuche.

Frage: Gibt es manuelle Prozesse oder Schnittstellen, die kritisch sind?

Müller: Alle Schnittstellen zu Fachbereichen, die Informationen für Transformationen liefern (Mappings).

Frage: Wie wirken sich diese manuellen Eingriffe auf den Betrieb aus?

Müller: Falsche Mappings können ganze Berichte unbrauchbar machen, da wir nicht jeden Userfehler abfangen können.

Frage: Wie sehen Ihre zukünftigen Pläne für die Weiterentwicklung der Datenarchitektur aus?

Müller: Kompletter Neustart mit einem leeren Data Warehouse.

Frage: Welche Rolle spielt der geplante Prototyp?

Müller: Der Prototyp dient der Technologieerprobung, insbesondere ob Open Source geeignet ist.

Frage: Haben Sie zusätzliche Anmerkungen oder Empfehlungen?

Müller: Die neue Architektur muss personell skalieren können, um viele Anforderungen parallel bearbeiten zu können. Sie darf kein Nischenprodukt sein.

Frage: Gibt es andere Ansprechpartner oder Quellen für weiterführende Informationen?

Müller: Nein.

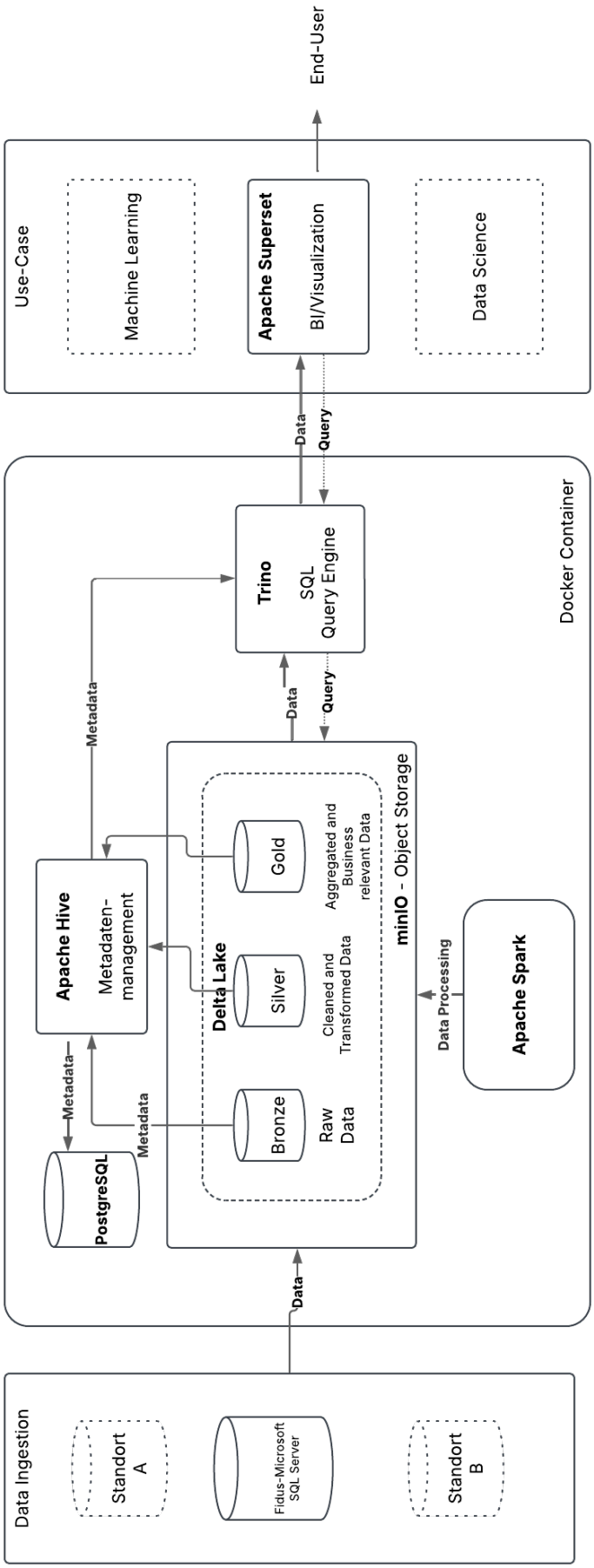
B FIDUS Data Pipeline

B.1 KPI 10 SQL Query

```
select DATEPART( MONTH, ekk.datum) [Monat], DATEPART(iso_week, ekk.datum) [KW], DATEPART( YEAR, ekk.datum) [Jahr], ekk.datum [Datum], ekk.Versicherungsstatus,
ekk.[BSNR] [Standort], count(*) [Anzahl], FORMAT (getdate(), 'yyyy-MM-dd') [Lieferdatum], DATEPART(iso_week, getdate()) [LieferKW] from (

  select distinct e.patnr, e.datum, iif(p.VERST = 8, 'Privat', 'Kasse') [Versicherungsstatus],
  iif( schein.arzt is null and rechnung.arzt is null, bsnr.[Standort], iif( schein.arzt is null, RIGHT( rechnung.arzt, 3), RIGHT(schein.arzt, 3) ) ) [BSNR]
  from ekktexte e
  left join patdat p on p.PATNR = e.PATNR
  outer apply ( select top 1 bh.arzt from behdatei bh where bh.KSBEHDAT = e.datum and bh.KSPATNR = e.patnr and bh.arzt <> ''  ) schein
  outer apply ( select top 1 rp.arzt from rechkopf rk, rechpos rp where rk.RKOPFID = rp.RKOPFID and rk.PATNR = e.PATNR and rp.LEISTDATUM = e.DATUM and rp.arzt <> ''    ) rechnung
  OUTER APPLY (  select top 1 SCHLUESSEL as [Standort] from ckeytabs where OBJEKT = 'OSG_STANDORT' ) bsnr
  where (e.fduschl in ('IVOMMDR:', 'IVOMMDL:',  'OPDKARR:', 'OPDKARL:', 'OPDAARR:', 'OPDAARL:') or e.fduschl like 'MBefA%' )
  and  e.DATUM > '2021-04-01' and DATEPART(dw, e.datum) not in (1,7) and
    e.DATUM between DATEADD(month, -3, getdate()) and getdate()
) as ekk
group by ekk.DATUM, ekk.Versicherungsstatus, ekk.[BSNR] order by ekk.DATUM, ekk.Versicherungsstatus, ekk.[BSNR]
```

C Data Lakehouse für FIDUS Pipeline



D Implementierung - Konfigurationen und Skripte

D.1 Docker Compose Konfiguration - compose.yaml

```
services:

  spark-master:
    build: ./spark-master
    container_name: spark-master
    networks:
      - lakehouse
    ports:
      - "4040:4040"  # Spark UI
      - "7077:7077"  # Spark Master
      - "8888:8888"  # Jupyter Notebook
      - "8080:8080"  # Spark Master UI
    volumes:
      - ./spark-master/notebooks:/opt/spark/work-dir
      - ./spark-master/test_data:/opt/spark/work-dir/data
      - ./spark-master/hive-site.xml:/opt/spark/conf/hive-site.xml
      - ./spark-master/spark-defaults.conf:/opt/spark/conf/spark-
    defaults.conf
      - ./hive/jars:/opt/spark/hive/jars
    tty: true
    stdin_open: true
    environment:
      SPARK_MASTER_HOST: spark-master
      SPARK_MASTER_PORT: 7077
    command: opt/spark/bin/spark-class org.apache.spark.deploy.master.
    Master

  spark-worker:
    build: ./spark-worker
    networks:
      - lakehouse
    environment:
      SPARK_WORKER_CORES: 5
      SPARK_WORKER_MEMORY: 5g
    volumes:
      - ./spark-worker/notebooks:/opt/spark/work-dir
      - ./spark-worker/test_data:/opt/spark/work-dir/data
      - ./spark-worker/hive-site.xml:/opt/spark/conf/hive-site.xml
      - ./spark-worker/spark-defaults.conf:/opt/spark/conf/spark-
    defaults.conf
      - ./hive/jars:/opt/spark/hive/jars
```

```
    tty: true
    stdin_open: true
    depends_on:
      - spark-master

  trino:
    image: 'trinodb/trino:latest'
    hostname: trino
    container_name: trino
    depends_on:
       - metastore
       - minio-host
    ports:
      - '8081:8081'
    volumes:
      - ./trino:/etc/trino  # Katalog-Konfiguration
    networks:
      - lakehouse


  minio-host:
    image: minio/minio:latest
    container_name: minio-host
    ports:
      - "9000:9000"      # Minio API-Port (S3-kompatibel)
      - "9001:9001"      # Minio Console (Admin-UI)
    environment:
      MINIO_ROOT_USER: minio
      MINIO_ROOT_PASSWORD: minio123
    volumes:
      - ./minio-data:/data    # Lokales Volume f r die persistente
   Speicherung von Objektdaten
    command: server /data --console-address ":9001" # Startparameter
   f r Minio
    networks:
      - lakehouse

  metastore:
    image: 'apache/hive:4.0.1'
    hostname: metastore
    container_name: metastore
    depends_on:
      - postgres
    ports:
        - '9083:9083' # thrift port
    environment:
```

```
      - SERVICE_NAME=metastore
      - DB_DRIVER=postgres
      - DB_HOST=postgres
      - DB_PORT=5434
      - DB_NAME=hive_db
    volumes:
      - ./hive/data:/opt/hive/data/warehouse
      - type: bind
        source: ./hive/config/hive-site.xml
        target: /opt/hive/conf/hive-site.xml
      - type: bind
        source: ./hive/TempPackages/hadoop-aws-3.3.4.jar
        target: /opt/hive/lib/hadoop-aws-3.3.4.jar
      - type: bind
        source: ./hive/TempPackages/aws-java-sdk-bundle-1.12.262.jar
        target: /opt/hive/lib/aws-java-sdk-bundle-1.12.262.jar
      - type: bind
        source: ./hive/TempPackages/postgresql-42.7.5.jar
        target: /opt/hive/lib/postgresql-42.7.5.jar
    command:
      - /opt/hive/bin/hive --service metastore
    networks:
      - lakehouse

  postgres:
    image: 'postgres:latest'
    hostname: postgres
    container_name: postgres
    environment:
    - POSTGRES_USER=hive_db
    - POSTGRES_PASSWORD=password
    - POSTGRES_DB=hive_db
    - LANG=en_US.UTF-8
    - LC_ALL=en_US.UTF-8
    - PGCLIENTENCODING=UTF8
    volumes:
      - ./hive/metastore_db:/var/lib/postgresql/data
    ports:
      - '5434:5434'
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready -U hive_db" ]
      interval: 5s
      timeout: 5s
      retries: 2
    networks:
      - lakehouse
```

```
129
130 networks:
131   lakehouse:
132     driver: bridge
133     external: true
134     name: lakehouse
```

D.2 Spark Konfiguration

D.2.1 Dockerfile für Spark-Master und Spark Worker

Docker Compose führt dieses Dockerfile aus, bevor ein Image erstellt wird, damit das Spark-Image die richtige Umgebung erhält.

```
1     FROM spark:3.5.4-scala2.12-java17-python3-ubuntu
2
3     USER root
4
5     RUN set -ex; \
6         apt-get update && \
7         apt-get install -y python3 python3-pip && \
8         rm -rf /var/lib/apt/lists/*
9     ENV PATH=$PATH:$JAVA_HOME/bin
10    # Install PySpark
11    RUN pip3 install --upgrade pip
12    COPY  requirements.txt .
13    RUN pip3 install -r requirements.txt
14
15    ADD https://repo1.maven.org/maven2/org/apache/hadoop/hadoop-aws
    /3.3.4/hadoop-aws-3.3.4.jar $SPARK_HOME/jars/
16
17    ADD https://repo1.maven.org/maven2/com/amazonaws/aws-java-sdk-bundle
    /1.12.262/aws-java-sdk-bundle-1.12.262.jar $SPARK_HOME/jars/
18
19    ADD https://repo1.maven.org/maven2/org/xerial/sqlite-jdbc/3.49.0.0/
    sqlite-jdbc-3.49.0.0.jar $SPARK_HOME/jars/sqlite-jdbc.jar
20
21    RUN rm -f requirements.txt
22    ENV SPARK_HOME=/opt/spark
23    ENV PATH=$PATH:$SPARK_HOME/bin
24    ENV PATH=$PATH:$SPARK_HOME/sbin
25    RUN mkdir -p /opt/spark/spark-events
26    COPY spark-defaults.conf /opt/spark/conf/spark-defaults.conf
```

```
    COPY hive-site.xml /opt/spark/conf/hive-site.xml
    RUN pip install jupyterlab
    COPY entrypoint.sh /entrypoint.sh
    RUN chmod +x /entrypoint.sh
    # Setze das Skript als ENTRYPOINT, sodass es beim Containerstart
   automatisch ausgef hrt wird
    ENTRYPOINT ["/entrypoint.sh"]
```

D.2.2 Entrypoint.sh Spark-Master

Hier wird beim Starten des Spark-Masters Container die Spark Master und Jupyter Notebook Umgebung gestartet.

```
    #!/bin/bash
set -e

echo "Starte Spark Master..."
start-master.sh

echo " ffne  interaktive Shell"
exec jupyter notebook --port=8888 --NotebookApp.token='' --allow-root --
   ip=0.0.0.0
```

D.2.3 Entrypoint.sh Spark-Woker

Hier wird beim Starten des Spark-Worker Container, die Spark Worker Umgebung gestartet.

```
#!/bin/bash
set -e

echo "Starte Spark Worker..."
$SPARK_HOME/sbin/start-worker.sh spark://spark-master:7077
echo "Spark Worker gestartet"


echo " ffne  interaktive Shell..."
exec /bin/bash
```

D.2.4 spark-defaults.config

Hier werden die Standardkonfigurationen eingetragen, die beim Start einer Spark-Session übernommen werden. Sie enthalten relevante Einstellungen zur Nutzung von Delta Lake, Hive und MinIO.

```
spark.jars.packages=io.delta:delta-spark_2.12:3.3.0
spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension
spark.hadoop.fs.s3a.connection.ssl.enabled=false
spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.
   DeltaCatalog
spark.hadoop.fs.s3a.aws.credentials.provider=org.apache.hadoop.fs.s3a.
   SimpleAWSCredentialsProvider
spark.hadoop.fs.s3a.endpoint=http://minio-host:9000
spark.hadoop.fs.s3a.path.style.access=true
spark.hadoop.fs.s3a.access.key=minio
spark.hadoop.fs.s3a.secret.key=minio123
spark.sql.catalogImplementation=hive
spark.sql.warehouse.dir=/user/hive/warehouse
spark.master=spark://spark-master:7077
spark.eventLog.enabled=true
spark.eventLog.dir=/opt/spark/spark-events
spark.history.fs.logDirectory=/opt/spark/spark-events
```

D.2.5 hive-site.xml

Wichtige Konfigurationen damit die Kombination aus Spark, Hive und MinIO funktioniert.

```
<configuration>
  <property>
      <name>hive.metastore.uris</name>
        <value>thrift://host.docker.internal:9083</value>
      <description>Thrift URI for the remote metastore. Used by
    metastore client to connect to remote metastore.</description>
  </property>

  <property>
    <name>metastore.warehouse.dir</name>
    <value>s3a://hive/</value>
    <description>Standard-Verzeichnis f r Hive-Tabellen</description>
  </property>

    <property>
```

```
<name>fs.defaultFS</name>
<value>s3://hive/</value>
</property>

  <!-- S3A / MinIO Konfiguration -->
  <property>
    <name>fs.s3a.endpoint</name>
    <value>http://minio-host:9000</value>
    <description>Endpoint f r S3A.</description>
  </property>

  <property>
    <name>fs.s3a.access.key</name>
    <value>minio</value>
    <description>Access Key f r MinIO.</description>
  </property>

  <property>
    <name>fs.s3a.secret.key</name>
    <value>minio123</value>
    <description>Secret Key f r MinIO.</description>
  </property>

  <property>
    <name>fs.s3a.connection.ssl.enabled</name>
    <value>false</value>
    <description>SSL deaktivieren, falls MinIO ber HTTP angesprochen
     wird.</description>
  </property>

  <property>
    <name>fs.s3a.path.style.access</name>
    <value>true</value>
    <description>Aktiviert den "path-style" Zugriff, der h ufig f r S3
     -kompatible Dienste wie MinIO ben tigt wird.</description>
  </property>
</configuration>
```

D.3 Hive Konfigurationen

D.3.1 Dockerfile

```
FROM openjdk:8-jre

```

```
WORKDIR /opt

ENV HADOOP_VERSION=3.3.4
ENV METASTORE_VERSION=4.0.1

ENV HADOOP_HOME=/opt/hadoop-${HADOOP_VERSION}
ENV HIVE_HOME=/opt/apache-hive-metastore-${METASTORE_VERSION}-bin

RUN curl -L https://archive.apache.org/dist/hive/hive-standalone-metastore-${METASTORE_VERSION}/hive-standalone-metastore-${METASTORE_VERSION}-bin.tar.gz | tar zxf - && \
    curl -L https://archive.apache.org/dist/hadoop/common/hadoop-${HADOOP_VERSION}/hadoop-${HADOOP_VERSION}.tar.gz | tar zxf -

RUN curl -o mysql-connector-java-8.0.19.jar https://repo1.maven.org/maven2/mysql/mysql-connector-java/8.0.19/mysql-connector-java-8.0.19.jar && \
    cp mysql-connector-java-8.0.19.jar ${HIVE_HOME}/lib/ && \
    curl -o delta-hive-assembly_2.11-0.2.0.jar https://github.com/delta-io/connectors/releases/download/v0.2.0/delta-hive-assembly_2.11-0.2.0.jar && \
    cp delta-hive-assembly_2.11-0.2.0.jar ${HIVE_HOME}/lib/ && \
    cp delta-hive-assembly_2.11-0.2.0.jar ${HADOOP_HOME}/share/hadoop/tools/lib/

COPY config/hive-site.xml ${HIVE_HOME}/conf
COPY entrypoint.sh /entrypoint.sh

RUN groupadd -r hive --gid=1000 && \
    useradd -r -g hive --uid=1000 -d ${HIVE_HOME} hive && \
    chown hive:hive -R ${HIVE_HOME} && \
    chown hive:hive /entrypoint.sh && chmod +x /entrypoint.sh

USER hive
EXPOSE 9083

ENTRYPOINT ["sh", "-c", "/entrypoint.sh"]
```

D.3.2 entrypoint.sh

```
#!/bin/sh

export HADOOP_HOME=/opt/hadoop-3.3.4
export HADOOP_CLASSPATH=${HADOOP_HOME}/share/hadoop/tools/lib/aws-java-sdk-bundle-1.12.262.jar:${HADOOP_HOME}/share/hadoop/tools/lib/hadoop-
```

```
    aws-3.3.4.jar:${HADOOP_HOME}/share/hadoop/tools/lib/delta-hive-
    assembly_2.11-3.3.0.jar
export JAVA_HOME=/usr/local/openjdk-8

/opt/apache-hive-metastore-3.1.2-bin/bin/schematool -initSchema -dbType
    postgres
/opt/apache-hive-metastore-3.1.2-bin/bin/start-metastore
```

D.3.3 hive-site.xml

```
#!/bin/sh

export HADOOP_HOME=/opt/hadoop-3.3.4
export HADOOP_CLASSPATH=${HADOOP_HOME}/share/hadoop/tools/lib/aws-java-
    sdk-bundle-1.12.262.jar:${HADOOP_HOME}/share/hadoop/tools/lib/hadoop-
    aws-3.3.4.jar:${HADOOP_HOME}/share/hadoop/tools/lib/delta-hive-
    assembly_2.11-3.3.0.jar
export JAVA_HOME=/usr/local/openjdk-8

/opt/apache-hive-metastore-3.1.2-bin/bin/schematool -initSchema -dbType
    postgres
/opt/apache-hive-metastore-3.1.2-bin/bin/start-metastore
```

D.4 Trino Konfigurationen

delta.properties

```
connector.name=delta_lake
fs.native-s3.enabled=true
s3.region=us-east-1
hive.metastore.uri=thrift://metastore:9083
s3.aws-access-key=minio
s3.aws-secret-key=minio123
s3.endpoint=http://minio-host:9000
s3.path-style-access=true
```

hive.properties

```
connector.name=hive
hive.non-managed-table-writes-enabled=true
fs.native-s3.enabled=true
s3.region=us-east-1
hive.metastore.uri=thrift://metastore:9083
s3.aws-access-key=minio
```

```
s3.aws-secret-key=minio123
s3.endpoint=http://minio-host:9000
s3.path-style-access=true
```

config.properties

```
# If enabled - both coordinator and worker will be running on a single instance
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8081
# Set the IP of the server running the trino container
discovery.uri=http://localhost:8081
query.max-memory-per-node=6GB
```

jvm.config

```
-server
-Xmx10G
-Xms10G
-XX:+UseG1GC
-XX:G1HeapRegionSize=32M
-XX:+ExplicitGCInvokesConcurrent
-XX:+HeapDumpOnOutOfMemoryError
-XX:+UseGCOverheadLimit
-XX:+ExitOnOutOfMemoryError
-XX:ReservedCodeCacheSize=256M
-Djdk.attach.allowAttachSelf=true
-Djdk.nio.maxCachedBufferSize=2000000
```

node.config

```
node.environment=docker
node.data-dir=/data/trino
plugin.dir=/usr/lib/trino/plugin
```

D.5 Skripte

D.5.1 Datenaufnahme

```
# SparkSession
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, IntegerType, StringType
import os
from pyspark.sql.functions import *
```

```
from delta import configure_spark_with_delta_pip
from delta.tables import DeltaTable

spark = SparkSession.builder \
    .appName("lakehouse") \
    .master("spark://spark-master:7077") \
    .config("spark.delta.columnMapping.mode", "name") \
    .config("spark.sql.catalogImplementation","hive") \
    .config("spark.sql.warehouse.dir","s3a://hive/") \
    .config("spark.sql.hive.metastore.version","3.1.3") \
    .config("spark.sql.hive.metastore.jars","path") \
    .config("spark.sql.hive.metastore.jars.path","file:///opt/spark/hive/jars/*") \
    .config("spark.sql.legacy.charVarcharAsString", True) \
    .config("spark.sql.sources.partitionOverwriteMode", "dynamic") \
    .config("hive.metastore.warehouse.dir","s3a://hive/") \
    .config("spark.hive.metastore.schema.verification","false") \
    .config("hive.exec.dynamic.partition", "true") \
    .config("hive.exec.dynamic.partition.mode", "nonstrict") \
    .enableHiveSupport() \
    .getOrCreate()

# Minimierung des LOGS
spark.sparkContext.setLogLevel("ERROR")
log4jLogger = spark._jvm.org.apache.log4j
logger = log4jLogger.LogManager.getLogger("LOGGER")
logger.setLevel(log4jLogger.Level.INFO)

# Liste der Datenbanken, in denen jeweils Tabellen mit denselben Namen existieren
databases = [
    "fidus_and",
    "fidus_hbs",
    "fidus_hrs",
    "fidus_mhw",
    "fidus_oaf",
    "fidus_sub",
    "fidus_sue",
    "fidus_wrw"
]

# Liste der Tabellennamen, die in jeder dieser Datenbanken existieren
tables = [
    "behdatei",
    "praxarzt",
    "patdat",
```

```
    "adrver",
    "begrun",
    "ekktexte",
    "arbehdat",
    "rechkopf",
    "rechpos",
    "ckeytabs",
    "patinfo"
]

# Erstelle (falls nicht vorhanden) die Zieldatenbank f r die zusammengef hrten Tabellen
spark.sql("CREATE DATABASE IF NOT EXISTS fidus_joined")

# Iteriere  ber  alle Tabellennamen
for tbl in tables:
    union_df = None
    print(f"Processing table: {tbl}")

    # F r jede Datenbank: lade die Tabelle und f ge per Union hinzu
    for db in databases:
        full_table_name = f"{db}.{tbl}"
        print(f"  Reading {full_table_name}")
        try:
            df_temp = spark.table(full_table_name)
            if union_df is None:
                union_df = df_temp
            else:
                union_df = union_df.union(df_temp)
        except Exception as e:
            print(f"    Error reading {full_table_name}: {e}")

    if union_df is not None:
        # Zeige das Ergebnis
        print(f"Final result for table {tbl}:")

        # Optional: Repartitioniere das DataFrame auf 100 Partitionen, um die Schreibperformance zu verbessern
        #union_df = union_df.repartition(200)

        # Speichere das zusammengef hrte DataFrame als Delta-Tabelle
        output_path = f"s3a://silver/fidus/fidus_joined/{tbl}/"
        union_df.write.format("delta").mode("overwrite").option("path", output_path) \
            .saveAsTable(f"fidus_joined.{tbl}")
        print(f"Saved union of table {tbl} to Delta table fidus_joined.{
```

```
    tbl} at {output_path}")
93          union_df.unpersist()
94      else:
95          print(f"No data found for table {tbl} in any database.")
```

D.5.2 Transformation

```
1  # SparkSession
2  from pyspark.sql import SparkSession
3  from pyspark.sql.types import StructType, StructField, IntegerType,
       StringType
4  import os
5  from pyspark.sql.functions import *
6  from delta import configure_spark_with_delta_pip
7  from delta.tables import DeltaTable
8
9  spark = SparkSession.builder \
10     .appName("lakehouse") \
11     .master("spark://spark-master:7077") \
12     .config("spark.delta.columnMapping.mode", "name") \
13     .config("spark.sql.catalogImplementation","hive") \
14     .config("spark.sql.warehouse.dir","s3a://hive/") \
15     .config("spark.sql.hive.metastore.version","3.1.3") \
16     .config("spark.sql.hive.metastore.jars","path") \
17     .config("spark.sql.hive.metastore.jars.path","file:///opt/spark/hive
       /jars/*") \
18     .config("spark.sql.legacy.charVarcharAsString", True) \
19     .config("spark.sql.sources.partitionOverwriteMode", "dynamic") \
20     .config("hive.metastore.warehouse.dir","s3a://hive/") \
21     .config("spark.hive.metastore.schema.verification","false") \
22     .config("hive.exec.dynamic.partition", "true") \
23     .config("hive.exec.dynamic.partition.mode", "nonstrict") \
24     .enableHiveSupport() \
25     .getOrCreate()
26
27 # Minimierung des LOGS
28 spark.sparkContext.setLogLevel("ERROR")
29 log4jLogger = spark._jvm.org.apache.log4j
30 logger = log4jLogger.LogManager.getLogger("LOGGER")
31 logger.setLevel(log4jLogger.Level.INFO)
32 database = "fidus_joined"
33 spark.sql(f"USE {database}")
34
35 # APK - kpi#10 Anzahl_Arzt-Patienten-Kontakte
36 spark.sql(f"USE {database}")
```

```
df = spark.sql("""
WITH
  bh_data AS (
    SELECT
      KSPATNR AS patnr,
      KSBEHDAT AS datum,
      arzt AS schein_arzt,
      row_number() OVER (PARTITION BY KSPATNR, KSBEHDAT ORDER BY arzt) AS rn
    FROM behdatei
    WHERE arzt <> ''
  ),
  rp_data AS (
    SELECT
      rk.PATNR AS patnr,
      rp.LEISTDATUM AS datum,
      rp.arzt AS rechnung_arzt,
      row_number() OVER (PARTITION BY rk.PATNR, rp.LEISTDATUM ORDER BY rp.arzt) AS rn
    FROM rechkopf rk
    JOIN rechpos rp ON rk.RKOPFID = rp.RKOPFID
    WHERE rp.arzt <> ''
  ),
  ckey AS (
    -- Hier wird der erste gefundene Standort ermittelt
    SELECT first(SCHLUESSEL) AS Standort
    FROM ckeytabs
    WHERE OBJEKT = 'OSG_STANDORT'
  )

SELECT
  month(e.datum) AS Monat,
  weekofyear(e.datum) AS KW,
  year(e.datum) AS Jahr,
  e.datum AS Datum,
  CASE
    WHEN p.VERST = 8 THEN 'Privat'
    ELSE 'Kasse'
  END AS Versicherungsstatus,
  CASE
    WHEN (bh.schein_arzt IS NULL AND rp.rechnung_arzt IS NULL) THEN ckey.Standort
    WHEN bh.schein_arzt IS NULL THEN right(rp.rechnung_arzt, 3)
    ELSE right(bh.schein_arzt, 3)
  END AS Standort,
  count(*) AS Anzahl,
```

```
80    date_format(current_date(), 'yyyy-MM-dd') AS Lieferdatum,
81    weekofyear(current_date()) AS LieferKW
82  FROM ekktexte e
83  LEFT JOIN patdat p ON p.PATNR = e.PATNR
84  LEFT JOIN (
85    SELECT patnr, datum, schein_arzt
86    FROM bh_data
87    WHERE rn = 1
88  ) bh ON e.patnr = bh.patnr AND e.datum = bh.datum
89  LEFT JOIN (
90    SELECT patnr, datum, rechnung_arzt
91    FROM rp_data
92    WHERE rn = 1
93  ) rp ON e.patnr = rp.patnr AND e.datum = rp.datum
94  CROSS JOIN ckey
95  WHERE
96    (e.fduschl IN ('IVOMMDR:', 'IVOMMDL:', 'OPDKARR:', 'OPDKARL:', '
       OPDAARR:', 'OPDAARL:')
97      OR e.fduschl LIKE 'MBefA%')
98    AND e.datum > '2021-04-01'
99    AND dayofweek(e.datum) NOT IN (1,7)
100   AND e.datum BETWEEN add_months(current_date(), -3) AND current_date()
101 GROUP BY
102   e.datum,
103   CASE
104     WHEN p.VERST = 8 THEN 'Privat'
105     ELSE 'Kasse'
106   END,
107   CASE
108     WHEN (bh.schein_arzt IS NULL AND rp.rechnung_arzt IS NULL) THEN ckey
       .Standort
109     WHEN bh.schein_arzt IS NULL THEN right(rp.rechnung_arzt, 3)
110     ELSE right(bh.schein_arzt, 3)
111   END
112 ORDER BY
113   e.datum,
114   CASE
115     WHEN p.VERST = 8 THEN 'Privat'
116     ELSE 'Kasse'
117   END,
118   CASE
119     WHEN (bh.schein_arzt IS NULL AND rp.rechnung_arzt IS NULL) THEN ckey
       .Standort
120     WHEN bh.schein_arzt IS NULL THEN right(rp.rechnung_arzt, 3)
121     ELSE right(bh.schein_arzt, 3)
122   END;
```

```
""")
spark.sql("CREATE DATABASE IF NOT EXISTS fidus_kpi")
spark.sql("USE fidus_kpi")
df.write.format("delta").mode("overwrite").option("path", 's3a://silver/
    fidus/kpi/kpi_10_apk').saveAsTable("kpi_10_apk")
spark.read.format("delta").load("s3a://silver/fidus/kpi/kpi_10_apk").
    show(5)
```

Eidesstattliche Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel benutzt wurden, und - alle Stellen der Arbeit, die wortwörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht wurden. Die vorliegenden Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung dieser Arbeit sowie die zusätzliche elektronische Fassung in anonymisierter Form gem. § 7 Abs. 10 RP0 stimmen inhaltlich überein.

Hamburg, 26.03.2025
Ort, Datum


Unterschrift