Hochschule für Technik
und Wirtschaft Berlin
University of Applied Sciences

Konzeption und prototypische Entwicklung einer hybriden Speicherlösung am Beispiel Datalakehouse bei KWS SAAT SE & Co. KGaA

Bachelorarbeit

zur Erlangung des akademischen Grades:

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Angewandte Informatik*

Erstgutachter: Prof. Dr. Christin Schmidt
Zweitgutachter: Prof. Dr. Tatiana Ermakova

vorgelegt von **Tovias Nunez Borrero**

Berlin, 17. März 2026

Abstract

Diese Abschlussarbeit im Bachelorstudiengang Angewandte Informatik befasst sich mit der Erweiterung eines Datalakehouse-Ansatzes durch eine hybride Speicher- und Zugriffsschicht. Ziel ist es, die Machbarkeit, sowie die Vor- und Nachteile einer solchen Architektur zu analysieren und zu bewerten. Hierfür wurde zunächst ein systematischer Überblick über aktuelle Technologien, Frameworks und Konzepte im Bereich Datenarchitekturen erarbeitet.

Anschließend erfolgte eine prototypische Erweiterung der bestehenden Datenarchitektur in einer isolierten Testumgebung, um zentrale Aspekte wie Skalierbarkeit, Interoperabilität, Performance und Sicherheit praktisch zu untersuchen. Die Ergebnisse zeigen, dass hybride Architekturen insbesondere durch die flexible Trennung von Speicher- und Verarbeitungsschicht sowie die Nutzung offener Standards deutliche Vorteile in Bezug auf Anpassungsfähigkeit und Zukunftssicherheit bieten. Gleichzeitig wurden Herausforderungen identifiziert, insbesondere hinsichtlich Integrations- und Wartungsaufwand.

Die Arbeit verbindet wissenschaftliche Analyse mit praktischer Anwendung. Am Beispiel der KWS SAAT SE & Co. KGaA werden konkrete Erkenntnisse gewonnen, die Unternehmen bei der Bewertung und Einführung moderner Datenplattformen unterstützen können.

Inhaltsverzeichnis

1 Einführung **1**

1.1 Gesellschaftliche, wirtschaftliche und politische Rahmenbedingungen der digitalen Datenverarbeitung . . . 1

1.2 Hintergrund der Arbeit . . . 2

1.3 Problemstellung und Forschungsfrage . . . 3

1.3.1 Problemstellung . . . 3

1.3.2 Forschungsfrage . . . 3

1.4 Aufbau der Arbeit . . . 4

2 Grundlagen **5**

2.1 Historische und organisatorische Einordnung der Datenverarbeitung . . . 5

2.2 Definition klassische und moderne Datenspeicher . . . 6

2.3 Relevante Datenarchitekturen . . . 7

2.4 Motivation für hybride Speicherlösungen . . . 14

2.5 Arten hybrider Speicherlösungen und Zugriffsschichten . . . 15

2.6 Anforderungen und Herausforderungen hybrider Speicher- und Zugriffsschichten . . . 17

2.7 Bewertungskriterien für hybride Speicher- und Zugriffslösungen . . . 18

3 Methodologie **19**

3.1 Anforderungserhebung und –analyse . . . 19

3.1.1 Vorgehen zur Anforderungsanalyse . . . 19

3.1.2 Beschreibung des Ist-Zustandes bei der KWS . . . 22

3.1.3 Analyse bestehender Frameworks und Technologien . . . 24

3.1.4 Methodische Ableitung des Soll-Zustands . . . 28

4 Konzeption, Entwurf **30**

4.1 Definition geeigneter Kriterien . . . 30

4.2 Evaluation geeigneter Technologien . . . 31

4.3 Auswahl und Begründung . . . 32

4.4 Architekturübergreifende Struktur- und Betriebsprinzipien . . . 34

4.5 Architekturentwurf . . . 35

5 Umsetzung (Implementierung) **38**

5.1 Technischer Rahmen . . . 39

5.2 Systemkomponenten des Prototyps . . . 40

5.3 Datenfluss und Prozessabläufe . . . 44

5.4 Implementierung der Speicher- und Zugriffsschicht . . . 48

5.5 Sicherheitsaspekte im Prototyp . . . 52

5.6 Einschränkungen des Prototyps . . . 53

5.7 Test . . . 54

6 Darstellung und Bewertung der Ergebnisse **56**
6.1 Prototypische Validierung der Lösung 56
6.2 Kostensimulationsmodell: Ergebnisse und Annahmen 57
6.3 Bewertung des Prototyps anhand definierter Kriterien 60
6.4 Identifizierte Optimierungspotenziale 61
6.5 Gültigkeit und Abgrenzung der Ergebnisse 62
6.6 Beantwortung der Forschungsfrage 63

7 Zusammenfassung **65**

Quellenverzeichnis **67**

Abkürzungsverzeichnis **74**

Anhang **I**
A.1 Bewertung möglicher Technologien I
A.2 Systemkomponenten und Schnittstellen VIII
A.3 KI-Verzeichnis IX

Abbildungsverzeichnis

2.1 Grafische Veranschaulichung der Datenstrukturierungsformen 7
2.2 Grafische Veranschaulichung der Data-Warehouse-Architektur 8
2.3 Zonenbasierte Architektur eines Data Lakes 9
2.4 Data-Lakehouse-Architektur 12
2.5 Lambda-Architektur 13
2.6 Kappa-Architektur 13
2.7 Datenhaltung in Public Clouds im Vergleich zu traditionellen Rechenzentren 15

3.1 PoC-Datalakehouse und parallele Nutzung von Oracle Blob Storage 20

4.1 C4-Container-Diagramm der Zielarchitektur 36

5.1 C4-Übersichtsdiagramm des Hybrid Data Sync Systems 39
5.2 C4-Container-Diagramm der Systemarchitektur 41
5.3 Statusmodell der Chunk-Verarbeitung 45

6.1 Trino-Abfragen im Validierungstest 56
6.2 Monitoring der ETL-Fehlermetrik 57
6.3 Kostensimulationsmodell 60

Tabellenverzeichnis

2.1 Vergleich relevanter Datenarchitekturen 14
2.2 Vergleich klassischer Datenarchitektur mit föderierter Lakehouse-Architektur 16

3.1 Vergleich verschiedener Datenarchitektur-Ansätze 26

4.1 Kriterien und Aspekte für die Auswahl geeigneter Technologien 30
4.2 Ausgewählte Kerntechnologien für die prototypische Umsetzung 31

5.1 Systemebenen und reale Beispielwerte aus dem PoC. 51

Listings

5.1 Idempotente UPSERT-Funktion zur Aktualisierung des Chunk-Status . . . 42
5.2 Ereignisbasierte Weiterleitung von RabbitMQ-Nachrichten an die WebUI über WebSockets . . . 43
5.3 Vereinfachter Codeauszug der ereignisgesteuerten ETL-Pipeline . . . 45
5.4 Batchweiser Streaming-Export einer PostgreSQL-Tabelle in Parquet-Dateien nach MinIO . . . 49
5.5 Automatisierte Registrierung von Parquet-Ordnern als Iceberg-Tabellen via Spark . . . 50

6.1 Parametrische Kostenberechnung einer Hybrid-Lakehouse-Architektur . . . 58

1 Einführung

1.1 Gesellschaftliche, wirtschaftliche und politische Rahmenbedingungen der digitalen Datenverarbeitung

Digitale Souveränität ist ein Begriff, der verstärkt in den Jahren zwischen 2010 und 2020 immer mehr an politischer Bedeutung gewonnen hat. Im Jahr 2021 appellierten führende EU-Staats- und Regierungschefs an die EU-Kommission, die digitale Souveränität als politisches Leitbild in Europa stärker zu forcieren. Dieser Impuls war ein klares politisches Signal, welches dazu führte, dass beim Gipfeltreffen zur sogenannten digitalen Souveränität Europas, im November 2025 in Berlin, alle EU-Mitgliedsstaaten eine gemeinsame Erklärung zur digitalen Souveränität verabschiedet haben. Darüber berichtet die Organisation mit dem Namen Digital Council auf ihrer Internetseite. Die Organisation beschäftigt sich mit der Analyse von politischen, wirtschaftlichen, technologischen und gesellschaftlichen Themen und ihre Professionalität wird z.B. für Studien und Strategien herangezogen. Der Digital Council berichtet, dass diese gemeinsame Erklärung aller EU-Mitgliedsstaaten eine europäische Vorstellung von Wettbewerbsfähigkeit und wirtschaftlicher Widerstandsfähigkeit abbildet. Etwas, was früher im digitalen Kontext weniger Aufmerksamkeit erhalten hatte. Er beschreibt den Wandel technischer Souveränität und stellt fest, dass dieses Thema mittlerweile eine wichtige Angelegenheit in europäischen politischen Debatten geworden ist und sogar handelspolitisch betrachtet wird. In diesem Zusammenhang werden unter anderem auch Optionen zukünftiger EU-Beschränkungen von Unternehmen außerhalb der EU geprüft. Interessant an der Erklärung erscheint, dass die Organisation in ihrem Onlinebericht dazu teilt, dass für die Beteiligten noch kein einheitliches Verständnis über die Definition für digitale Souveränität existiert. Unstrittig ist dabei, wie stark dieses Thema nicht nur im digitalen Kontext an Bedeutung gewinnt. Seit Jahren – und mit zunehmender Intensität – prägen digitale Dienste den Alltag auf eine selbstverständliche und dominante Weise. Fragen zur Datenhoheit und Cloud-Souveränität stehen dabei aktuell für die EU-Mitgliedsstaaten im Fokus. Dies mit dem Ziel, Europa und seine Wirtschaft von externen Akteuren unabhängiger zu machen und die eigene Eigenständigkeit zu stärken. Der Bericht verweist zudem, auf die unterschiedlichen Auffassungen zwischen den europäischen Akteuren und außereuropäischen Technologieanbietern im Zusammenhang mit Souveränität und digitalen Regeln und verweist auf die daraus entstehenden Spannungen bei Fragen der Datennutzung und -hoheit sowie Cloud-Souveränität.Vgl. [A2026].

Die Europäische Kommission hatte bereits im Jahr 2022 in ihrem Entwurf für eine Richtlinie mit dem Titel „Regulation of the European Parliament and of the Council on harmonised rules on fair access to and use of data“ abgebildet, dass die Digitalisierung zunehmend mit größeren Datenmengen einhergeht und reine Cloud-Lösungen Abhängigkeiten und Unsicherheiten für Unternehmen bergen. Dieser Entwurf bildete die Grundlage des sogenannten EU Data Act, der 2024 auch formal verabschiedet wurde. Der EU-Gipfel im November 2025 ist daher nicht als Auslöser für den EU Data Act zu verstehen, sondern als Signal

dafür, Gesetze zu regeln. Vgl. [E2022]. Der EU Data Act ist ein Teil der EU-Datenstrategie und gibt seither den Regelungsrahmen u.a. bezüglich Zugangs- und Nutzerrechten, sowie Cloud- und Datenverarbeitungsdienste vor. Zusammen mit diversen Förderinitiativen wird damit ein rechtlicher, technischer und politischer Rahmen geschaffen, damit hybride Speicherlösungen in Europa kosteneffizient, souverän und interoperabel eingesetzt werden können. Seit September 2025 haben Unternehmen ein Recht auf Wechsel & Portabilität. Cloud-Anbieter müssen demnach Migration ermöglichen und Daten können so jederzeit aus der Cloud ins eigene Rechenzentrum oder zu einem anderen Anbieter verschoben werden.

Zusammengefasst, verspricht der Data Act für Unternehmen, die hybride Lösungen einsetzen möchten, mehr Sicherheit. Besonders auch deshalb, weil seit letztem Jahr die Pflicht für Anbieter besteht, Portabilität, Interoperabilität und Notfallregelungen zu gewährleisten. Ab dem Jahr 2027 soll sogar ein vollständiger Kostenabbau für Datenbewegung beabsichtigt sein. Vgl. [B2026]; [E2023].

Die vorangegangene Einordnung stellt den politischen und wirtschaftlichen Rahmen dieser Abschlussarbeit dar und gibt einen groben Eindruck, welche thematischen Schnittstellen zu den Rahmenbedingungen gezogen werden können.

1.2 Hintergrund der Arbeit

Die KWS SAAT SE & Co. KGaA ist ein weltweit führendes Familienunternehmen mit über 5000 Mitarbeitenden in der Pflanzenzüchtung und spezialisiert sich auf die Entwicklung leistungsfähiger Saatgutlösungen. Die Kerngeschäftsfelder sind im Schwerpunkt Zuckerüben, Mais, Getreide, Gemüse und andere Kulturen wie Öl- und Sonnenblumensamen. Um genetische Eigenschaften zu analysieren, Züchtungsmethoden zu optimieren und fundierte Entscheidungen für die Pflanzenentwicklung zu treffen, setzt das Unternehmen auf moderne Datenanalyse- und Verarbeitungstechnologien wie z.B. (Data-Driven Breeding).Vgl. (KWS Intranet) [K2026].

Im Zuge der digitalen Transformation arbeitet das Unternehmen mit umfangreichen Datenmengen, die aus genetischen Sequenzierungen und experimentellen Feldstudien, phänotypischen Messungen und Umwelt- und Klimadaten stammen. Die Verarbeitung der Daten erfolgte zunächst über eine zentral eingesetzte Oracle-Datenbank. In dieser wurden insbesondere strukturierte Informationen gehalten. Zusätzlich wurden größere und teilweise semistrukturierte Daten in AWS S3 gespeichert. Der Zugriff auf beide Systeme wurde über Services realisiert, die sowohl die On-Premises-Datenbank als auch den Cloud-Speicher integrierten.

Die Systemlandschaft war insgesamt in getrennte Verarbeitungsschritte aufgebaut, weshalb eine einheitliche Nutzung und Vergleichbarkeit der verschiedenen Datenquellen nur eingeschränkt möglich war. Die eingesetzte Lösung führt jedoch zu hohen Kosten, unzufriedenen Nutzern und der Duplizierung von Daten im gesamten Unternehmen und daraus ergibt sich auch die Problemstellung, die im Rahmen dieser Abschlussarbeit beantwortet werden soll.

Im Rahmen des projektbegleitenden Vorhabens bei der KWS SAAT SE & Co. KgaA, sollten zunächst Erkenntnisse und Ergebnisse für die Konzeption und prototypische Entwicklung einer hybriden Speicherlösung am Beispiel Data Lakehouse gesammelt und ausgewertet

werden. Alle in dieser Abschlussarbeit enthaltenen projektbezogenen Informationen und Ergebnisse wurden über sechs Monate hinweg, im Rahmen regelmäßiger Teammeetings mit den Experten aus der Abteilung Research & Development (R&D), sowie durch eigenständige Recherche auf internen Plattformen und Informationskanälen der KWS erarbeitet. Die gewonnenen Erkenntnisse wurden durch eigenständige Konzeption und prototypische Umsetzung konkretisiert sowie im Rahmen gezielter Validierungs- und Funktionstests überprüft.

Eine Herausforderung im Projektverlauf lag in der zeitnahen Bereitstellung notwendiger Zugriffsrechte auf interne Systeme und Datenquellen. Aufgrund verteilter Zuständigkeiten und paralleler Projektprioritäten, kam es insbesondere bei Recherchen und Auswertungen vereinzelt zu zeitlichen Verzögerungen.

1.3 Problemstellung und Forschungsfrage

1.3.1 Problemstellung

Die Daten, die bei KWS verarbeitet werden, umfassen sowohl strukturierte als auch semistrukturierte Formate. Bereits mit Projektstart stand für KWS fest, dass ein rein relationales Datenbankmanagementsystem, nicht langfristig geeignet ist, um den steigenden Anforderungen des Business und Marktes gerecht zu werden. Die Daten aus den bereits zuvor genannten Bereichen bilden eine hohe Datenheterogenität, gekennzeichnet von großen Dateien, semistrukturierten Formaten und strukturierten Tabellen. Dabei wird Oracle von der KWS als zentrales Datenbanksystem für strukturierte, geschäftsrelevante Daten genutzt.

Die Datenverarbeitung war über mehrere interne Prozesse und technische Komponenten verteilt organisiert. Eine einheitliche, standardisierte Verarbeitungsschicht war dabei nicht durchgängig erkennbar. Die unterschiedlichen Zugriffswege führten dazu, dass Daten je nach System und Anwendungsfall separat abgefragt oder repliziert werden mussten, was die Vergleichbarkeit und Konsistenz von Analysen zusätzlich erschwerte. Eine zentrale Herausforderung bestand darin, stark wachsende, heterogene und datenintensive Analyseprozesse in einer fragmentierten Systemlandschaft effizient, skalierbar und kostentransparent speichern zu können und zugreifbar zu machen. Dabei sollen bestehende On-Premises-Systeme nicht aufgegeben werden. Die KWS hatte sich mit dem Projekt das Ziel gesetzt, eine kostenattraktivere Speicherlösung zu definieren, die große Datenmengen speichern kann, um u.a. die Datenhoheit zu gewinnen. Jedoch war zu Beginn des Projektes noch nicht zu erkennen, welche zentrale Rolle ein Kostensimulationsmodell haben wird. Zu Beginn stand im Vordergrund, einen tieferen Einblick in die IT-Infrastruktur der KWS zu erhalten.

1.3.2 Forschungsfrage

Denklogisch stellt sich daher die nachstehende zentrale Forschungsfrage, die mit dieser Abschlussarbeit Beantwortung finden soll. Wie kann ein Data Lakehouse um eine Speicher- und Zugriffsschicht erweitert werden, die eine effiziente performante und skalierbare Datenverwaltung sowohl für Cloud-Daten als auch für On-Premises-Daten in einem System ermöglicht?

Neben dieser zentralen Forschungsfrage werden allerdings auch relevante, untergeordnete Fragestellungen beleuchtet mit der Absicht, diese auch beantworten zu können. Dazu zählen unter anderem folgende Fragen: Wie stellen sich die Kosten im Vergleich zu alternativen Lösungsansätzen dar? Wie entwickeln sich die Kosten im Vergleich zu historischen Kostenstrukturen? Welche Lösung stellt aus wirtschaftlicher Sicht die kostengünstigste Option für die KWS dar? Welche technologischen Anforderungen, Voraussetzungen und Grenzen sind zu berücksichtigen?

Die Beantwortung der zentralen und untergeordneten Fragen sollen dazu beitragen, dem Ziel der KWS näher zu kommen. Im Fokus steht dabei, eine fundierte Entscheidungsgrundlage für die konzeptionelle Erweiterung der bestehenden On-Premises-Datenarchitektur um eine Cloud- und On-Premises-übergreifende Speicher- und Zugriffsschicht zu erarbeiten.

Diese Abschlussarbeit verbindet die Theorie mit einer prototypischen Implementierung und liefert Erkenntnisse zur technischen Umsetzbarkeit sowie zu Performance-Aspekten. Die gewonnenen Erkenntnisse können zur Orientierung von anderen Unternehmen herangezogen werden, die ähnliche architektonische Fragestellungen im Kontext hybrider Datenarchitekturen untersuchen. Die Arbeit folgt einem systematischen methodischen Vorgehen, um geeignete Technologien und Ansätze für die Erweiterung der bestehenden Datenarchitektur, um eine hybride Speicher- und Zugriffsschicht zu identifizieren und zu bewerten.

Im Rahmen der Evaluierung stellte für KWS die Gegenüberstellung der Kosten unterschiedlicher Datenmengen einen entscheidenden Aspekt dar. Mithilfe eines eigens konzipierten und programmierten Kostensimulationsmodells ist es gelungen, Variablen wie Datenvolumen, Datentransfers und Datenabrufe quantitativ darzustellen, historische Entwicklungen zu visualisieren und zukünftige Szenarien realitätsnah zu prognostizieren. Insbesondere orientiert sich der Inhalt dieser Abschlussarbeit an relevanten Prinzipien der Software- und Datenarchitektur im Ergebnis z.B. an internationalen Standards, akademischer Forschung und/oder langjähriger industrieller und etablierter Software- und Datenarchitekturpraxis. Dafür wurden relevante Akteure und Quellen herangezogen, die diese Prinzipien geprägt bzw. weitergetragen haben.

1.4 Aufbau der Arbeit

Diese Abschlussarbeit ist wie folgt strukturiert: Kapitel 2 vermittelt theoretischen Grundlagen und ordnet das Thema der Arbeit in den aktuellen Stand der Forschung ein. Kapitel 3 analysiert die Anforderungen und legt den Schwerpunkt auf geeignete Technologien.

Das darauffolgende und zentrale Kapitel 4 und Kapitel 5 beschreibt die Konzeption und die prototypische Umsetzung der Zielarchitektur. Im vorletzten (Kapitel 6) werden die Ergebnisse dargestellt und Kapitel 7 fasst die gewonnenen Erkenntnisse zusammen, nimmt Bezug auf die Fragestellung und gibt einen Ausblick auf mögliche Weiterentwicklungen.

2 Grundlagen

Im Folgenden werden die theoretischen Grundlagen der Datenspeicherung, Datenarchitekturen sowie Arten hybrider Speicher- und Zugriffsschichten einschließlich Funktionalität der Lösungen und Herausforderungen näher beleuchtet. Ein weiterer zentraler Bestandteil dieses Kapitels sind die Anforderungen an hybride Speicherlösungen und Zugriffsschichten sowie deren Bewertungskriterien wie z.B. Sicherheit, Governance, Betrieb und Wartung.

2.1 Historische und organisatorische Einordnung der Datenverarbeitung

An dieser Stelle bietet sich ein kurzer Rückblick auf die bisherige Entwicklung an, im Speziellen, in die Zeit vor der digitalen Transformation vor der Jahrtausendwende (2000). Damals genutzte Lösungen für die Speicherung von Daten waren meist zentral stark kontrollierte Systeme, Magnetbänder und Mainframes. Computersysteme wurden in dieser Zeit primär als rechnerische Arbeitsmittel verstanden, und die analogen Lösungen boten nur wenig Flexibilität, Speichervolumen und Batchverarbeitung.

Mit Beginn der Digitalisierung in Europa zwischen 1990 und 2010 wurden rationalere Datenbanken, Client-Server-Architekturen und On-Premises-Infrastrukturen erschaffen, die jedoch durch starre Datenmodelle und hohe Kosten die Unternehmen vor Herausforderungen stellten. Vgl. [CML2014]. Ab dem Jahr um 2010 und mit dem Cloud-Zeitalter sowie der Datenexplosion, entwickelten sich auch neue Speicher- und Verarbeitungsmodelle. Unternehmen mussten zunehmend heterogene Datenquellen, darunter Big Data, Internet of Things (IoT), Web- und Sensordaten in skalierbaren Architekturen wie Data Warehouses und Data Lakes integrieren.

Das weltweit generierte Datenvolumen ist explosionsartig gewachsen, dass ist die Kernaussage einer Studie der Boston Consulting Group aus dem Jahr 2023. Darin heißt es, dass bereits bis zum Jahr 2021 das generierte Datenvolumen in der Welt auf 84 Zettabyte (ZB) angestiegen ist. Vgl. [B2023]. Von diesem Ergebnis berichtet auch der Autor Serra in seinem Buch und vergleichbare Entwicklungen sind ebenfalls in der Literatur von Pradeep Menon beschrieben. Dieses Wachstum überfordert klassische relationale Data-Warehouse-Architekturen und verstärkt den Bedarf an skalierbaren, hybriden Datenarchitekturen wie dem Data Lakehouse. Serra verweist dabei auf Schätzungen von Forschern, die eine jährliche Wachstumsrate (CAGR) von 21% zwischen den Jahren 2021 und 2024 vorhersagten. Damit hätte sich das generierte Datenvolumen in der Welt von 84 ZB auf 149 ZB erhöht. Vgl. [S2025], S. 27; [M2022], S. 5-11.

Diese Schätzungen untermauern den Anspruch von besonders modernen datengetriebenen Unternehmen, die zunehmend Plattformen einsetzen müssen, die große und diverse Datenmengen flexibel analysieren und effizient speichern können. Dabei wünschen sich

Unternehmen auch die Möglichkeit, Datenmengen für z.B. Maschine Learning bereitstellen zu können, doch spätestens an dieser Stelle stoßen manche Unternehmen schon auf die ersten Herausforderungen.

Darüber hinaus nutzt eine Vielzahl der Unternehmen bewusst oder historisch gewachsen mehrere Systeme parallel. Diese parallelen Architekturen verursachen häufig erhöhte Betriebs- und Wartungskosten und erhöhen durch redundante Datenhaltung die Komplexität für den Nutzer. Klassische Datenarchitekturen wie Data Warehouses oder Data Lakes stoßen insbesondere bei stark wachsenden, heterogenen Datenbeständen hinsichtlich Skalierbarkeit, Kostenstruktur und flexibler Verarbeitung unterschiedlicher Datentypen an ihre Grenzen. Entscheiden sich Unternehmen für eine Data-Lake-Architektur, profitieren sie in der Regel von flexiblen und kosteneffizienten Speicherlösungen. Gleichzeitig können jedoch Herausforderungen hinsichtlich Datenqualität, Governance und performanter analytischer Abfragen entstehen.

Unumstritten ist, dass Unternehmen zunehmend Ergebnisse binnen weniger Sekunden benötigen, auch dann, wenn hinter den Daten große Tabellen stehen, viele Benutzer gleichzeitig zugreifen müssen und komplexe SQL-Abfragen durchführt werden. Vgl. [S2025], S. 27–28. Im Wegweiser von Priebe et al. wird das Data Lakehouse als ein möglicher Lösungsansatz für die beschriebenen Herausforderungen moderner Datenarchitekturen dargestellt. Daraus wird deutlich, dass unterschiedliche Ausprägungen und Implementierungsvarianten von Lakehouse-Architekturen existieren. Sie kombinieren grundsätzlich die Stärken der Data Warehouse- und Data Lake – Welt. Moderne Lakehouse-Formate wie Delta Lake, Apache Iceberg oder Apache Hundi sollen dabei die ersehnte Zuverlässigkeit bieten. Sie sind dafür bekannt, hoch performante Abfragen zu tätigen und bieten damit die technischen Grundlagen, die Unternehmen in der heutigen Zeit erwarten. Vgl. [PNM2022], S. 32

Um die bisherigen Ausführungen inhaltlich besser einzuordnen, wird im Folgenden unter den Abschnitt 2.2 bis Abschnitt 2.7 näher auf die theoretischen Grundlagen eingegangen, denn diese dienen als Grundlage für die spätere Implementierung des Prototyps und die Validierung.

2.2 Definition klassische und moderne Datenspeicher

Die Definition für klassische und moderne Datenspeicher wurde aus der konzeptionellen Beschreibung nach Serra abgeleitet. Die Vergleiche und Abgrenzungen in seiner Literatur schlussfolgern, dass klassische Datenspeicher überwiegend relationale, schemabasierte Systeme sind, die für transaktionale Verarbeitung (OLTP) und klar strukturierte Datenmodelle entwickelt wurden. Die klassischen Datenspeicher folgen dem sogenannten Schema-on-Write Prinzip. Das bedeutet, sie charakterisieren sich durch vordefinierte Tabellen, Spalten und Datentypen, und die Struktur der Daten wird dabei vor dem Speichern festgelegt. Es können also nur Daten gespeichert werden, die genau diesem Schema entsprechen. Das bedingt, dass Änderungen an der Struktur sehr aufwändig sein können, da dementsprechend z.B. die Tabellen angepasst werden müssen. Typische Vertreter für klassische Datenspeicher sind relationale Datenbanken wie MySQL, Oracle Database oder klassische Data Warehouses. Diese Datenspeicher sind in ihrer Skalierbarkeit bei sehr großen Datenmengen

eingeschränkt, mit semistrukturierten oder unstrukturierten Daten sind sie eher unflexibel und im Umgang mit wachsenden Datenquellen haben Unternehmen einen hohen Integrations- und Mirgrationsaufwand. Vgl. [S2025], S. 25 ff.

Moderne Datenspeicher hingegen folgen dem Schema-on-Read Prinzip, was sich im Gegensatz zu klassischen Datenspeichern als flexible und hochskalierbare Speicherlösungen definiert. Moderne Datenspeicher wurden für die Verarbeitung heterogener, großer und verteilter Datenmengen entworfen. Die Speicherung und Verarbeitung erfolgt in modernen Datenspeichern losgelöst voneinander. Der Charakter moderner Datenspeicher ist geprägt von horizontaler Skalierbarkeit (Scale-out), Trennung von Storage und Compute. Sie unterstützen strukturierte, semi- und unstrukturierte Daten und erst bei einer Abfrage wird entschieden, welche Felder gelesen werden, wie die Daten zu interpretieren sind und welche Datentypen verwendet werden.

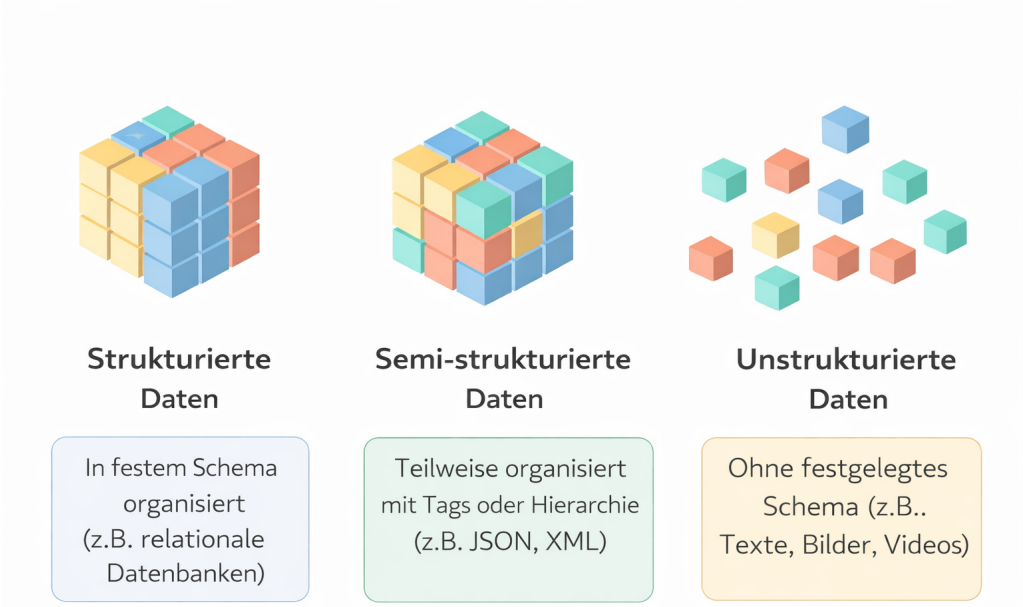


Abbildung 2.1: Grafische Veranschaulichung der unterschiedlichen Datenstrukturierungsformen
Quelle: Eigene Darstellung (KI-unterstützt erstellt) in Anlehnung [AAA2016].

Typische Infrastrukturen, die das Schema-on-Read-Prinzip ermöglichen, sind Object-Storage-Lösungen wie Amazon S3 oder MinIO, die als Grundlage für Data-Lake- Architekturen dienen. Aufbauend darauf kombinieren Data-Lakehouse-Ansätze flexible Speicherung mit strukturierten Zugriffskonzepten. Solche Architekturen eignen sich insbesondere für datenintensive Anwendungen wie Machine Learning oder Big-Data-Analysen. Vgl. [S2025], S. 40–41

2.3 Relevante Datenarchitekturen

Die wichtigsten Architekturen, die wesentliche Herausforderungen in der modernen Datenverarbeitung lösen sollen, sind u.a. Klassische Data-Warehouse-, Data-Lake-, Data-Lakehouse-, Kappa-, und Lambda-Architekturen und jede dieser Architekturen bedient

einen eigenen und deutlich abgegrenzten Bedarf. Im Rahmen des Projektes, bei der KWS und in dieser Abschlussarbeit stehen allerdings nicht alle diese Architekturen im Fokus. Vor diesem Hintergrund werden nachstehend nur die Architekturen in der Tiefe beleuchtet und priorisiert, die für diese Abschlussarbeit relevant sind. Die übrigen Architekturen werden zum Ende dieses Unterkapitels mit weniger Detailtiefe erläutert.

Die Definition klassischer Data-Warehouse-Architekturen basiert auf der konzeptionellen Beschreibung analytischer Datenarchitekturen in der Literatur, insbesondere von dem Autor Dr. Farkisch. Eine klassische Data-Warehouse-Architektur definiert sich als zentrales System zur Analyse von Daten. Diese Architektur integriert Daten aus unterschiedlichen Quellensystemen, dabei müssen die Daten strukturiert und beispielsweise in Tabellen mit festen Spalten enthalten sein. Die Daten kommen aus operativen Systemen z.B. OLTP und können aus internen Unternehmenssystemen als auch aus externen Quellen stammen.

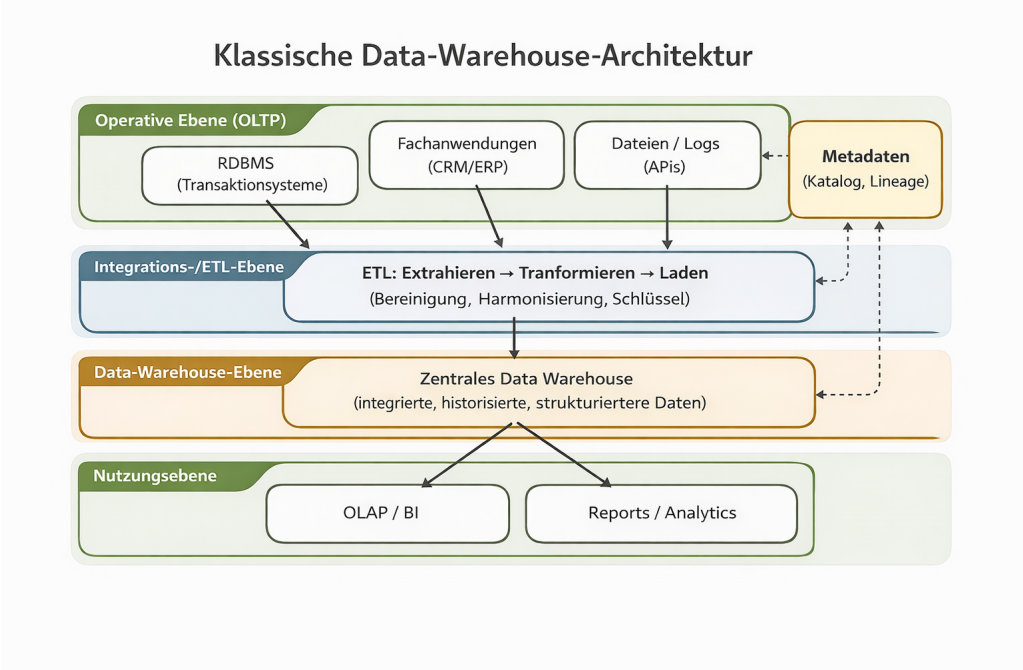


Abbildung 2.2: Klassische Data-Warehouse-Architektur mit zentraler Datenhaltung und vorgelagerter Datenintegration.
Quelle: Eigene Darstellung (PowerPoint) in Anlehnung [AAA2016]

Die gezeigte Abbildung 2.2 visualisiert den grundlegenden Aufbau und die Funktionsweise auf architektonischer Ebene, wie sie unter anderem auch von Serra im Kontext seiner Fachliteratur über Datenarchitekturen beschrieben wird. Die Abbildung verdeutlicht die klare Trennung zwischen operativen Quellsystemen und einer dazwischenliegenden ETL-Schicht sowie einer zentralen Datenhaltung im Data Warehouse. In der ETL-Schicht erfolgt der Datenintegrationsprozess, bei dem Daten extrahiert, transformiert und in das Zielsystem geladen werden. Dadurch werden sie für analytische Anwendungen strukturiert und aufbereitet. Auf die zentrale Datenhaltung greifen anschließend Analyse- und Reporting-Werkzeuge wie OLAP oder Business-Intelligence-Systeme zu. Die zentrale, harmonisierte Speicherung unterstützt zudem Governance-Anforderungen, indem sie konsistente und integrierte Daten für Auswertungen bereitstellt. Vgl. [F2011]. Diese klassische Architektur wird für gewöhnlich auf relationalen Datenbanksystemen umgesetzt und sie muss auch bei datenintensiven und komplexen Analyseanfragen leistungsfähig, stabil und zuverlässig sein. Das System ist

vor allem für Berichte, Auswertungen und Business-Intelligence-Anwendungen ausgelegt, damit Unternehmen bessere Entscheidungen treffen können. In der Praxis werden Ladevorgänge oftmals entgegen der Hauptnutzungszeit durchgeführt, um laufende Analysen im Tagesbetrieb nicht zu stören. Diese soeben beschriebenen Eigenschaften sowie damit einhergehende Grenzen, insbesondere hinsichtlich Flexibilität bei neuen Datentypen, des Aufwands bei Schemaänderungen und der Skalierbarkeit bei sehr großen Datenmengen, trugen maßgeblich zur Weiterentwicklung moderner hybrider Datenarchitekturen bei. Eine dieser Entwicklungen ist die Data-Lakehouse-Architektur.Vgl. [PNM2022].

Nach Dittmar und Schulz orientieren sich Data-Lake-Architekturen wiederum an verschiedenen Zonen. Wobei jede Zone einen bestimmten Verarbeitungsgrad der enthaltenen Daten beschreibt. Im Standardmodell sind oftmals die Zonen aufeinander aufgebaut und die genaue Ausgestaltung und die Anzahl der Zonen kann dabei verschieden sein. Die nachstehende Abbildung 2.3 visualisiert die Data-Lake-Architektur und veranschaulicht nachstehende Erläuterungen.

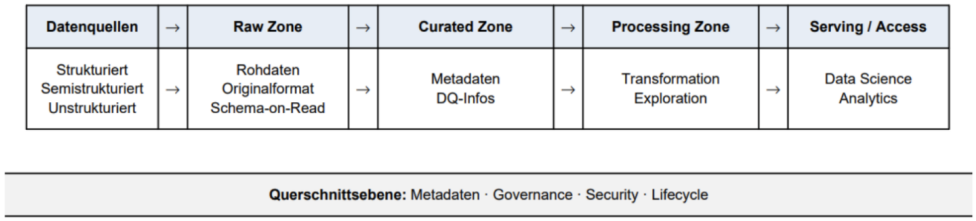


Abbildung 2.3: Zonenbasierte Architektur eines Data Lakes.
Quelle: Eigene Darstellung (PowerPoint) in Anlehnung [DS2023], S. 157 - 166

Die Raw Zone (Rohdatenzone) ist die erste Zone in dieser Architektur, in der alle Daten aus den Quellensystemen aufgenommen werden. In dieser Zone liegen strukturierte, semi- und unstrukturierte Daten unverändert im Originalformat ab und werden weder fachlich noch technisch umgeformt. Die Curated Zone (aufbereitete Zone) sorgt für die technische Aufbereitung der Rohdaten, so dass diese les- und nutzbar gemacht werden. Die Daten bleiben inhaltlich unverändert, jedoch kommen an der Stelle zusätzliche Informationen hinzu z.b. technische Qualitätsinformationen oder einfache Formatkonvertierungen. Die fachliche Verarbeitung der Daten erfolgt dann in der Processing Zone, in der die Kombinierung, Filterung und Aggregation der Daten stattfindet. Diese Zone ist stark Use-Case-orientiert und Daten können dort mehrfach und unterschiedlich verarbeitet werden, obwohl die Ergebnisse nicht zwingend einheitlich sein müssen. Die vierte Zone nennt sich Access Zone. Dort werden die Daten bereitgestellt und für Zugriffe und Performance optimiert. Die dort enthaltenen Daten stammen aus den vorgelagerten Zonen. Diese Eigenschaften eignen sich für z.B. explorative Analysen, Data Science und Machine Learnings.

Jedoch bringt auch diese Architektur gewisse Nachteile mit sich. Es besteht die Gefahr inkonsistenter Ergebnisse, wenn Governance-Strukturen nicht zentral geregelt sind und eine übergreifende fachliche Harmonisierung der Daten fehlt. Dadurch entsteht für Unternehmen ein hoher Aufwand, die vorhandenen Daten zu verstehen und deren Qualität zu bewerten. Durch fehlende oder unzureichende Governance-Konzepte sinkt die Nutzbarkeit der Daten, während gleichzeitig die Komplexität und das Risiko steigen, dass große Datenmengen

gespeichert werden, deren Herkunft, Bedeutung und Aktualität nicht mehr nachvollziehbar sind. Dieser Zustand wird auch als Data Swamp bezeichnet.

Neue Datenquellen können ohne vorgelagerte Strukturierung oder aufwendige Modellierungsprozesse in das System integriert werden. Die Daten werden zunächst im ursprünglichen Rohzustand gespeichert, wodurch eine hohe Flexibilität erhalten bleibt. Darüber hinaus ist es möglich, Informationen auf ihrer detailliertesten Ebene abzulegen, ohne dass zuvor eine Aggregation oder Verdichtung erfolgen muss. Die Architektur unterstützt batchbasierte und kontinuierliche Datenverarbeitung sowie eine Bereitstellung der Daten ohne große Verzögerung. Daten können über mehrere Jahre gespeichert werden und eine konsequente Datenerhaltung und Verarbeitung ist in dieser Architektur möglich, in dem sie Daten in unterschiedlichen Zonen mehrfach verarbeiten kann, ohne die Rohdaten zu verlieren. Daten werden nicht einmalig vollständig aufbereitet, sondern schrittweise und je nach Anwendungsfall unterschiedlich transformiert. Data-Lake-Architekturen ermöglichen durch ihre Eigenschaften freie, explorative Analysen, Simulationen und Prognosen.

Sie stellen Daten im Self-Service für unterschiedliche Nutzergruppen bereit und sie erlaubt den Zugriff über verschiedene Analysewerkzeuge und Technologien. Ihre Tauglichkeit im Zusammenhang mit großen Datenmengen und die Flexibilität bei der Erweiterbarkeit machen diese Architektur zu einer stabilen Lösung. Zusammenfassend erlaubt die Data–Lake–Architektur eine flexible Speicherung heterogener Datenformate. Daten können zunächst ohne feste Vereinheitlichung gespeichert werden. Damit sie später dennoch sinnvoll ausgewertet werden können, sind jedoch Metadaten und klare Regeln für den Umgang mit den Daten erforderlich. Ohne eine konsequente Nutzung von Metadaten und klaren Governance-Regeln, wird die Struktur unübersichtlich und Daten können nicht mehr sinnvoll genutzt werden.Vgl. [DS2023], S. 157 - 166.

Unter Berücksichtigung der Literatur von Inmon [ILS2021], Priebe et al. [PNM2022] und Serra [S2025] lässt sich feststellen, dass sich aus den jeweiligen Beschreibungen die grundlegende Idee einer Data-Lakehouse-Architektur ableiten lässt. Die Grundidee besteht darin, die flexible und skalierbare Speicherung eines Data Lakes mit den strukturierten, qualitätsgesicherten und transaktionalen Eigenschaften eines Data Warehouse in einer gemeinsamen Architektur zu vereinen. Sie stellt ein Datenökosystem zur Verfügung, welches Einheitlichkeit, Effizienz und Skalierbarkeit vereint und dabei alle fünf Phasen eines Datenlebenszyklus abdeckt. Diese zukunftsorientierte Architektur versteht es, strukturierte, semistrukturierte und unstrukturierte Daten zu verarbeiten. Eine Data-Lakehouse-Architektur unterstützt den gesamten Datenprozess von der Aufnahme der Daten über deren Bereinigung und Aufbereitung bis hin zur Analyse und Nutzung in Machine-Learning-Anwendungen. Alle diese Schritte können innerhalb einer integrierten Architektur erfolgen, was ihr wesentliches Merkmal darstellt. Dadurch eignet sich das Data Lakehouse besonders für analytische Prozesse, Business-Intelligence-Anwendungen und Machine-Learning-Szenarien.

Diese Architektur stellt die Zielarchitektur dieser Abschlussarbeit dar und die Grundlage für hybride Speicher- und Zugriffskonzepte. Die Data-Lakehouse-Architektur ist häufig zonenbasiert aufgebaut, und dieses Prinzip kommt ursprünglich aus dem Data-Lake. In der Data-Lakehouse-Architektur werden die Zonen jedoch besser kontrolliert, technisch stärker integriert und enger mit Analyse- und BI-Funktionen verbunden.

Die Abbildung 2.4 ergänzt die nachfolgenden Erläuterungen und veranschaulicht die Zonen und deren Funktion. In der ersten Zone, der sogenannten Landing- oder Ingestion-Zone, werden die Daten aus Quellensystemen aufgenommen und die Speicherung erfolgt häufig

roh oder leicht vorbereitet. Dadurch wird eine schnelle Datenaufnahme gewährleistet und neue Datenquellen können flexibel integriert werden, was als Grundlage für alle weiteren Verarbeitungsschritte benötigt wird. In der darauffolgenden Storage Zone werden alle Daten zentral gespeichert und in tabellenartigen Strukturen verwaltet. Dabei werden spaltenorientierte Dateiformate oder Object Storage verwendet. Diese Zone gewährleistet u.a. Transaktionssicherheit, eine konsistente Sicht auf alle Daten und eine Versionsverwaltung. An dieser Stelle wird deutlich, dass das Data Lakehouse wesentliche Eigenschaften des klassischen Data Warehouse aufgreift. Dazu zählen insbesondere strukturierte Datenmodelle, einheitlich definierte Kennzahlen sowie eine für analytische Abfragen optimierte Verarbeitung. Diese Merkmale werden in die flexible Speicherumgebung des Data Lakes integriert und miteinander kombiniert.

Die dritte Zone wird als Processing Zone bezeichnet. In dieser Zone werden die Daten verarbeitet und transformiert. Dazu zählen unter anderem fachliche Transformationen, Aggregationen, Berechnungen sowie die Ableitung von Kennzahlen aus den zuvor aufbereiteten Daten. In der vierten Zone, die der Modellierung dient, werden die Daten fachlich strukturiert und für die spätere Nutzung vorbereitet. Die zuvor erzeugten Informationen werden dort konsistent, verständlich und performant für Analysen und Visualisierungen bereitgestellt. Das gelingt durch die Definition von Datenmodellen, Sichten (Views) und Zugriffsschichten. Daten werden also dort strukturiert und organisiert, sodass sie einheitlich, leicht verständlich und schnell in der fünften Zone für Visualisierungen und Analysen genutzt werden können. Während die Modellierungsphase die fachliche Struktur der Daten definiert, dient die Nutzungs- und Visualisierungsphase der Analyse und Darstellung dieser Daten zur Entscheidungsunterstützung. Die Abhängigkeit von Metadaten- und Zugriffsschichten stellen neben der hohen konzeptionellen Komplexität dieser Architektur sowie die Notwendigkeit eines guten Governance-Konzepts prägende Schwächen dieser Architektur dar. Bei unzureichender Umsetzung können unter anderem Performance- und Konsistenzprobleme auftreten.Vgl. [S2025], S. 212-218 ; [ILS2021], S. 19-29 ; [J2022], Abschnitt 4.3.3.

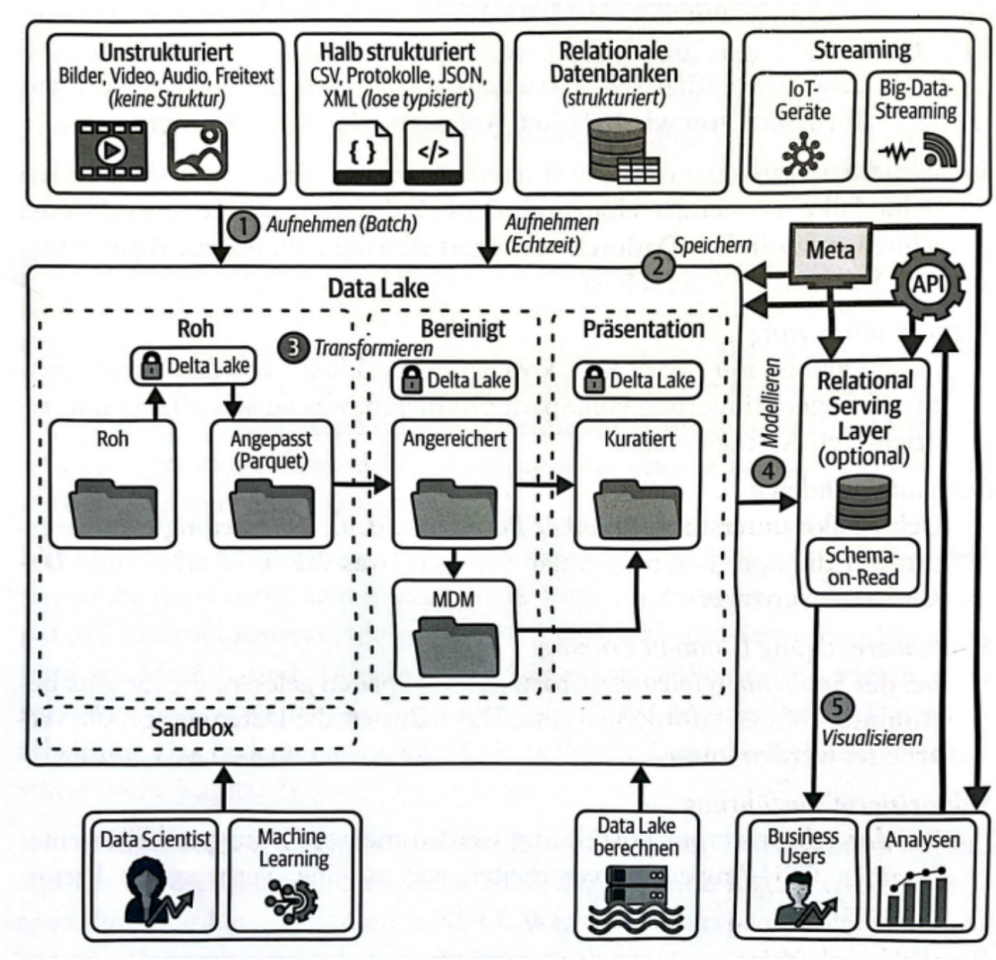


Abbildung 2.4: Data-Lakehouse-Architektur.
Quelle: [S2025], S. 212.

Weitere Architekturansätze wie Lambda und Kappa werden im Folgenden der Vollständigkeit halber dargestellt, jedoch nicht vertieft analysiert, da sie für die Zielsetzung dieser Abschlussarbeit von untergeordneter Bedeutung sind. Die Kappa- und Lambda-Architektur sind zwei Konzepte, die große Datenmengen verarbeiten können, insbesondere historische und aktuelle Daten können damit ausgewertet werden. Serra vertritt die Auffassung, dass beide Architekturen eine wichtige Grundlage für moderne Datenplattformen bilden und sie maßgeblich zur Entwicklung heutiger Data-Lakehouse-Konzepte beigetragen haben.

Die auf der nächsten Seite abgebildete Lambda-Architektur Abbildung 2.5 verdeutlicht, dass Daten über unterschiedliche Verarbeitungspfade verarbeitet werden. Das zugrunde liegende Konzept basiert auf drei zentralen Komponenten. Dabei unterteilt sich das Modell in die Batch-Schicht, die für die Stapelverarbeitung zuständig ist und die parallel existierende Stream-Schicht. Diese sorgt für die Verarbeitung von sehr exakten große Mengen an historischen Daten. Neue Daten können gleichzeitig sofort verarbeitet werden, während ältere Daten im Hintergrund ausgewertet werden. In der Präsentationsschicht werden die Ergebnisse der anderen beiden Schichten dargestellt. Präzise Analysen müssen aufgrund der voneinander entkoppelten Schichten auch in zwei unterschiedlichen Systemen implementiert und gewartet werden. Die technische Umsetzung von Analysen wird dadurch aufwendiger, was sowohl Wartung und Betrieb verteuert als auch die Systemkomplexität erhöht. Zudem wird es schwieriger, einheitliche Governance-Regeln konsequent umzusetzen. Andererseits

kann eine Entkopplung besonders bei hohen Datenmengen und vielen Nutzern sowie häufigen Änderungen auch vorteilhaft sein, wenn sie ein echtes Problem löst (Skalierung, Robustheit usw.).

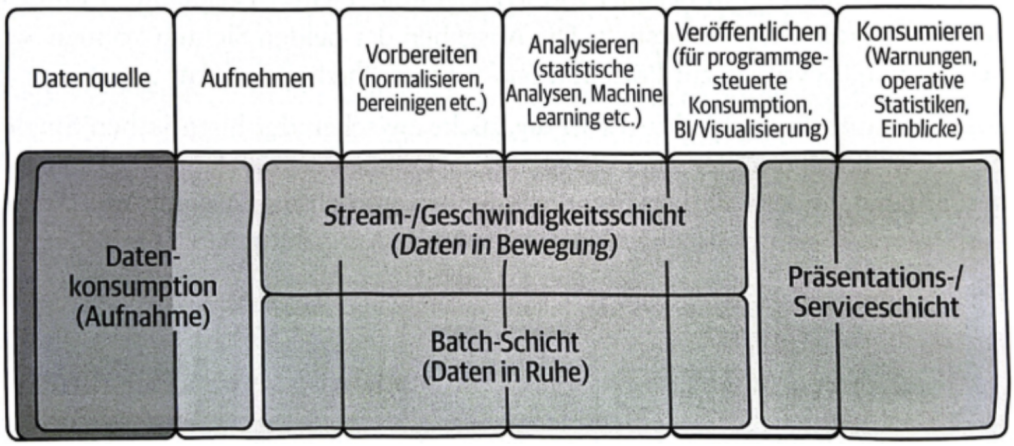


Abbildung 2.5: Lambda-Architektur.
Quelle: [S2025], S. 138.

Im Gegensatz dazu verzichtet die Kappa-Architektur auf den kompletten Batch-Pfad und vereinfacht das vorherige Modell. Diese Architektur ist für die Verarbeitung von Echtzeitdaten und die zeitgleiche Verarbeitung von großen Datenmengen entwickelt worden. Serra fasst die Grundidee dieser Architektur dahingehend zusammen, dass alle Daten in Echtzeit in einem einzigen kontinuierlichen Datenstrom (Ereignisstrom) verarbeitet werden. Historische Auswertungen gelingen durch das erneute Abspielen dieses Datenstroms. Dadurch reduziert sich die Systemkomplexität deutlich und Governance-Regeln wie Schema-Validierung, Zugriffsrechte und Monitoring lassen sich zentraler und konsistenter umsetzen. Allerdings kann die erneute Verarbeitung großer Datenmengen zeit- und ressourcenintensiv sein.

Da die Kappa-Architektur für die Echtzeitverarbeitung ausgelegt ist, eignet sie sich nicht besonders gut für Ad-hoc–Abfragen im Zusammenhang mit großen historischen Daten. Darüber hinaus kann das erneute Abspielen von Datenströmen die Nachvollziehbarkeit einzelner Verarbeitungsschritte erschweren und höhere Anforderungen an die Protokollierung und Versionierung der Daten stellen.Vgl. [S2025], S. 137-141. Die nachstehende Abbildung 2.6 von Serra vermittelt einen Eindruck über den Aufbau einer klassischen Kappa-Architektur.



Abbildung 2.6: Kappa-Architektur.
Quelle: Serra 2025, S. 140.

Eine Übersicht über die wesentlichen Merkmale sowie zentrale Eigenschaften der zuvor beschriebenen Architekturen findet sich in Tabelle 2.1 wieder.

Tabelle 2.1: Vergleich relevanter Datenarchitekturen.
Quelle: Eigene Darstellung (Latex) in Anlehnung an [DS2023]

Architektur	Merkmale	Stärken	Schwächen
Klassisches Data Warehouse	Zentrale relationale Datenhaltung; strukturierte Daten; Schema-on-Write; ETL-Prozesse	Hohe Datenqualität; stabile Performance; BI & Reporting; ACID-Eigenschaften	Geringe Flexibilität; hoher Modellierungsaufwand; nicht geeignet für unstrukturierte Daten
Data Lake	Zentrale Ablage großer Rohdatenmengen; Objektspeicher; Schema-on-Read; heterogene Formate	Hohe Flexibilität; kostengünstige Speicherung; sehr gut skalierbar	Schwache Governance; variable Datenqualität; Performance toolabhängig
Data Lakehouse	Kombination aus Data Lake & Warehouse; Transaktionen; Metadatenmanagement; offene Formate (z. B. Parquet)	Vereint Flexibilität & Analytics; weniger Datenkopien; einheitliche Plattform	Höhere technische Komplexität; abhängig vom Technologie-Stack
Lambda-Architektur	Batch-, Speed- und Serving-Layer; parallele Verarbeitung	Echtzeit- & Batch-Verarbeitung; hohe Skalierbarkeit; fehlertolerant	Hohe Systemkomplexität; doppelte Logik; hoher Wartungsaufwand
Kappa-Architektur	Stream-basierte Verarbeitung; einheitlicher Datenpfad; kein Batch-Layer	Vereinfachte Architektur; Echtzeit-Fokus; weniger Redundanz	Reprocessing historischer Daten komplex; weniger geeignet für tiefe Historienanalysen

2.4 Motivation für hybride Speicherlösungen

Studien aus den letzten Jahren zeigen, dass das globale Datenvolumen in den nächsten Jahren exponentiell ansteigen wird und sich die Datenhaltung immer mehr in die Richtung Cloud orientiert. In diesem Zusammenhang weisen auch Reinsel et al. darauf hin, dass sich bis zum Jahr 2029 das globale Datenvolumen auf bis zu 290 ZB entwickeln wird. Zur Erinnerung, für das Jahr 2024 lagen die Prognosen zum globalen Datenvolumen bei 184 ZB.

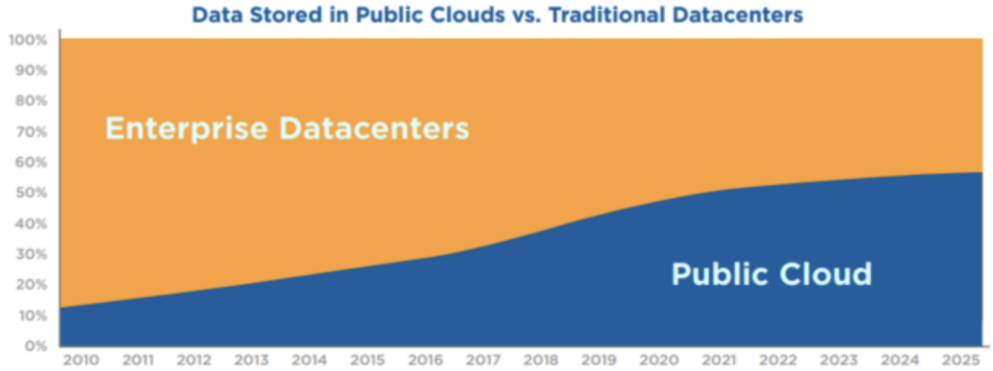


Abbildung 2.7: Datenhaltung in Public Clouds im Vergleich zu traditionellen Rechenzentren.
Quelle: [RGR2018], S. 11.

Mit dem Diagramm in Abbildung 2.7 wird die Verschiebung der Datenhaltung von klassischen Unternehmensrechenzentren in die Richtung der Public-Cloud-Umgebung verdeutlicht und es unterstreicht damit den Trend, dass hybride Speicherlösungen zunehmend an Bedeutung gewinnen werden. Die Gründe für das Wachstum sind vielfältig und liegen insbesondere in der zunehmenden Nutzung von Cloud-Computing, der Entstehung von IoT- und Sensordaten, dem Einsatz von KI- und ML-Anwendungen sowie der fortschreitenden Digitalisierung von Geschäftsprozessen Vgl. [M2022], S. 5-11.

2.5 Arten hybrider Speicherlösungen und Zugriffsschichten

Während Speicherlösungen die physische und technische Implementierung der Datenhaltung beschreiben, definieren Zugriffsschichten die logische Ebene einer Datenarchitektur und regeln den Zugriff auf gespeicherte Daten. Da Speicherung und Verarbeitung voneinander entkoppelt sind, lässt sich die benötigte Rechenleistung flexibel anpassen, ohne die zugrundeliegende Datenhaltung verändern zu müssen.

Über Zugriffsschichten greifen Analysewerkzeuge, Anwendungen oder Nutzer auf gespeicherte Daten zu und das, ohne an die physische Speicherstruktur gebunden zu sein. Hybride Speicherlösungen sind nicht als Produkte zu verstehen, sondern als Architekturkonzepte, die sich aus Speicherort, Technologien, Zugriffsmustern und Governance-Anforderungen kombinieren lassen.

Es gibt eine Vielzahl von hybriden Speicherlösungen und in der Praxis häufig anzutreffende Ausprägungen sind klassische Datenbanken + Object Storage, Data Warehouse + Data Lake, Data Lakehouse mit föderierter Zugriffsschicht sowie On-Premises + Cloud (Hybrid Cloud Storage). Auch bei den Zugriffsschichten gibt es keine feste Anzahl. In der Praxis lassen sich operative-, analytische-, föderierte- sowie Governance- und Metadatenschichten als typische Zugriffsebenen moderner Datenarchitekturen unterscheiden.

Die folgende Betrachtung beschränkt sich bewusst auf die hybriden Speicherlösungen und Zugriffsschichten, die für den Prototyp und die Zielsetzung der Abschlussarbeit maßgeblich sind. Dahingehend werden die zwei wichtigsten hybriden Speicherlösungen für das Projekt bei der KWS und diese Abschlussarbeit in der folgenden Tabelle 2.2 gegenübergestellt.

Tabelle 2.2: Vergleich klassischer Datenarchitektur mit föderierter Lakehouse-Architektur
Quelle: Eigene Darstellung (Latex) in Anlehnung [S2025]

Architekturmerkmal	**Klassische Datenbank + Object Storage**	**Data Lakehouse mit föderierter Zugriffsschicht**
Grundidee	Getrennte Nutzung spezialisierter Speicher für Betrieb und Analyse	Zentrale Analyseplattform mit Anbindung verteilter Datenquellen
Hybriditätsprinzip	Hybrid durch Kombination verschiedener Speichertechnologien	Hybrid durch integrierte Speicher- und Zugriffskonzepte
Rolle der Speicher	Datenbank für operative Prozesse, Object Storage für Analyse	Object Storage als gemeinsame zentrale Speicherbasis
Logische Kopplung	Keine übergreifende logische Sicht auf alle Daten	Einheitliche logische Sicht auf alle angebundenen Daten
Datenorganisation	Daten werden häufig exportiert oder dateibasiert bereitgestellt	Daten werden als strukturierte Tabellen mit definiertem Schema verwaltet
Zugriffskonzept	Unterschiedliche Zugriffspfade je nach System	Zentraler Zugriff über eine gemeinsame Abfrageebene
Analytische Nutzung	Analysen meist auf einzelne Systeme beschränkt	Systemübergreifende Analysen über mehrere Quellen möglich
Datenkopien	Datenkopien häufig aus Integrations- oder Performancegründen	Datenkopien werden weitgehend vermieden
Datenaktualität	Zeitverzögerungen durch Exporte oder Batch-Prozesse	Nahezu aktuelle Auswertungen durch direkte oder föderierte Abfragen
Metadaten	Metadaten verteilt oder manuell gepflegt	Zentrale Verwaltung von Metadaten
Governance	Systemspezifische und uneinheitliche Steuerung	Einheitliche Governance über alle Datenquellen
Erweiterbarkeit	Neue Datenquellen erfordern zusätzlichen Integrationsaufwand	Neue Datenquellen können flexibel angebunden werden
Architektonische Einordnung	Übergangslösung mit begrenzter Integration	Moderne Zielarchitektur für hybride Datenplattformen

Klassische Datenbanken in Kombination mit Object Storage stellen eine einfache Form der hybriden Speicherung dar, bieten jedoch nur begrenzte Möglichkeiten für integrierte Analysen. Daten müssen häufig kopiert oder exportiert werden, was zusätzlichen Aufwand verursacht. Solche Strukturen konnten auch im Rahmen der Analyse der bestehenden Systemlandschaft im Projektkontext bei der KWS beobachtet werden.

Im Gegensatz dazu vereint ein Data Lakehouse mit föderierter Zugriffsschicht die Speicherung und den Zugriff in einer gemeinsamen Architektur. Dadurch können Daten aus verschiedenen Systemen zentral analysiert werden, ohne sie mehrfach zu speichern. Dieser Ansatz bietet mehr Flexibilität und eignet sich besser als moderne Zielarchitektur für hybride Datenplattformen.

Die erste relevante Zugriffsschicht wird die analytische Zugriffsschicht benannt. Sie nimmt eine wichtige Rolle ein, da sie eine einheitliche Schnittstelle für Analysen bereitstellt. Vom Speicher entkoppelt ist die analytische Zugriffsschicht für die Analyse von Daten zuständig, die mittels SQL-Abfragen bereitgestellt werden. Ohne diese Zugriffsschicht würden operative Systeme belastet werden und es käme zu Dopplung von Daten. Darüber hinaus unterstützen analytische Zugriffsschichten unterschiedliche Zugriffsmuster, wie etwa Batch-Verarbeitung, interaktive Abfragen und analytische Workloads.

Die föderierte Zugriffsschicht ist eine entscheidende Schicht, die einen einheitlichen Zugriffspfad auf mehrere Datenquellen bereitstellt. Sie ermöglicht gemeinsame Abfragen über unterschiedliche Speicher hinweg, ohne das Daten physisch zusammengeführt werden müssen. Sie definiert sich als zentrale Komponente hybrider Architekturen. Ein weiteres Highlight dieser Schicht ist, dass sie aktuelle Auswertungen auf operativen Daten ohne zeitliche Verzögerung liefern kann. Sie verringert den Integrations- und Pflegeaufwand für ETL-Strecken, auch wenn sie klassische ETL-Kopien nicht vollständig ersetzt. Die Schicht reduziert jedoch deren Notwendigkeit drastisch, indem sie aktuelle systemübergreifende Analysen ohne physische Datenverschiebung ermöglicht. Insgesamt stellt sie eine zentrale funktionale Komponente dar, um verteilte Datenquellen effizient und konsistent in hybride Architekturen einzubinden.

Mit wachsender Datenmenge steigt der Bedarf an Ordnung, Transparenz und Kontrolle. Die Governance- und Metadatenschicht nimmt dabei eine zentrale Rolle ein. Ohne entsprechende Governance-Konzepte wäre eine kontrollierte Skalierung der Datenverarbeitung kaum möglich, da weder vertrauenswürdige und nutzbare Daten bereitgestellt werden könnten noch sensible Daten ausreichend geschützt wären. Auch gesetzliche Vorgaben und interne Compliance-Regeln könnten ohne entsprechende Governance-Mechanismen nicht zuverlässig eingehalten werden. Die Governance- und Metadatenschicht erfasst Metadaten zentral. In ihr werden unter anderem Zugriffsregeln definiert und Richtlinien, beispielsweise für Zugriffsberechtigungen, automatisiert durchgesetzt. Dadurch wird ermöglicht, dass Daten in hybriden Architekturen kontrolliert, nachvollziehbar und konsistent genutzt werden können. Daraus lässt sich ableiten, dass hybride Architekturen zwar technisch auch ohne Governance realisierbar wären, ihre fachliche Nutzung jedoch stark eingeschränkt wäre.Vgl. [S2025], S. 132 ff. sowie [A2021], insb. S. 2–5. Die Auswahl hängt am Ende von den bestehenden Systemen, Datenarten sowie Anforderungen an Performance und den organisatorischen Rahmenbedingungen ab.

Die dargestellten Funktionalitäten bilden die konzeptionelle Grundlage für die im Prototyp umgesetzten Zugriffsschichten. Die operative Zugriffsschicht wird lediglich als bestehender Bestandteil der Quellsysteme berücksichtigt, ist jedoch nicht Gegenstand der prototypischen Umsetzung.

2.6 Anforderungen und Herausforderungen hybrider Speicher- und Zugriffsschichten

Eine berechtigte Frage in diesem Zusammenhang ist die Folgende. Was muss eigentlich erfüllt sein, damit hybride Architekturen praktisch nutzbar sind und an welcher Stelle kann es Herausforderungen geben? Nicht nur Serra geht in seinem Buch darauf ein, auch eine

Vielzahl empirischer Untersuchungen aus der Literatur greifen dieses Thema auf. Dazu gehört auch die Masterarbeit von Janssen auf Basis von Experteninterviews.

Daraus abzuleiten ist insbesondere das Bedürfnis nach Skalierbarkeit aufgrund der steigenden Abfragekomplexitäten und wachsenden Datenmengen. Um wachende Datenmengen effizient verarbeiten zu können, müssen hybride Architekturen einen einheitlichen Zugriff auf verteilte Daten ermöglichen, ohne operative Systeme zu beeinträchtigen. Im selben Umfang steigen auch die Anforderungen an Datenqualität, Transparenz und Governance, da Daten in der Praxis häufig über mehrere Systeme hinweg genutzt werden müssen. Die zunehmende architektonische Komplexität, insbesondere beim Betrieb förderierter Zugriffsschichten stellt eine zentrale Herausforderung dar und Governance- sowie Metadatenkonzepte können zusätzlichen technischen und organisatorischen Aufwand erfordern.

2.7 Bewertungskriterien für hybride Speicher- und Zugriffslösungen

Für die Einordnung von hybriden Speicher- und Zugriffslösungen können Bewertungskriterien hilfreich sein. Ausgangspunkt dafür ist die grundlegende Funktionsweise der Architekturen sowie deren Fähigkeit, sich in bestehende Systemlandschaften integrieren zu lassen. Die jeweiligen Vor- und Nachteile hybrider Ansätze spielen darüber hinaus auch eine entscheidende Rolle, etwa im Zusammenhang mit Flexibilität, Komplexität und dem Umgang mit verteilten Daten.

Zu diesen funktionalen Aspekten gesellen sich auch sicherheitsrelevante Anforderungen, da der Zugriff auf Daten über mehrere Systeme hinweg kontrolliert und nachvollziehbar erfolgen muss. Ebenso sind Governance-Konzepte sowie der damit verbundene Betriebs- und Wartungsaufwand entscheidend, da sie maßgeblich beeinflussen, wie stabil, transparent und langfristig nutzbar eine hybride Architektur ist. Vgl. [S2025], S. 137 ff. [J2022], S. 56–60.

Neben den konzeptionellen Grundlagen nach Serra und den empirischen Erkenntnissen aus der Arbeit von Janssen flossen auch eigene reflektierte praktische Erfahrungen aus dem Projekt- und Berufskontext in die Ableitung der Anforderungen und Bewertungskriterien ein und bilden zugleich die Grundlage für die spätere Bewertung der im KWS-Projekt betrachteten Lösungsansätze.

3 Methodologie

3.1 Anforderungserhebung und –analyse

3.1.1 Vorgehen zur Anforderungsanalyse

Vor dem Hintergrund eines Proof-of-Concepts wurde im Rahmen der Anforderungsanalyse ganz bewusst ein vereinfachtes Architekturmodell abstrahiert und erstellt, um die wichtigsten Speicher, Systeme und Datenflüsse der KWS innerhalb ihrer Datenverarbeitungs- und Speicherlandschaft abzubilden. In diesem Kapitel wird verdeutlicht, was das Ziel dieser Analyse ist. Es wird ein Verständnis geschaffen, wie Daten gespeichert und verarbeitet werden und was dies für Auswirkungen auf Zugriffe, Datenwege sowie die Komplexität des Systems verursacht. Im weiteren Verlauf wird zur Veranschaulichung ein Architekturdiagramm die Systemlandschaft bildlich untermauern. Schnell war zu erkennen, dass die KWS-Systemlandschaft in einzelnen, klar voneinander getrennten Verarbeitungsschritten aufgebaut ist. Diese Trennung, zwischen analytischer Nutzung und operativer Datenverarbeitung stellt ein zentrales Architekturmerkmal dar.

Beginnend mit dem Longform Storage (editable), in dem die Daten in diesem Speicherbereich noch verändert werden können. Die darin gespeicherten Daten können zusammengefasst und an der Stelle noch umgewandelt werden. Nachfolgend werden die Daten in nicht mehr veränderbare Speicherbereiche verschoben, in diesem Fall „BLOB Storage (readonly)“ genannt. Diese sogenannten read-only-Bereiche bilden ein stabiles Fundament bei der Durchführung von Auswertungen und komplexen Analysen.

Als ein wichtiges Architekturmerkmal zeichnet sich die Speicherlandschaft durch die Möglichkeit aus, unterschiedliche Speicherarten parallel zu nutzen. Dadurch wird eine flexible und bedarfsgerechte Ablage der Daten ermöglicht. Jedoch bedeutet das zugleich, dass dadurch Daten verteilt und unterschiedliche Zugriffsmechanismen und Schnittstellen erforderlich sind. Im Detail wurde neben klassischen dateibasierten Datenspeichern wie z.B. CSV-Dateien auf SharePoint, Netzlaufwerken oder lokalen Datenträgern auch Objektspeicher wie z.B. BLOB Storage (read-only), Datalake (read-only) oder Longform Storage Data Lake (readonly) erkannt. Der Longform Storage DataLake (readonly) bildet den Sockel der Data-Lake-basierten Speicherung und eignet sich vor allem für analytische Auswertungen.

Bei der Betrachtung der Zugriffe auf Daten wurde erkannt, dass Zugriffe eng an die jeweilige Speicherart gekoppelt sind. Mittels Amazon Athena und Galaxy als Such- und Abfragewerkzeuge ist es möglich, Datenanalysen im Data Lake durchzuführen. Für operative Anwendungen werden spezielle Fachservices wie z.B. Marker Data Service, Compressor Service, Aggregation Service verwendet, die als Zwischenschicht zu verstehen sind. Diese greifen nicht direkt auf die Datenablage bspw. auf CSV-Dateien zu und erkennen somit

nicht, wo die Daten gespeichert sind. Ein prägendes Merkmal an der Stelle ist, dass diese Fachservices, die Daten vorher fachlich richtig vorbereiten und bereitstellen.

Nicht zu erkennen ist eine vollständig einheitliche Zugriffsschicht im Unternehmen, die alle Daten auf die gleiche Weise nutzbar macht. Dieser Zustand stellte eine Hürde im Zusammenhang mit der einheitlichen Verwendung verschiedener Datenquellen und der Vergleichbarkeit von Abfragen über mehrere Datenquellen dar. Die bestehende Systemlandschaft und die parallele Nutzung unterschiedlicher Speicherlösungen sind in Abbildung 3.1 exemplarisch dargestellt.

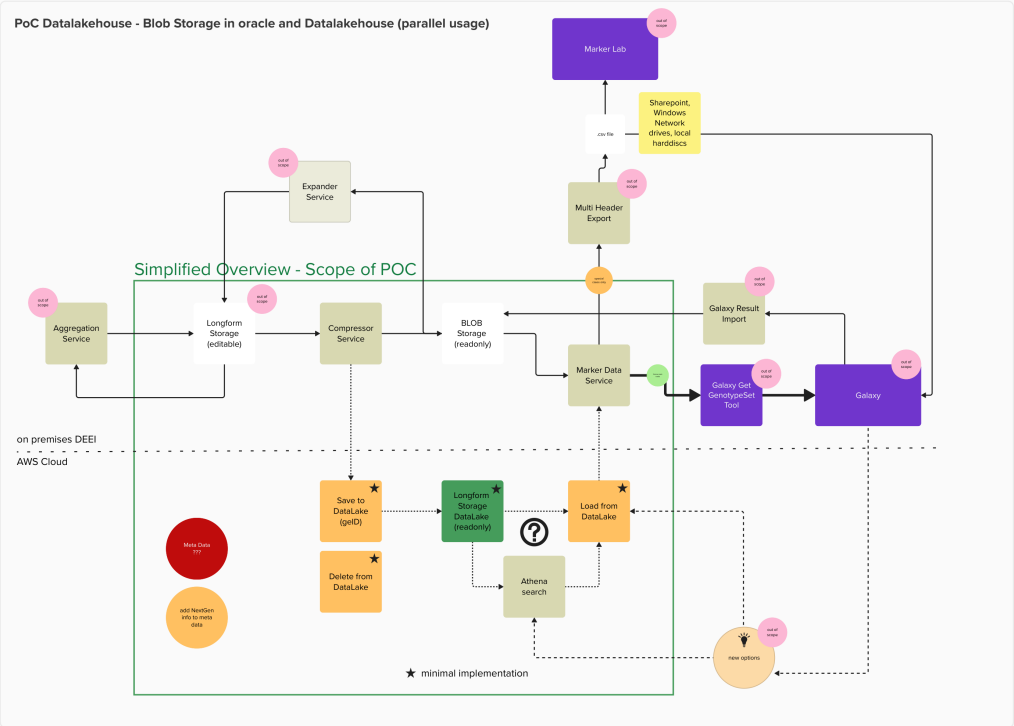


Abbildung 3.1: PoC-Datalakehouse und parallele Nutzung von Oracle Blob Storage
Quelle: KWS SAAT SE & Co. KGaA (Mural)

Mit diesem Ergebnis als Ausgangspunkt im Rahmen der Analyse der bestehenden Datenverarbeitungslandschaft und Speicherlandschaft der KWS, konnten fachliche und technische Anforderungen des Unternehmens zusammengetragen werden. Dabei sollten die Anforderungen zum einen die Bedürfnisse der KWS-Fachbereiche, und zum anderen die technischen Gegebenheiten der bestehenden IT-Systeme berücksichtigen.

Unter den fachlichen Anforderungen im Unternehmen ist zu erkennen, dass täglich unterschiedliche und sehr große Datenmengen aus verschiedensten Quellen verarbeitet werden müssen. Besonders diese Daten erfordern eine effiziente Speicherung, die für Auswertungen verwendet werden. Die KWS hat zudem den Bedarf, verschiedene Datenarten wie strukturierte, teil- und unstrukturierte Daten parallel zu speichern, und zwar in der Form, dass sie auch zuverlässig genutzt werden können. Darüber hinaus ist es notwendig, Daten in Analyse- und Auswertungssystemen zu verwenden, weshalb klare und nachvollziehbare Datenwege benötigt werden. Vor diesem Hintergrund muss eine klare Trennung zwischen analytischer und operativer Datenverarbeitung sichergestellt sein, und diese Anforderung ist in der bestehenden Architektur auch vorhanden, sodass sie als zentrales Kriterium für die weitere Planung bestehen bleibt.

Die technischen Anforderungen lassen sich aus der bestehenden hybriden Systemlandschaft ableiten. Es werden parallel verschiedene Speicher- und Zugriffsschichten verwendet, und KWS sucht eine Lösung, die sich in vorhandene Systeme integrieren lässt und zugleich die Möglichkeit von Verarbeitung und Speicherung voneinander trennt. Das Unternehmen geht auch zukünftig davon aus, mit weiterhin wachsenden Datenmengen konfrontiert zu sein. Um die Lösung auch dauerhaft einsetzen zu können, liegt die Wichtigkeit darin, Anpassungsfähigkeit (Skalierbarkeit) an wachsende Datenmengen sowie Erweiterbarkeit und eine offene Technologieauswahl sicherzustellen. Es wurde schnell deutlich, dass das Unternehmen keine weitere Insellösung verfolgt und zusätzlich eine einheitliche Zugriffsschicht anstrebt, über die Daten unabhängig vom Speicherort auf gleiche Weise abgefragt und genutzt werden können. Die Zugriffsschicht soll dabei umfangreiche Analysen und große Datenmengen sowie die fachlichen Anwendungen unterstützen und einbinden.

Um ein möglichst vollumfängliches und zuverlässiges Verständnis der bestehenden Datenverarbeitungslandschaft und Speicherlandschaft bei der KWS zu erhalten, wurden die benötigten Informationen im Rahmen des Projektes sukzessiv mittels unterschiedlicher Methoden zusammengetragen. Zunächst wurden vorhandene technische Dokumente analysiert. Dazu gehörten Übersichten zur Systemarchitektur, technische Dokumente aus bestehenden oder zurückliegenden Projekten sowie Beschreibungen einzelner Systeme. Dabei lag das Ziel darin, mittels dieser technischen Dokumente einen Überblick zu erlangen, welche Technologien im Einsatz sind und aus welchen Komponenten das System besteht. Die Dokumente haben zudem Erkenntnisse geliefert, wie die Technologien und Systeme miteinander zusammenarbeiten, aber vor allem halfen sie dabei, vorangegangene Architekturentscheidungen sowie die bestehenden Konzepte zur Datenverarbeitung und Datenspeicherung nachvollziehen zu können.

Gleichzeitig haben Gespräche und Team Meetings mit fachlichen Ansprechpartnern im Unternehmen dazu beigetragen, nicht nur die technischen Dokumente besser zu verstehen. In den Gesprächen mit den Experten konnten praktische Erfahrungen aus dem Tagesgeschäft sowie bestehende Herausforderungen und Anforderungen besprochen und analysiert werden. Das war insofern sehr hilfreich, da die technischen Dokumente genau diese wertvollen Informationen nicht oder nur begrenzt beinhaltet haben. Durch diese Methode war es möglich, technische Informationen besser einzusortieren und Annahmen aus der Dokumentenanalyse besser zu verstehen.

Ergänzend zu den Gesprächen und der Analyse technischer Dokumente wurden Speicherorte, Datenflüsse und Zugriffswege detaillierter untersucht. Dabei stand im Vordergrund zu verstehen, wie Daten aus verschiedenen Quellen in die Systeme gelangen, wie die Verarbeitung erfolgt und wo die Daten abgespeichert werden. Darüber hinaus sollte diese Analyse Erkenntnisse liefern, wie die Daten für operative Aufgaben sowie für Analysen verwendet werden. Dabei wurden tatsächliche Speicherorte der Daten, also auch die logischen Zugriffswege, über unterschiedliche Dienste und Abfragesysteme betrachtet. Dies mit dem Zweck, die Abhängigkeiten zwischen Systemen, ihren Schnittstellen und mögliche Unterbrechungen oder Schwachstellen im Datenverarbeitungsprozess zu erkennen. Eine erste Erkenntnis, welche an der Stelle als typisches Beispiel zur Veranschaulichung dienen soll, sind die bei KWS verarbeiteten Marker- und Longform-Daten. Diese werden zunächst in editierbaren Datenspeichern verarbeitet, anschließend in read-only Speicherbereiche überführt und schließlich über unterschiedliche Dienste und Abfragesysteme für operative sowie analytische Zwecke genutzt.

Durch die Kombination dieser drei verschiedenen Analysewege und ihrer Informationsquellen wurde ein zusammenhängendes und umfassendes Bild der vorherrschenden Systemlandschaft erfasst und das Ergebnis diente als Fundament, um davon technische und fachliche Anforderungen für die weitere Konzeption abzuleiten.

Wie zu erkennen ist, sind die fachlichen und technischen Anforderungen ein prägender Bestandteil dieser Abschlussarbeit. Nicht weniger von Bedeutung sind allerdings auch die wirtschaftlichen Ziele, die eine weitere wesentliche Rolle bei der Ermittlung der Anforderungen im Unternehmenskontext der KWS einnehmen.

Während der Analyse wurde wie bereits erwähnt deutlich, dass KWS von wachsenden Datenmengen sowie von immer vielfältigeren Datenformaten ausgeht. Diese Situation könnte langfristig dazu führen, dass die Speicher- und Betriebskosten ansteigen würden, was in einem Zielkonflikt zwischen einer leistungsfähigen technischen Lösung und der Kontrolle der entstehenden Kosten steht.

Das Unternehmen setzte sich das zentrale wirtschaftliche Ziel, eine praktikable und kostengünstige Lösung zur Speicherung verschiedener Datenformate zu finden. Angestrebt ist, ein ausgewogenes Verhältnis zwischen Anschaffungs- und Betriebskosten, praktischem Nutzen und langfristiger Wartbarkeit zu erreichen und nicht etwa jede einzelne technische Komponente bis auf ein Maximum zu optimieren. Insbesondere stand im Vordergrund, eine doppelte Datenhaltung, unnötige Datenkopien und teure Speziallösungen zu vermeiden. Beispielsweise war dahingehend ein erster Gedanke, dass im BLOB Storage und Data Lake (readonly) – Bereich, dieselben Daten gespeichert werden können, somit Dopplungen vermieden werden. Im späteren Verlauf der wissenschaftlichen Arbeit wurde dieser Gedanke jedoch nicht mehr verfolgt, da er mit einem Kostenanstieg einhergehen würde.

Wirtschaftliche Ziele wie eine praktikable und kostengünstige Lösung geht für Unternehmen meistens einher mit dem Bedarf an mehr Transparenz im Zusammenhang mit den Kosten. Auch für die KWS bestand der klare Bedarf darin, aufgrund ihrer unterschiedlichen Speicher- und Zugriffsszenarien zu ersehen, welche Kosten im Einzelnen entstehen. Die Kosten für die Nutzung unterschiedlicher Speicherarten und die bestehende hybride Architektur verursachen für die KWS eine nur schwer nachvollziehbare Kostentransparenz der im Einsatz befindlichen Architekturentscheidungen. Die neue Lösung muss demnach nicht nur kosteneffizient sein, sondern auch ermöglichen, verschiedene Speicher- und Zugriffskonzepte miteinander zu vergleichen.

Das Streben nach einer skalierbaren, ressourcenschonenden und gut betreibbaren Speicherlösung beschreibt zusammenfassend die wirtschaftliche Zielsetzung der KWS. Diese bildet eine wichtige Grundlage für die Auswahl geeigneter Technologien sowie für die Entwicklung des prototypischen Lösungsansatzes, weshalb die Zielsetzung in die weitere Planung verankert und berücksichtigt wurde.

3.1.2 Beschreibung des Ist-Zustandes bei der KWS

Der bis hierher gegebene Einblick hinter die Daten- und Speicherkulissen der KWS hat bereits verraten, dass eine hybride Lösung zur Datenverarbeitung und –speicherung vor Projektstart im Einsatz war. Diese Lösung wird im laufenden Betrieb für tägliche operative sowie für analytische Aufgaben eingesetzt. Sie verbindet klassische, datenbankbasierte Speicher mit Cloud- und Data-Lake-Ansätzen.

In der bestehenden Architektur werden drei objektspeicherbasierte Speicherbereiche verwendet: zwei read-only Speicherbereiche in Form von BLOB Storage und Data Lake für analytische Zwecke sowie ein editierbarer Longform Storage für die operative Verarbeitung. Diese Speicher ergänzen die vorhandenen relationalen Datenbanksysteme, insbesondere Oracle Database, die als klassische Speicher für strukturierte, operative Daten dienen. Diese Kombination erlaubt eine flexible Speicherung verschiedener Datenformate, führt jedoch dazu, dass die Daten auf mehrere Speicherorte verteilt sind und jeweils unterschiedliche Zugriffswege benötigen. In Summe ist diese Kombination eine wichtige Grundlage für die fachlichen Anwendungen und Services.

Die Datenverarbeitung im Unternehmen ist in mehreren getrennten Verarbeitungsschritten unterteilt: Aufnahme, Vorverarbeitung und Speicherung. Der Zugriff erfolgt über spezialisierte Dienste, die jeweils klar definierte Aufgaben haben. Analysen auf Daten im Data Lake erfolgen über spezielle Abfrage- und Suchwerkzeuge wie Athena und operative Anwendungen greifen wiederum über die bekannten datenbanknahen Zugriffswege mittels Marker Data Service, Aggregation Service und Compressor Service auf strukturierte, operative Daten zu. Dies zeigt, dass nicht alle Anwendungen über Data Lake, BLOB Storage oder Athena zugreifen und bestimmte Anwendungen weiterhin klassische, direkte Zugriffe über fachliche Services nutzen. Es existierten im Unternehmen mehrere Speicher- und Zugriffskonzepte nebeneinander. Die hybride Architektur war damit grundsätzlich vorhanden und funktionsfähig, allerdings nicht vollständig vereinheitlicht.

Die Herausforderung besteht also drin, verschiedene Speicher- und Zugriffskonzepte gleichzeitig zu verwenden. Die Daten der KWS sind auf objektbasierten Speichern und Data-Lake-Strukturen verteilt, was eine einheitliche und konsistente Nutzung der Daten erschwert. Die Trennung der o.g. einzelnen Verarbeitungsschritte und die Beteiligung einzelner Komponenten erschwert, dass Datenflüsse über mehrere Systeme hinweg nachzuvollziehen sind, was zudem den Aufwand für Betrieb und Wartung erhöht. Zugriffe sind stark an eingesetzte Technologien gebunden, wodurch eine durchgängige Sicht auf alle Datenbestände fehlt.

Damit wurde deutlich, dass der bisherige Umgang der KWS mit Datenspeicherung, Datenzugriff und Datennutzung pragmatisch und stark an den jeweiligen Anwendungsfällen orientiert war sowie durch die eingesetzten Technologien bestimmt wird. Die Konzepte dahinter wurden im Laufe der Zeit schrittweise erweitert, um auf die Anforderungen neuer Entwicklungen im Business reagieren zu können. Diese Vielfalt ermöglicht eine flexible und leistungsfähige Datenverarbeitung, führt jedoch gleichzeitig zu einer heterogenen Systemlandschaft, die im weiteren Verlauf der Abschlussarbeit konzeptionell vereinheitlicht und weiterentwickelt wird. Eine übergreifende Schicht, die alle Speicher- und Zugriffskonzepte einheitlich verbindet, steht dabei im Fokus.

Diese zuvor ausführlich geschilderte Ausgangslage erschwert insbesondere den Vergleich von Analysen und die Wiederverwendung von Abfragen. Die heterogene Architektur schränkt die Betrachtung der Speicher-, Zugriffs- und Kostenaspekte ein und beschreibt organisatorische und strukturelle Herausforderungen mit der bestehenden Systemlandschaft.

Im Rahmen der sechsmonatigen Projekteinbindung bei der KWS traten Herausforderungen auf, die sowohl technischer als auch methodischer Natur waren. Diese Herausforderungen hatten einen spürbaren Einfluss auf den Ablauf und die Umsetzung der prototypischen Entwicklung.

Im Groben kann von drei prägenden Herausforderungen berichtet werden, die die Arbeit im Rahmen des Projektes anspruchsvoll gestaltet haben. Da wäre zunächst die bereits erläuterte Auswahl und Einrichtung geeigneter technischer Komponenten, um die bestehende hybride Architektur auf Basis des Data-Lakehouse-Ansatzes prototypisch zu erweitern. Die zweite Herausforderung lag in der Automatisierung der Datenspeicherung. Dabei war das Ziel, den gesamten Weg der Daten von der Aufnahme über die Verarbeitung bis hin zur Speicherung und Registrierung im Data Lakehouse möglichst ohne manuelle Eingriffe zu gestalten und das unter Berücksichtigung unterschiedlicher Datenformate, verschiedener Verarbeitungsschritte und mehrerer Speicherziele. Ebenfalls zu berücksichtigen ist die Notwendigkeit, operative und analytische Speicherbereiche klar voneinander zu trennen. In diesen Zusammenhang erwies es sich besonders anspruchsvoll, die Daten zuverlässig und wiederholbar zu registrieren, was für eine stabile und nachvollziehbare Nutzung entscheidend ist. Die dritte Herausforderung im weiteren Verlauf des Projekts war der nicht zu erahnende Fokus, der sich zunehmend auf die wirtschaftliche Betrachtung der Architektur verlagerte. Seitens der KWS bestand ein zunehmendes Interesse daran, die Auswirkungen der Datenspeicherung und Datennutzung nicht nur allgemein, sondern auch anhand konkreter Zahlen bewerten zu können. Deshalb rückte eine realistische Kostensimulation in den Mittelpunkt. Dafür wurde es erforderlich, ein Kostensimulationsmodell zu entwickeln, was sowohl vergangene als auch zukünftige Kostenentwicklungen analysieren konnte.

Diese Summe an Herausforderungen skizzieren an diesem Punkt der Abschlussarbeit die auftretenden Herausforderungen, dass die Entwicklung eines prototypischen Konzepts zur Erweiterung eines Data Lakehouse nicht nur eine technische Aufgabe ist. Der Grad der Automatisierung, die Fähigkeit zur Integration in bestehende Systeme sowie eine transparente Kostenbetrachtung stehen gemeinsam im Fokus und bilden eine zentrale Rolle. In die Weiterentwicklung des Prototyps als auch in die methodische Bewertung der Projektergebnisse flossen diese Erkenntnisse mit ein.

3.1.3 Analyse bestehender Frameworks und Technologien

An diesem Punkt stellte sich die Frage, welche Frameworks und Technologien sich für den Aufbau hybrider Speicher- und Zugriffsschichten in einer Data-Lakehouse-Architektur eignen würden. Das Ziel dieser Analyse bestand darin, technische Lösungen zu ermitteln, mit denen Daten über verschiedene Speicherorte hinweg einheitlich verwaltet und abgefragt werden können. Zudem sollte die Lösung im gleichen Maß auch gut in die bestehende Systemlandschaft der KWS integrierbar sein. Also konzentrierte sich die Analyse auf drei zentrale Technologiebereiche: Tabellen- und Metadatenframeworks, Abfrage- und Zugriffstechnologien und Frameworks zur Datenverarbeitung und Integration.

Zunächst wurden Tabellenformate und Metadatenlösungen für Lakehouses analysiert. Durch sie gelingt es, Dateien in objektspeicherbasierten Systemen wie relationale Datenbanktabellen zu nutzen. Mit dieser Eigenschaft sind große Datenmengen besser zu strukturieren und können gut organisiert werden. Zudem unterstützt diese Lösung bei der Verwaltung von Datenstrukturen sowie bei der Sicherstellung, in dem sie die Versionierung von Daten zulässt. In Summe würde es den Umgang mit großen Datenmengen strukturierter, nachvollziehbarer und kontrollierbarer machen, auch wenn die Datenmengen nicht in klassischen Datenbanken gespeichert sind.

Darauffolgend wurden die Technologien für den analytischen Datenzugriff betrachtet. Sie definieren sich dadurch, dass sie eine gemeinsame Abfrageschicht über verschiedene Speicher- und Datenquellen bereitstellen. Mit der Unterstützung dieser Technologien könnten Daten aus Datenbanksystemen und Data-Lake-Strukturen über eine einheitliche Schnittstelle abgefragt werden, vor allem vor dem Hintergrund der Unterstützung von Metadaten. Attraktiv ist hier die Eigenschaft, wie gut große Datenmengen verarbeitet werden können und wie einfach sich die Lösungen in bestehende Anwendungen und Services integrieren lassen.

Abschließend wurden bei der Analyse die Frameworks zur Datenverarbeitung und –integration näher betrachtet. In hybriden Architekturen sind solche Komponenten besonders bedeutsam. Sie vereinheitlichen unterschiedliche Quellen, überprüfen diese und überführen Daten anschließend in passende Speicherbereiche. Sie unterstützen das Umwandeln, Zusammenführen und Verarbeiten von Daten, häufig auch automatisiert und parallel. Der Fokus lag bei der Analyse auf drei Schwerpunkten: die Möglichkeit zur Automatisierung, Kompatibilität mit zuvor betrachteten Speicher- und Zugriffstechnologien und die Unterstützung verschiedener Datenformate. Für den Aufbau einer hybriden Speicher- und Zugriffsschicht wird es also erforderlich, eine abgestimmte Kombination mehrerer Technologien zu erreichen. In Verbindung mit hybriden Datenarchitekturen gewinnen strukturierte und wiederverwendbare Software- und Systemlösungen immer mehr an Bedeutung, denn sie bieten allgemeine Lösungsansätze für typische Aufgaben wie das Verarbeiten und Speichern von Daten. Solche Frameworks sind unabhängig von konkreten technischen Details einsetzbar.

In dieser Abschlussarbeit wird vor allem Apache Iceberg analysiert, da dieses Framework zulässt, verschiedene Speicher- und Zugriffskonzepte miteinander zu verbinden. Das ist jedoch nicht alles, weshalb nachfolgend erklärt werden soll, warum Apache Iceberg im Fokus steht. Ein zentrales Merkmal ist Trennung zwischen der eigentlichen Datenspeicherung und der logischen Organisation der Daten, da die Daten selbst in skalierbaren, objektbasierten Speichern abgelegt werden können. Ein weiterer Vorteil ist die Wiederverwendbarkeit. Es gibt eine gemeinsame, einheitliche Schnittstelle für den Zugriff auf die Daten, Metadaten sind standardisiert und sind für alle Werkzeuge gleich aufgebaut. Die verschiedenen Analyse- und Verarbeitungswerkzeuge nutzen dieselben Daten, und Daten selbst müssen nicht mehrfach gespeichert werden. Diese Eigenschaften sind besonders wertvoll in hybriden Systemlandschaften, in denen mehrere Technologien gleichzeitig eingesetzt werden.

Darüber hinaus unterstützt Apache Iceberg und solche Frameworks einen einheitlichen Zugriff auf Daten über unterschiedliche Speicherorte hinweg und stellen eine Verbindungsschicht zwischen den Analysewerkzeugen und physischen Speichern dar. Für die KWS ist diese Eigenschaft eine wichtige Voraussetzung für hybride Speicher- und Zugriffslösungen, und in dieser Abschlussarbeit werden solche Software- und Systemlösungen bewusst auf konzeptioneller Ebene betrachtet.

Folglich bestand die Absicht darin, verschiedene Frameworks und Technologien strukturiert und nachvollziehbar vergleichen zu können. Dafür mussten einheitliche Bewertungskriterien festgelegt werden, die die Grundlage für eine faire Bewertung bilden, um unterschiedliche Lösungsansätze für hybride Speicher- und Zugriffsschichten zu analysieren.

Das erste Bewertungskriterium ist der Betriebs- und Wartungsaufwand. Es stellte sich die Frage nach: Wie viel Aufwand verursacht der Betrieb einer solchen Technologie im

Tagesgeschäft? Dabei wurde auf z.B. Installationsaufwand, Konfigurationen, die Durchführung von Updates und das Monitoring des Systems sowie der Umfang manueller Eingriffe im Tagesgeschäft geschaut, denn KWS benötigt eine Lösung, die stabil funktioniert und möglichst wenig Wartung erfordert.

Zusätzlich standen auch die Kriterien Integrationsfähigkeit in bestehende Systeme und Skalierbarkeit und Performance im Fokus. Wie bereits zuvor ausführlich erläutert, ist es dem Unternehmen wichtig, dass es keine umfangreichen Änderungen an bestehenden Systemen vornehmen muss. Zudem liegt der Wunsch im Zusammenhang mit der Skalierbarkeit und der Performance, bei hoher Auslastung und in Verbindung mit wachsenden Datenmengen, auf zuverlässige und stabile Antwortzeiten. Dies kann nur gelingen, wenn das System in die Breite und nicht in die Höhe wächst und die Verteilung von Verarbeitungs- und Abfragemodelle unterstützt.

Die Bewertung der wirtschaftlichen Kriterien bilden den Abschluss an dieser Stelle. Hierzu zählen zum einen indirekte Kosten, die durch eine hohe Komplexität entstehen können oder etwa durch eine eingeschränkte Erweiterbarkeit und/oder einen hohen Wartungsaufwand. Zum anderen gehören auch direkte Kosten zu den wirtschaftlichen Aspekten. Dazu zählt nicht, wie kostenintensiv die neue Lösung per se ist. Gemeint sind Kosten im Rahmen der Infrastruktur und im Zusammenhang mit Servern, Speichern und Netzwerken sowie Kosten für den Ressourcenverbrauch wie z.B. Rechenleistung, Speicherplatz, Netzwerkverkehr und Stromverbrauch. Es muss eine neue Lösung identifiziert werden, die einen guten Kompromiss zwischen Nutzen und Kosten bietet.

Anhand der zuvor definierten Bewertungskriterien gelang eine Gegenüberstellung in tabellarischer Form zu erstellen, die dazu beitragen soll, noch mehr strukturierte Einordnung und Übersichtlichkeit in diesen Abschnitt zu bringen. Darin aufgeführt sind die zuvor erläuterten Technologien, die das Unternehmen in der Vergangenheit im Einsatz hatte, die zum Zeitpunkt des Projektstarts genutzt wurden sowie Technologien, die künftig angestrebt sind. Der Inhalt der nachstehenden Tabelle verdeutlicht, dass die betrachteten Technologien unterschiedliche Schwerpunkte setzen und jeweils spezifische Stärken und Einschränkungen aufweisen. Die in der Tabelle 3.1 dargestellten Einschätzungen basieren dabei zum Teil auch auf den im Projektkontext gewonnenen praktischen Erfahrungen.

Tabelle 3.1: Vergleich verschiedener Datenarchitektur-Ansätze
Quelle: Eigene Darstellung (Latex)

Framework / Technologie	**Betriebs-& Wartungs-aufwand**	**Integrations-fähigkeit**	**Skalierbarkeit Performance**	**Wirtschaft-liche Aspekte**
Lakehouse-Framework A (avisierte hybride Lösung) Offenes, leichtgewichtiges Lakehouse-Framework	Gering bis mittel; weitgehend automatisierbarer Betrieb, zusätzlicher Aufwand für Metadatenverwaltung	Hohe Integrationsfähigkeit durch standardisierte Schnittstellen und Unterstützung externer Query-Engines	Hohe Skalierbarkeit durch Trennung von Speicher und Compute; für große Datenvolumina geeignet	Kosteneffizient bei Nutzung objektbasierter Speicher; moderater Betriebsaufwand

Fortsetzung auf der nächsten Seite

Framework / Technologie	Betriebs-& Wartungs-aufwand	Integrations-fähigkeit	Skalierbarkeit Performance	Wirtschaft-liche Aspekte
Lakehouse-Framework B (Alternative hybride Lösung) Schwergewichtigere Lakehouse-Plattformen (z. B. Delta-Lake-zentrische Setups, Managed Lakehouse-Stacks)	Mittel; erfordert zusätzliche Infrastruktur-komponenten und regelmäßige Wartung	Gute Integrations-fähigkeit, jedoch teilweise eingeschränkte Kompatibilität mit bestehenden Abfragewerkzeu-gen	Sehr gute Performance bei analytischen Workloads; hohe Skalierbarkeit	Höherer Infrastruktur- und Betriebs-aufwand, abhängig vom Einsatzszenario
Klassischer Data-Lake-Ansatz hybride Lösung, wie vor Projektstart	Geringer initialer Be-triebsaufwand; eingeschränkte Verwaltungs-funktionen	Eingeschränkte Integrationsfähig-keit ohne zusätzliche Metadaten- oder Abstraktions-schichten	Gute Skalierbarkeit auf Speicher, jedoch eingeschränkte Performance bei komplexen Abfragen	Niedrige Speicherkosten, jedoch potenziell höherer Aufwand für Erweiterungen
Datenbanknahe Lösung wie vor der Zeit hybrider Lösung bei KWS (Ursprung)	Hoher War-tungsaufwand bei steigenden Datenvolumina	Gute Integration in bestehende operative Systeme, eingeschränkt für Lakehouse-Szenarien	Begrenzte horizontale Skalierbarkeit; Performance abhängig von Systemressourcen	Höhere Kosten bei großen Datenmengen und Skalierung

Der Inhalt der zuvor abgebildeten Tabelle 3.1 soll als Grundlage für die nachfolgende Auswahl und detaillierte Evaluierung geeigneter Technologien unter Berücksichtigung der Unternehmensanforderungen der KWS dienen. Sie zeigt klare Stärken, Schwächen und Grenzen der betrachteten Frameworks und Technologien auf.

Festzuhalten bleibt, das leichtgewichtige Lakehouse-Framework (A) lässt eine schrittweise Erweiterung bestehender Data-Lake-Architekturen zu und stellt strukturierte Metadaten auf objektspeicherbasierten Daten bereit. Es trennt die Speicherung konsequent von der Verarbeitung und erlaubt gleichzeitig einen Zugriff auf diverse Analyse- und Abfragewerkzeuge. Dabei müssten bestehende Speicher- und Zugriffskonzepte nicht vollständig ersetzt werden. Diese Lösung glänzt durch wenig Betriebs- und Wartungsaufwand und eignet sich sehr gut für hybride Szenarien mit paralleler operativer und analytischer Nutzung.

Das Lakehouse Framework B als Alternative im Vergleich betrachtet, bietet umfangreiche Funktionen für die Metadatenverwaltung, Datenpipelines und das Monitoring. Diese Lösung unterstützt ebenso die Skalierung und Trennung von Speicher und Verarbeitung, was allerdings zusätzliche Infrastruktur erfordert und mit einem höheren organisatorischen Aufwand einhergeht. Diese Lösung definiert sich u.a. durch einen höheren Einführungs- und Betriebsaufwand im Vergleich zur leichtgewichtigeren Lösung A. Für kleinere oder schrittweise wachende Architekturen ist diese Lösung nur eingeschränkt geeignet, da der Aufbau und der Betrieb einer solchen Plattform einen deutlich größeren technischen und organisatorischen Aufwand erfordern. Bei sehr großen Datenmengen oder wenn eine zentrale Plattform benötigt wird, kann dieser Ansatz jedoch sinnvoll sein.

Der klassische Data-Lake-Ansatz bietet eine kostengünstige Speicherlösung von großen Datenmengen, hohe Flexibilität bei Datenformaten und ist technisch gut auf der Speicherebene skalierbar. Allerdings birgt diese Lösung Schwächen bei der Strukturierung, dem einheitlichen Zugriff und bei der Metadatenverwaltung und ist aufgrund der fehlenden Zusatzschichten, wenig für komplexe und reproduzierbare Analysen geeignet. Solche Einschränkungen konnten auch im Projektkontext bei der KWS beobachtet werden.

Die reine datenbanknahe Lösung beschreibt sich als zuverlässige, stabile Lösung für strukturierte, transaktionale Daten. Sie lässt sich gut in operative Systeme integrieren, verursacht jedoch hohe Kosten und ist bei wachsenden Datenmengen nur begrenzt skalierbar. Demnach ist diese Lösung für heterogene und sehr große analytische Datenbestände keine erkennbare Option für die KWS mehr. Die Einschätzung stützt sich dabei auch auf interne Dokumentationen der KWS, die zu Beginn des Projekts ausgewertet wurden.

3.1.4 Methodische Ableitung des Soll-Zustands

Die zuvor erforschten Ergebnisse aus dieser Analyse dienen als Grundlage für die anschließende Bewertung und Auswahl geeigneter Technologien im weiteren Verlauf. Ziel ist, ein Verständnis über grundlegende Architekturprinzipien zu erlangen sowie das Potenzial, Speicher- und Zugriffskonzepte zu vereinheitlichen. Diese Einordnung dient als Grundlage für die anschließende Auswahl und Bewertung konkreter Frameworks, die den fachlichen, technischen und wirtschaftlichen Anforderungen gerecht werden sollen. Die im vorherigen Unterkapitel festgelegten Bewertungskriterien gestalten zusammen einen klaren und einheitlichen Rahmen für die anschließende Analyse und Bewertung möglicher Technologien. Dadurch ist eine transparente und vergleichbare Entscheidungsfindung, die sich an den Anforderungen des Unternehmens orientiert, möglich.

Der Soll-Zustand beschreibt folglich, welche avisierte zukünftige Zielarchitektur der Datenverarbeitungslandschaft und Speicherlandschaft bei der KWS geeignet ist. Die Analyse des Ist-Zustandes zeigte eine bestehende hybride und grundsätzlich leistungsfähige Architektur. Allerdings weist die parallele Nutzung unterschiedlicher Speicher-, Verarbeitungs- und Zugriffskonzepte eine hohe strukturelle Komplexität auf. Die Tatsache einer fehlenden Speicher- und Zugriffsschicht sowie die eingeschränkte Transparenz von Lebenszyklen, Datenflüssen und Kostenstrukturen hindern das Unternehmen in Summe daran, die Architektur ganzheitlich zu betrachten und weiterzuentwickeln.

Der konzeptionelle Soll-Zustand setzt genau an diesen Schwächen an, er orientiert sich an den architektonischen Prinzipien, ohne eine konkrete technische Implementierung vorwegzunehmen. Der Soll-Zustand verlangt die Einführung einer vereinheitlichenden, logisch abstrahierenden Ebene, welche bestehende Speicher- und Verarbeitungssysteme funktional miteinander verbindet. Die Trennung zwischen editierbaren Speicherbereichen und read-only Speicherzonen bleibt dabei erhalten. Aus den erhobenen Anforderungen leitet sich ab, dass der Soll-Zustand fachliche und technische Aspekte berücksichtigen muss. Technisch soll eine klare Trennung von Verarbeitung, Speicherung und Zugriff gewährleistet sein, um die Erweiterbarkeit, Integrationsfähigkeit und Skalierbarkeit zu gewährleisten. Fachlich besteht der Anspruch darin, unterschiedliche Datenformate und Datenquellen einheitlich zu verwalten und zuverlässig für Analyse- und Auswertungszwecke bereitzustellen. Nicht zu vergessen sind die wirtschaftlichen Aspekte, die zuvor bereits im

Abschnitt 3.1.2 ermittelt worden, sodass sie an der Stelle nicht nochmals aufgeführt werden müssen.

KWS hat den Anspruch an den Prototypen, dass er in der Lage sein soll, unterschiedliche Speicherorte und Datenformate parallel zu unterstützen, ohne sie technisch oder organisatorisch zu vermischen. Im Fokus steht hierbei die strukturierte Einbindung in ein gemeinsames Zugriffskonzept, das Medienbrüche reduziert und die Konsistenz der Datennutzung erhöht. Zum anderen soll der Prototyp beweisen, dass eine architektonische Entkopplung zu erhöhter Skalierbarkeit, Flexibilität und Wartbarkeit führt. Durch die klare Trennung von Speicherung, Verarbeitung und Zugriff soll bewiesen werden, dass ein solches Architekturprinzip auch innerhalb einer bereits bestehenden hybriden Systemlandschaft praktikabel umzusetzen ist.

Wirtschaftliche und betriebliche Aspekte, die Automatisierung zentraler Datenprozesse sowie die Integrationsfähigkeit der prototypischen Lösungen bilden abschließend drei weitere Zielkriterien mit gleicher Wichtigkeit. Das bedeutet zusammengefasst, dass der Prototyp zentrale Datenprozesse automatisieren kann und dabei eine nahtlose Integration in bestehende Analyse-, Abfrage-, und Nutzungspfade gewährleisten.

Zudem spielt die Möglichkeit der Überführung von konsistenten und nachvollziehbaren Daten in read-only Analysezonen und die transparente Betrachtung von Speichern und Zugriffskonzepten eine prägende Rolle. Darüber hinaus muss der Prototyp als erweiterbares Element der vorhandenen Systemlandschaft funktionieren, bestehende Systeme der KWS nicht beeinträchtigen oder ersetzen und die unterschiedlichen Nutzungs- und Speicherszenarien vergleichbar machen. Das zuvor beschriebene konzeptionelle Zielbild des Soll-Zustandes stellt die zukünftige Weiterentwicklung der hybriden Datenverarbeitungs- und Speicherlandschaft der KWS dar.

Der Prototyp soll die technische Machbarkeit überprüfen, als Entscheidungs- und Bewertungsgrundlage fungieren bzw. als analytisches und konzeptionelles Referenzmodell verstanden werden und nicht als produktionsreife umsetzbare Lösung. Dementsprechend beschränkt sich die prototypische Implementierung bewusst auf einen klar definierten Funktionsumfang und bildet nicht die vollständige bestehende Systemlandschaft ab.

Die im Methodologie-Kapitel dieser Abschlussarbeit durchgeführten Analysen und konzeptionellen Gedanken sind die Grundlage für die darauffolgende Implementierungs- und Evaluationsphase. Insgesamt trägt das Ergebnis in diesem Kapitel dazu bei, die gewonnenen Erkenntnisse in einen belastbaren wissenschaftlichen Kontext einzuordnen und ihre Aussagekraft im Hinblick auf die zentrale Fragestellung zu sichern.

4 Konzeption, Entwurf

Die Konzeption bildet die Grundlage für die technologische Ausgestaltung eines hybriden Data-Lakehouse-Prototyps bei KWS. Durch einen strukturierten Vergleich relevanter Lösungsansätze anhand definierter Kriterien wird eine konsistente und begründete Architekturentscheidung vorbereitet. Im Entwurf wird konkret dargestellt, wie die ausgewählten Technologien im Prototyp miteinander kombiniert werden können. Dabei werden Verantwortlichkeiten, Schnittstellen und Datenflüsse definiert, sodass eine klar strukturierte und logisch aufgebaute Architektur entsteht. Die einzelnen Komponenten sind bewusst entkoppelt konzipiert, um Erweiterbarkeit, Skalierbarkeit und Transparenz zu gewährleisten.

4.1 Definition geeigneter Kriterien

Für die Auswahl geeigneter Technologien wurden im Projekt zunächst die relevanten Anforderungen ermittelt. Die folgende Tabelle 4.1 fasst die daraus abgeleiteten Bewertungskriterien zusammen und zeigt, welche technischen und architektonischen Fragestellungen bei der Evaluierung berücksichtigt werden müssen.

Tabelle 4.1: Kriterien und Aspekte für die Auswahl geeigneter Technologien
Quelle: Eigene Darstellung (Latex)

Kriterium	Abgedeckte Aspekte	Was wird damit geprüft?
Performance	Query-Latenz; Scan-Effizienz; Pruning-Mechanismen; Metadatenoptimierung; Einfluss des Speicherformats; Transaktions-Overhead	Wie effizient analytische Abfragen ausgeführt werden, wie gut unnötige Datenzugriffe vermieden werden und wie stark technische Mechanismen wie ACID oder Versionierung die Ausführungszeit beeinflussen.
Skalierbarkeit	Horizontale Skalierung; Parallelisierbarkeit; Concurrency-Fähigkeit; Ressourcen-Elastizität	Inwieweit das System mit wachsenden Datenmengen und steigender Nutzeranzahl umgehen kann und ob zusätzliche Ressourcen flexibel integriert werden können.
Cloud-/On-Prem-Kompatibilität	Cloudneutralität; Betrieb in Cloud und On-Premises; konsistente Nutzung über Umgebungen hinweg	Inwieweit die Technologie sowohl lokal als auch in Cloud-Umgebungen eingesetzt werden kann und keine starke Anbieterbindung erzeugt.

Fortsetzung auf der nächsten Seite

Kriterium	Abgedeckte Aspekte	Was wird damit geprüft?
Integration in bestehende Systeme	Anbindung relationaler Datenbanken (z. B. PostgreSQL); Integration in ETL-Prozesse; Monitoring- und Observability-Unterstützung	Wie gut sich die Technologie in bestehende Systeme und Prozesse einfügt.
Open-Source-Reifegrad / Vendor-Unabhängigkeit	Aktive Community; regelmäßige Weiterentwicklung; offene Lizenzmodelle; Vermeidung proprietärer Abhängigkeiten	Inwieweit die Technologie langfristig wartbar ist und keine starke Abhängigkeit von einzelnen Anbietern erzeugt.

4.2 Evaluation geeigneter Technologien

Auf Grundlage der zuvor betrachteten Kriterien werden im nächsten Schritt vergleichbare Technologien systematisch gegenübergestellt. Dabei wird analysiert, in welchem Maß die einzelnen Lösungen die formulierten Anforderungen erfüllen und welche architektonischen Stärken sowie mögliche Einschränkungen sich daraus ergeben. Ziel ist es, eine nachvollziehbare und technisch begründete Entscheidungsbasis für die Auswahl einer geeigneten Technologie im Rahmen des hybriden Data-Lakehouse-Konzepts zu schaffen. Eine detaillierte tabellarische Gegenüberstellung aller Schichten, Kriterien und Technologien befindet sich im Anhang A.1. Aus den Ergebnissen der Gegenüberstellung wurden die für den Prototyp ausgewählten Kerntechnologien abgeleitet und in Tabelle 4.2 übersichtlich zusammengefasst.

Tabelle 4.2: Ausgewählte Kerntechnologien für die prototypische Umsetzung
Quelle: Eigene Darstellung (Latex)

Architekturschicht	Auswahl	Kurzbegründung
Tabellenformat (Lakehouse)	Apache Iceberg	Engine-neutral, ACID-Unterstützung, effizientes Metadata- und Partition-Pruning sowie gute Integration in Trino.
Object Storage	MinIO	On-Prem einsetzbar, S3-kompatibel und integrierbar in die containerisierte PoC-Umgebung.
Query Engine	Trino	Niedrige Latenz für interaktive SQL-Abfragen, native Iceberg-Integration und hohe Concurrency-Fähigkeit.
ETL / Processing	Apache Spark	Verteilte Verarbeitung für Batch-Workloads; geeignet zur Transformation, zum Schreiben von Parquet-Daten nach MinIO sowie zur Registrierung in Iceberg.
Messaging	RabbitMQ	Ereignisbasierte Entkopplung der ETL-Stufen bei geringem Betriebs-Overhead im Proof of Concept.

Fortsetzung auf der nächsten Seite

Architekturschicht	Auswahl	Kurzbegründung
Monitoring	Prometheus + Grafana	Open-Source-Standardstack zur Sammlung und Visualisierung von Metriken.
API-Schicht	FastAPI	Asynchrones, performantes Framework, gut integrierbar in Python-basierte Microservices.
Metadaten / Control Plane	PostgreSQL	Zuverlässige relationale Datenhaltung für Status-Tracking und Control-Plane-Daten; einfach im Docker-Setup betreibbar.
Security / Auth	Keycloak	OAuth2-/OIDC-konforme Authentifizierung, rollenbasiert, Open Source und hybridfähig.

4.3 Auswahl und Begründung

Auf Grundlage der definierten Bewertungskriterien sowie der vergleichenden Analyse der betrachteten Technologien lassen sich fünf zentrale architektonische Strukturprinzipien ableiten. Diese beruhen auf einer klaren funktionalen Trennung zwischen Datenhaltung (Data Plane), Abfrage (Query Plane), Verarbeitung (Processing Plane), Steuerung (Control Plane) sowie einer übergreifenden Governance-Struktur. Vgl. [ILS2021], S. 87-104. Durch diese Aufteilung entsteht eine modulare und cloudneutrale Lakehouse-Architektur, die sowohl lokal als auch in hybriden Umgebungen eingesetzt werden kann und zugleich eine langfristige Weiterentwicklung ohne Abhängigkeit von einzelnen Anbietern ermöglicht. Die zugrunde liegenden Strukturprinzipien wurden während des Projektes gemeinsam und in Absprache mit den beteiligten Experten der KWS definiert. Im Folgenden werden fünf Strukturprinzipien näher erläutert und hinsichtlich ihrer jeweiligen Aufgaben, Komponenten und Abgrenzungen beschrieben.

Die Data Plane besteht aus Apache Iceberg als Table-Format und einem S3-kompatiblen Object Storage. Iceberg übernimmt dabei zentrale Lakehouse-Funktionen wie ACID-Transaktionen, Schema-Evolution und Snapshot-basierte Versionierung. Dadurch können analytische Daten im Data-Lake-Kontext konsistent, nachvollziehbar und langfristig wartbar gespeichert werden. Als Object Storage wird MinIO eingesetzt, da es eine S3-kompatible Schnittstelle bietet und sich effizient in containerisierten On-Prem-Umgebungen betreiben lässt. So entsteht eine cloudneutrale Speicherlösung ohne Abhängigkeit von proprietären Cloud-Diensten. Im Vergleich zu klassischen Data-Warehouse-Ansätzen ermöglicht diese Kombination eine höhere Flexibilität bei Datenformaten und Engines sowie eine bessere Interoperabilität mit Analysewerkzeugen wie Trino oder Spark. Für KWS bedeutet dieser Aufbau eine cloudneutrale, erweiterbare und zukunftssichere Grundlage für analytische Datenhaltung, die hybride Szenarien unterstützt und eine schrittweise Weiterentwicklung der Architektur ermöglicht.

In der Query Plane wird Trino als zentrale SQL-Engine eingesetzt. Trino ermöglicht performante, interaktive Abfragen und unterstützt föderierte Analysen über unterschiedliche

Datenquellen hinweg. Dazu zählen insbesondere Iceberg-Tabellen im Object Storage sowie relationale Systeme wie PostgreSQL. Ein zentrales Architekturprinzip ist die klare Trennung zwischen Storage und Compute. Die Daten werden unabhängig von der Abfrageverarbeitung im Object Storage gespeichert, während Trino als skalierbare Compute-Schicht horizontal erweitert werden kann. Dadurch lassen sich sowohl analytische Ad-hoc-Abfragen als auch parallele Nutzerzugriffe effizient verarbeiten, ohne die Datenhaltung anpassen zu müssen. Für KWS bedeutet dieser Ansatz eine flexible und erweiterbare Analyseschicht, die unterschiedliche Datenquellen integrieren kann und gleichzeitig eine kontrollierte, engineunabhängige Auswertung ermöglicht.

Die Verarbeitungsebene basiert auf einem Microservice-orientierten ETL-Ansatz. Die einzelnen Services (etl-inject, etl-export, etl-register) übernehmen klar definierte Aufgaben innerhalb der Pipeline und sind funktional voneinander getrennt. Dadurch lassen sich Verarbeitungsschritte unabhängig skalieren und gezielt erweitern, ohne die gesamte Pipeline verändern zu müssen. Die Struktur bleibt modular, nachvollziehbar und wartbar. Zur Orchestrierung wird RabbitMQ als zentrales Messaging-System eingesetzt. Die Kommunikation erfolgt ereignisbasiert, wodurch direkte Abhängigkeiten zwischen Services vermieden werden. Neue Verarbeitungsschritte können als zusätzliche Consumer ergänzt werden, ohne bestehende Komponenten eng zu koppeln. Für KWS bedeutet dieser Ansatz eine flexible und robuste Datenverarbeitung, die sich an veränderte Anforderungen anpassen lässt und gleichzeitig eine klare Trennung zwischen einzelnen Verarbeitungsschritten gewährleistet.

Die Control Plane übernimmt die Verwaltung von Steuerungs- und Metadaten sowie die Überwachung des Systemzustands. PostgreSQL dient dabei als zentrale, transaktionssichere Datenbasis für Prozesszustände, ETL-Status, Benchmark-Ergebnisse und Query-Metriken. Die Metadatenhaltung ist bewusst getrennt. Dabei werden Steuerungs- und Kontrollinformationen in PostgreSQL gespeichert, während tabellen- und datenbezogene Metadaten wie Schema, Partitionen oder Snapshots im Iceberg-Katalog verwaltet werden. Dadurch bleibt die Verantwortlichkeit klar abgegrenzt. Redis ergänzt diese Ebene als schneller, nicht persistenter State-Store für temporäre Zustände, Caching und Koordinationsmechanismen zwischen Services. Zur Sicherstellung von Transparenz und Nachvollziehbarkeit werden Monitoring- und Benchmarking-Komponenten integriert. Prometheus und Grafana erfassen technische Metriken, während projektspezifische Monitoring- und Benchmark-Services diese Informationen fachlich aggregieren und vergleichbar machen. So werden Performance, Stabilität und Reproduzierbarkeit der Datenverarbeitung messbar. Für KWS schafft diese Struktur eine kontrollierbare und nachvollziehbare Steuerungsebene, die sowohl den operativen Betrieb als auch analytische Vergleiche und Simulationen zuverlässig unterstützt.

Die Governance-Schicht stellt die konsistente Absicherung der Plattform sicher. Hierfür wird Keycloak als zentrale Identity- und Access-Management-Komponente eingesetzt. Keycloak unterstützt Single Sign-On sowie rollenbasierte Zugriffskontrolle auf Basis standardisierter Protokolle wie OpenID Connect und OAuth2. Dadurch können alle Services einheitlich abgesichert werden, ohne individuelle Authentifizierungsmechanismen implementieren zu müssen. Die zentrale Verwaltung von Benutzern, Rollen und Zugriffsrechten schafft transparente und nachvollziehbare Zugriffspfade. Token-basierte Authentifizierung ermöglicht zudem eine kontrollierte Weitergabe von Berechtigungen zwischen Services. Aus diesem Ansatz ergibt sich für KWS eine einheitliche und skalierbare Zugriffskontrolle, die Governance-Anforderungen wie klare Rollenmodelle, Nachvollziehbarkeit und langfristige Compliance-Unterstützung gewährleistet.

4.4 Architekturübergreifende Struktur- und Betriebsprinzipien

Neben den zentralen Architekturentscheidungen für Data Plane, Query Plane, Verarbeitungsebene sowie Control Plane und Governance wurden im Entwurf zusätzlich übergreifende Aspekte berücksichtigt, die den stabilen Betrieb, die Erweiterbarkeit und die langfristige Reproduzierbarkeit der Plattform sicherstellen. Diese betreffen nicht einzelne Komponenten isoliert, sondern wirken systemweit und bilden die Grundlage für ein technisch konsistentes Gesamtsystem. Im Fokus stehen dabei insbesondere Deployment, Service-Kommunikation, Automatisierung, Nachvollziehbarkeit von Prozessen sowie die nutzerorientierte Interaktion mit der Plattform.

Die Architektur sieht einen containerisierten Betrieb der einzelnen Komponenten vor. Jeder Service läuft in einem eigenen Docker-Container und wird über Docker Compose orchestriert. Dadurch kann die gesamte Systemlandschaft reproduzierbar bereitgestellt werden, unabhängig von der zugrundeliegenden Infrastruktur. Die Containerisierung sorgt zudem für eine klare Trennung von Verantwortlichkeiten, erleichtert Updates einzelner Komponenten und unterstützt die portable Ausführung sowohl in On-Premises- als auch in Cloud-Umgebungen.

Die Kommunikation zwischen den Services erfolgt über klar definierte Netzwerke. Funktional zusammengehörige Komponenten sind logisch gruppiert, wodurch eine strukturierte und nachvollziehbare Service-Interaktion entsteht. Diese Segmentierung erhöht die Transparenz der Architektur und schafft eine solide Grundlage für zukünftige Erweiterungen, etwa durch zusätzliche Sicherheitsmechanismen oder gezielte Zugriffsbeschränkungen. Alle Services stellen standardisierte Health-Endpunkte bereit, über die ihr Betriebszustand überprüft werden kann. Diese Schnittstellen ermöglichen eine kontinuierliche Überwachung des Systems und können in Monitoring- und Observability-Werkzeuge integriert werden. Ausfälle oder Störungen einzelner Komponenten lassen sich dadurch frühzeitig erkennen und systematisch analysieren, ohne dass tief in die jeweilige Implementierung eingegriffen werden muss.

Wiederkehrende Prozesse werden weitgehend automatisiert, um manuelle Eingriffe zu minimieren und eine konsistente Ausführung sicherzustellen. Dazu zählen beispielsweise das Nachladen benötigter Bibliotheken für Query- oder Processing-Komponenten sowie automatisierte Ingestion- und Exportabläufe innerhalb der ETL-Pipeline. Durch diese Automatisierung wird sichergestellt, dass wiederholbare Abläufe kontrolliert und fehlerarm durchgeführt werden.

Die ETL-Prozesse sind idempotent ausgelegt. Das bedeutet, dass eine mehrfache Ausführung derselben Verarbeitungsschritte keine inkonsistenten Zustände oder Daten-Duplikate erzeugt. Diese Eigenschaft ist insbesondere für robuste Datenpipelines entscheidend, da sie Wiederholbarkeit, Fehlertoleranz und kontrollierte Recovery-Szenarien unterstützt.

Große Tabellen werden bereits beim Export logisch partitioniert. Die Partitionierung orientiert sich an fachlich oder zeitlich sinnvollen Schlüsseln und trägt dazu bei, die Effizienz der Weiterverarbeitung in der Lakehouse-Schicht zu verbessern. Auf diese Weise werden unnötige Datenzugriffe reduziert und sowohl Scan-Effizienz als auch Abfrageperformance langfristig unterstützt.

Der Fortschritt von ETL- und Verarbeitungsprozessen wird systematisch dokumentiert. Statusinformationen und Verarbeitungsschritte werden in dedizierten Tabellen erfasst,

wodurch eine transparente Nachverfolgbarkeit einzelner Verarbeitungseinheiten entsteht. Dieses strukturierte Status-Tracking kann die Analyse von Laufzeiten, Teilprozessen und möglichen Fehlerquellen erleichtern.

Zur Interaktion mit der Plattform wird eine webbasierte Benutzeroberfläche bereitgestellt. Diese dient nicht nur der Visualisierung von Monitoring- und Benchmark-Ergebnissen, sondern ermöglicht auch die interaktive Abfrage von Daten, das Durchsuchen von Tabellenstrukturen sowie die Steuerung von Simulationen. Dadurch entsteht eine zentrale Schnittstelle, die technische und fachliche Perspektiven der Plattform zusammenführt.

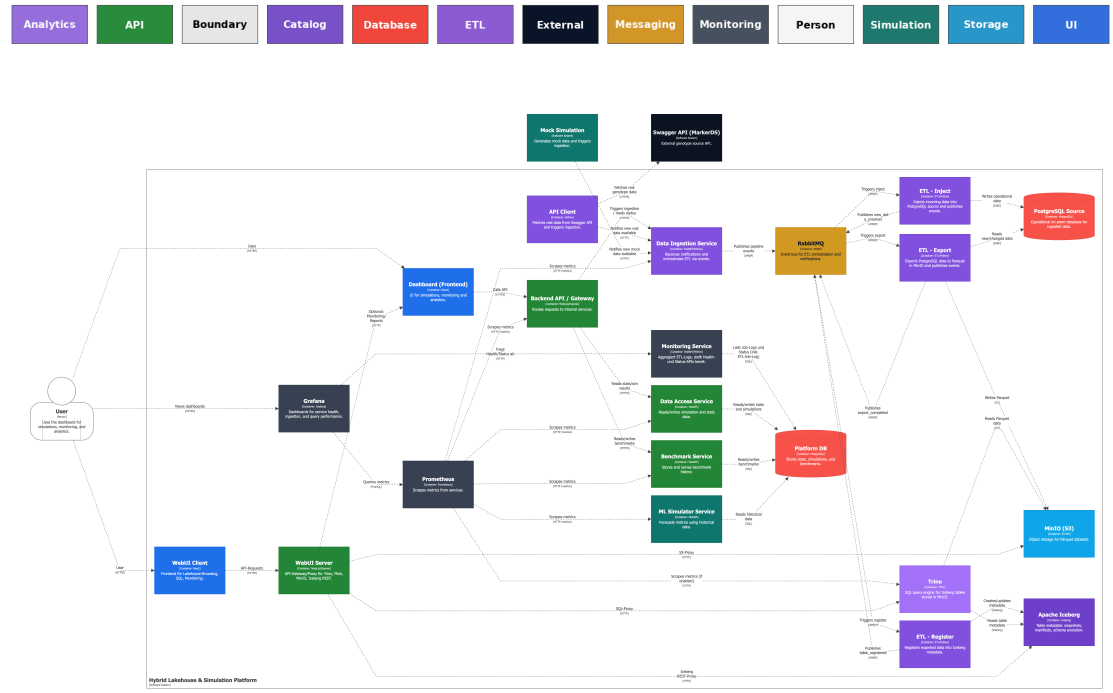
Ein wesentliches Element der Architektur ist die Integration eines Kosten- und Performancemodells innerhalb der Statistik- und Simulationskomponenten. Auf dieser Grundlage können unterschiedliche Architekturszenarien wie On-Premises-, Hybrid- oder Cloud-Varianten quantitativ verglichen werden. Die Simulation greift dabei auf reale Benchmark-Daten zurück und ermöglicht eine datenbasierte Bewertung architektonischer Entscheidungen über längere Zeiträume hinweg.

Darüber hinaus berücksichtigt die Architektur spezifische Anforderungen der genetischen Datenverarbeitung. Innerhalb der Statistik- und Simulationsschicht existieren spezialisierte Verarbeitungspipelines für Genotyp-Daten sowie für ML gestützte Analysen. Diese fachlichen Komponenten sind bewusst von der generischen Lakehouse-Infrastruktur getrennt, um die Kernarchitektur schlank zu halten und gleichzeitig eine klare Erweiterbarkeit für zukünftige Analysemodelle zu gewährleisten.

4.5 Architekturentwurf

Der Architekturentwurf soll das Zielbild der im Projekt entwickelten hybriden Lakehouseplattform und des Simulationsmodells beschreiben. Er strukturiert die funktionalen Ebenen, zeigt das Zusammenspiel der zentralen Komponenten und trennt klar zwischen dem fachlichen Datenfluss und technischen Steuerfluss. Ein Architekturdiagramm dient dabei als konzeptionelle Referenz für die spätere Umsetzung und macht sowohl den Weg der Daten als auch die Mechanismen zur Steuerung, Überwachung und Simulation transparent.

Einen Gesamtüberblick über die beschriebenen Ebenen und deren Zusammenspiel ist aus dem Architekturdiagramm in Abbildung 4.1 zu entnehmen. Abgebildet sind die geplanten Systemcontainer sowie deren Kommunikationsbeziehungen im Rahmen der Zielarchitektur, gemäß dem C4-Modell nach Simon Braun, Entwickler des Modells.



C2 - Container Diagram: Platform + Lakehouse + Observability

Abbildung 4.1: C4-Container-Diagramm der konzipierten hybriden Lakehouse- und Simulationsplattform (Level 2).
Quelle: Eigene Darstellung (DSL)

Die Plattform ist als containerisierte, serviceorientierte und ereignisgetriebene Architektur konzipiert. Sie besteht aus mehreren logisch getrennten Ebenen. Auf der Interaktions- und Präsentationsebene greifen Nutzer über eine Web UI beziehungsweise ein Dashboard auf die Plattform zu. Darunter liegt eine API- und Steuerungsebene, über die Datenaufnahme, Abfragen und Simulationen kontrolliert angestoßen werden. Die Verarbeitungsebene umfasst die ETL-Microservices sowie RabbitMQ zur ereignisbasierten Koordination. Die Data Plane bildet die persistente Datenbasis mit PostgreSQL als relationales Quellsystem sowie einem S3-kompatiblen Object Storage, in dem analytische Daten über Apache Iceberg verwaltet werden. Die Query Plane wird durch Trino realisiert, das interaktive SQL-Abfragen und föderierte Analysen ermöglicht. Ergänzt wird die Architektur durch eine Control-, Monitoring- und Simulationsebene mit Benchmark-, Monitoring- sowie Simulations- und ML-Services.

Externe Datenquellen, etwa eine Swagger API für genetische Daten, sowie Mock-Simulationen sind klar vom Kernsystem getrennt und werden ausschließlich über definierte Schnittstellen integriert. Alle Komponenten laufen in separaten Docker-Containern und werden über Docker Compose orchestriert, was eine reproduzierbare Bereitstellung und hybride Betriebsmodelle unterstützt.

Der fachliche Datenfluss beginnt mit realen oder synthetisch erzeugten genetischen Daten. Diese werden über einen Data-Ingestion-Service aufgenommen und ereignisbasiert zur weiteren Verarbeitung angestoßen. Innerhalb der ETL-Pipeline persistiert etl-inject die Rohdaten im relationalen Quellsystem. etl-export extrahiert neue oder geänderte Datensätze, transformiert sie und speichert sie partitioniert in spaltenbasiertem Format im Object Storage. Anschließend registriert etl-register die erzeugten Daten als Iceberg-Tabellen in

der Lakehouse-Schicht. Die ETL-Prozesse sind idempotent ausgelegt, sodass wiederholte Ausführungen keine inkonsistenten Zustände oder Duplikate verursachen. Die Partitionierung großer Tabellen verbessert dabei die Scan-Effizienz und unterstützt die langfristige Skalierbarkeit. Trino greift auf die Iceberg-Tabellen zu und ermöglicht interaktive Analysen sowie föderierte Abfragen über zusätzliche Datenquellen.

Vom Datenfluss getrennt ist der Steuerfluss, der die Koordination und Überwachung der Plattform regelt. RabbitMQ fungiert als zentrales Messaging-Backbone und steuert den Ablauf der Pipeline über Ereignisse. Status- und Metadaten, etwa Fortschrittsinformationen einzelner Verarbeitungseinheiten, werden in PostgreSQL gespeichert, wodurch eine detaillierte Nachvollziehbarkeit der Prozesse gewährleistet wird. Technische Metriken werden über Prometheus erfasst und mit Grafana visualisiert. Ergänzend aggregieren projektspezifische Monitoring- und Benchmark-Services fachlich interpretierbare Kennzahlen über ETL-, Query- und Simulationsprozesse hinweg. Ein integriertes Kostenmodell ermöglicht die quantitative Simulation und den Vergleich von On-Premises-, Hybrid- und Cloud-Szenarien und bildet einen zentralen Bestandteil der Plattform.

Sicherheits- und Governance-Aspekte werden durch eine zentrale Authentifizierungs- und Autorisierungskomponente auf Basis von Keycloak adressiert. Diese unterstützt Single Sign-On und rollenbasierte Zugriffskontrolle. Eine vollständige IAM-Integration ist im Proof of Concept nicht umgesetzt, kann jedoch ohne grundlegende Architekturänderung ergänzt werden. Alle Services stellen Health-Endpunkte bereit, um den Betriebszustand systematisch zu überwachen. Die Architektur ist modular angelegt, sodass zusätzliche Datenquellen, ETL-Services oder alternative Analyse-Engines ergänzt werden können, ohne bestehende Komponenten grundlegend zu verändern.

Bewusst nicht Bestandteil des Proof of Concept sind unter anderem eine produktive Hochverfügbarkeits- oder Disaster-Recovery-Strategie, ein umfassendes Data-Quality- oder Data-Catalog-System sowie ein vollständiges Lifecycle-Management. Diese Abgrenzungen dienen der klaren Fokussierung auf die technische Machbarkeit, Vergleichbarkeit und Erweiterbarkeit eines hybriden Lakehouse- und Simulationsansatzes im Kontext des Projekts. Die Architektur stellt bewusst ein strukturiertes Zielbild dar, das im Rahmen des Proof of Concepts schrittweise umgesetzt und validiert wurde.

5 Umsetzung (Implementierung)

Ziel dieses Kapitels ist der Nachweis der technischen Machbarkeit der im vorangegangenen Kapitel prototypischen Zielarchitektur. Es wird im Folgenden die prototypische Implementierung der hybriden Speicher- und Zugriffsschicht im Kontext eines Datalakehouse–Ansatzes beschrieben. Dabei liegt der Schwerpunkt auf der technischen Implementierung sowie auf der Integration der Systemkomponenten innerhalb einer realitätsnahen jedoch vom Live-System der KWS getrennten Prototypumgebung sowie der praktischen Validierung im Rahmen eines Proof of Concept (PoC).

Die praktische Umsetzung gliedert sich in zwei eng miteinander verbundene Systembereiche. Zum einen wurde eine modular strukturierte Hybrid-Data-Sync-Plattform entwickelt, die für die automatisierte Verarbeitung und Überführung der Daten in das Lakehouse zuständig ist. Zum anderen wurde eine Stat-Engine implementiert, die genetische Datensätze verarbeitet, Kostenmodelle für Cloud- und On-Premise-Szenarien simuliert, ML-basierte Auswertungen durchführt und die Ergebnisse in Echtzeit visualisiert. Beide Bereiche sind als containerisierte Microservices realisiert und werden über Docker Compose gemeinsam betrieben und koordiniert.

Im Zuge der prototypischen Umsetzung wurden verschiedene zentrale Technologien integriert. PostgreSQL dient dabei der Verwaltung von Statusinformationen und Metadaten. MinIO wird als S3-kompatibler Speicher für den Data Lake eingesetzt. Apache Iceberg übernimmt die Rolle des tabellarischen Formats innerhalb des Lakehouse-Ansatzes, während Trino und Apache Spark für die analytische SQL-Verarbeitung zuständig sind. Die Kommunikation zwischen einzelnen Diensten erfolgt über RabbitMQ als Messaging-Komponente. Ergänzend wurden Backend-Services auf Basis von FastAPI und Node.js entwickelt sowie eine React-basierte Weboberfläche zur interaktiven Analyse und Darstellung der Ergebnisse umgesetzt. Für das Monitoring und die Systembeobachtung kommen Prometheus und Grafana zum Einsatz.

Während der prototypischen Implementierung standen nicht nur funktionale Aspekte im Fokus, sondern auch technische Herausforderungen, die sich im praktischen Betrieb des Prototyps gezeigt haben. Dazu zählen insbesondere Fragestellungen zur Datenkonsistenz und Idempotenz, die Verarbeitung größerer Datenmengen durch Partitionierung und Parquet-Export, die serviceübergreifende Kommunikation innerhalb der Docker-Netzwerkstruktur sowie das Management von Konfigurationsparametern über Umgebungsvariablen. Darüber hinaus wurde eine strukturierte Fehlerbehandlung implementiert, die auf Statusmechanismen und getrennten Fehlerverzeichnissen basiert.

Das Kapitel stellt damit nicht nur die technische Umsetzung der konzipierten Architektur dar, sondern beschreibt auch die konkreten Integrationsschritte, Abhängigkeiten zwischen den Komponenten sowie notwendige Anpassungen und Optimierungen. Auf dieser Basis wird nachvollziehbar gezeigt, dass die entwickelte hybride Speicher- und Zugriffsschicht

innerhalb einer prototypischen Umgebung unter realitätsnahen Bedingungen funktionsfähig und grundsätzlich skalierbar umgesetzt werden kann.

5.1 Technischer Rahmen

Die technische Betriebsumgebung des Prototyps basiert auf einer vollständig containerisierten Infrastruktur, die lokal auf einem macOS-basierten Entwicklerrechner betrieben wird. Sämtliche Komponenten werden über Docker und Docker Compose (Version 4.63) gestartet und verwaltet. Durch die Containerisierung werden alle Dienste isoliert ausgeführt, wodurch Abhängigkeitskonflikte vermieden und reproduzierbare Testbedingungen geschaffen werden. Eine vereinfachte Übersicht der containerbasierten Systemlandschaft ist in der nachstehenden Abbildung 5.1 visualisiert.

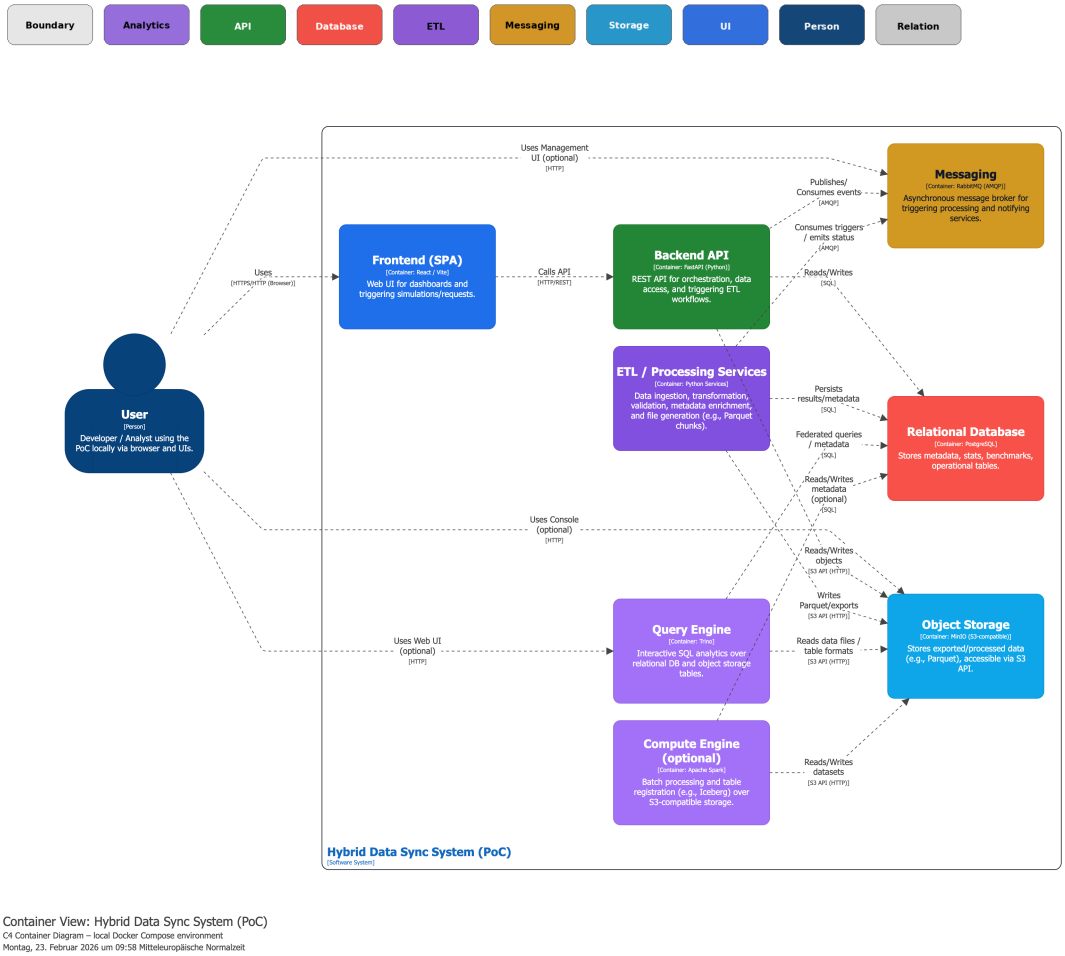


Abbildung 5.1: C4-Architekturübersicht (Level 2) des Hybrid Data-Sync-Systems (PoC).
Quelle: Eigene Darstellung (DSL)

Zu erkennen sind die zentralen Systemcontainer sowie deren Kommunikationsbeziehungen. Jede funktionale Einheit des Systems läuft in einem eigenen Container. Hierzu zählen unter anderem ETL-Services, Backend-Komponenten, Analyse-Engines, Messaging-Systeme sowie Datenbank- und Storage-Dienste. Auch zustandsbehaftete Systeme wie PostgreSQL,

MinIO oder RabbitMQ sind vollständig in die Containerstruktur integriert. Die Service-Kommunikation erfolgt innerhalb definierter Docker-Bridge-Netzwerke. Für unterschiedliche Systembereiche werden getrennte Netzwerke verwendet, um Kommunikationspfade kontrolliert zu gestalten. Externe Zugriffe erfolgen ausschließlich über explizit freigegebene Ports. Persistente Docker-Volumes stellen sicher, dass Daten von PostgreSQL, MinIO und weiteren zustandsbehafteten Diensten auch nach einem Neustart der Container erhalten bleiben. Dies ermöglicht wiederholbare End-to-End-Tests und konsistente Validierungsszenarien. Für die Kommunikation zwischen den Services werden gängige Standardprotokolle eingesetzt. HTTP/REST wird für die API-Kommunikation zwischen Frontend und Backend verwendet. AMQP dient der asynchronen Nachrichtenübertragung über RabbitMQ. SQL wird für relationale Datenbankzugriffe sowie für analytische Abfragen über Trino eingesetzt. Der Zugriff auf MinIO erfolgt über S3-kompatible Schnittstellen, die in Spark und Trino über entsprechende S3-Connectoren eingebunden sind. Die gesamte Infrastruktur wird ausschließlich lokal betrieben. Es existieren keine Hochverfügbarkeitsmechanismen, keine Cluster-Installationen und keine Multi-Node-Setups. Der technische Rahmen ist auf eine stabile, isolierte Testumgebung ausgelegt und dient nicht der Simulation eines produktiven Betriebs.

5.2 Systemkomponenten des Prototyps

Der Prototyp setzt sich aus mehreren eigenständigen Systemkomponenten zusammen, die jeweils eine klar definierte Aufgabe übernehmen. Durch diese modulare Aufteilung entsteht eine strukturierte Architektur, in der Daten schrittweise geladen, verarbeitet, persistiert, analysiert und visualisiert werden. Die Abbildung 5.2 zeigt die containerbasierte Gesamtarchitektur des PoC in Form eines C4-Container-Diagramms.

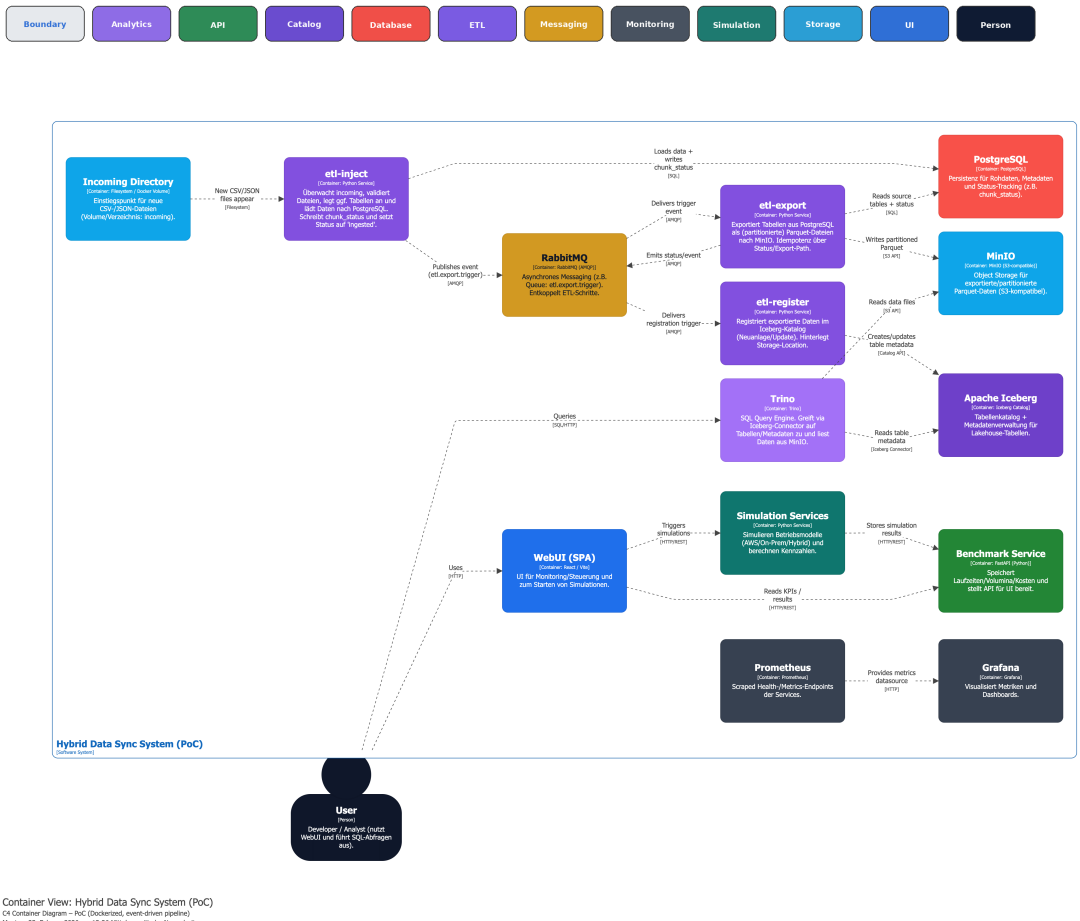


Abbildung 5.2: C4-Container-Diagramm (Level 2) des hybriden Data-Sync-Systems (PoC).
Quelle: Eigene Darstellung (DSL)

Dargestellt sind die zentralen Systemkomponenten sowie deren Kommunikationsbeziehungen. Besonders hervorgehoben werden die ETL-Services, das Messaging-System (RabbitMQ), die operative Datenhaltung (PostgreSQL), die Lakehouse-Schicht (MinIO, Iceberg, Trino) sowie die Simulations- und Monitoring-Komponenten. Die Darstellung verdeutlicht die klare funktionale Trennung der Schichten sowie die lose Kopplung der einzelnen Container. Den Einstieg bildet der Data Ingestion Service. Er überwacht ein definiertes Verzeichnis („incoming“) und erkennt neu abgelegte CSV- oder JSON-Dateien. Nach dem Einlesen werden die Daten in PostgreSQL gespeichert. Falls notwendig, erzeugt der Service automatisch passende Tabellenstrukturen. Zusätzlich wird der Verarbeitungsstatus in einer Status-Tabelle dokumentiert. Nach erfolgreicher Aufnahme sendet der Service eine Nachricht an RabbitMQ, um die weitere Verarbeitung anzustoßen. Die wichtigsten Schnittstellen sind damit das lokale Dateisystem, PostgreSQL (SQL) sowie RabbitMQ (AMQP).

Die Weiterverarbeitung erfolgt über mehrere spezialisierte ETL-Services. Der Service etl-inject initialisiert den Verarbeitungsvorgang. etl-export übernimmt den Export der Daten aus PostgreSQL in das Parquet-Format und speichert diese im Object Storage MinIO. Dabei können große Datenmengen partitioniert werden, um spätere Abfragen zu erleichtern. etl-register sorgt dafür, dass neu exportierte Daten im Iceberg-Katalog registriert werden und somit für analytische Abfragen zur Verfügung stehen. Die Koordination zwischen diesen Services erfolgt über RabbitMQ, wodurch die Verarbeitungsschritte voneinander entkoppelt

sind. RabbitMQ fungiert als Messaging-Komponente des Systems. Anstelle direkter Service-Aufrufe werden Nachrichten über Queues ausgetauscht. Dadurch bleiben die einzelnen Dienste unabhängig voneinander start- und wartbar. Im Rahmen des Prototyps wird auf komplexe Persistenz- oder Hochverfügbarkeitsmechanismen verzichtet.

Für analytische Abfragen wird Trino eingesetzt. Trino greift über den Iceberg-Connector auf die im Lakehouse registrierten Tabellen zu, die physisch im Object Storage MinIO gespeichert sind. Dadurch können strukturierte SQL-Abfragen auf großen Datenbeständen ausgeführt werden, ohne dass die Daten in eine klassische Data-Warehouse-Struktur überführt werden müssen. PostgreSQL übernimmt mehrere Aufgaben innerhalb des Systems. Zum einen dient es als Zielsystem für die initial eingelesenen Rohdaten. Zum anderen speichert es Metadaten, Statusinformationen sowie Verwaltungsdaten der ETL-Prozesse.

Zur Steuerung der Verarbeitung wird für jeden Daten-Chunk ein Eintrag in der Tabelle `chunk_status` geführt. Jeder Chunk wird über eine eindeutige `chunk_id` identifiziert, anhand derer sein aktueller Zustand nachvollzogen werden kann. Der Status beschreibt, in welchem Verarbeitungsschritt sich der Chunk befindet (z. B. `ingested`, `exported` oder `registered`). Zusätzlich werden technische Metadaten wie `row_count` sowie der zugehörige `export_path` gespeichert. Der Verarbeitungsfortschritt bleibt dadurch pro Chunk eindeutig rekonstruierbar. Unvollständig verarbeitete Chunks können gezielt identifiziert und erneut angestoßen werden, ohne dass doppelte Einträge entstehen. Die Statusaktualisierung erfolgt idempotent über eine UPSERT-Operation auf Basis von `INSERT ... ON CONFLICT`. Existiert für eine `chunk_id` bereits ein Eintrag, wird dieser aktualisiert; andernfalls wird ein neuer Datensatz angelegt. Damit wird verhindert, dass wiederholte Verarbeitungsschritte zu inkonsistenten Zuständen oder Duplikaten führen. Listing 5.1 zeigt einen Auszug des Python-Skripts zur Aktualisierung des Chunk-Status.

```
#-- PostgreSQL: Tabelle fr Status-Tracking (chunk_status)
CREATE TABLE IF NOT EXISTS chunk_status (
    chunk_id      TEXT PRIMARY KEY,          -- Eindeutige ID pro Datei/Chunk
    status        TEXT NOT NULL,             -- z.B. 'ingested', 'exported', 'registered',
    'error'
    row_count     BIGINT,
    export_path   TEXT,
    created_at    TIMESTAMP DEFAULT now(),
    last_updated  TIMESTAMP DEFAULT now()
);

CREATE INDEX IF NOT EXISTS idx_chunk_status_status ON chunk_status(status);

# Python: Idempotente Status-Update-Funktion (UPSERT)
def upsert_chunk_status(conn, chunk_id, status, row_count=None, export_path=None):
    """
    Schreibt oder aktualisiert den Status eines Chunks (idempotent, traceable).
    """
    with conn.cursor() as cur:
        cur.execute("""
            INSERT INTO chunk_status (chunk_id, status, row_count, export_path,
    last_updated)
            VALUES (%s, %s, %s, %s, now())
            ON CONFLICT (chunk_id) DO UPDATE
            SET status = EXCLUDED.status,
                row_count = COALESCE(EXCLUDED.row_count, chunk_status.row_count),
                export_path = COALESCE(EXCLUDED.export_path, chunk_status.export_path),
```

```
                last_updated = now()
        """, (chunk_id, status, row_count, export_path))
    conn.commit()
```

Listing 5.1: Idempotente UPSERT-Funktion zur Aktualisierung des Chunk-Status
Quelle: Eigene Implementierung (VS Code, Python).

Die Datenbank läuft in einem eigenständigen Container und verwendet persistente Volumes, sodass der Datenbestand über Container-Neustarts hinweg erhalten bleibt. MinIO bildet den Object Storage des Prototyps. Hier werden die im Parquet-Format exportierten Daten abgelegt. Der Zugriff erfolgt über S3-kompatible Schnittstellen, die in Spark und Trino über entsprechende Connectoren eingebunden sind. In Kombination mit Apache Iceberg entsteht so die technische Grundlage des Lakehouse-Ansatzes.

Zur Systemüberwachung werden Prometheus und Grafana verwendet. Prometheus sammelt Metriken über definierte Endpunkte einzelner Services. Grafana stellt diese Informationen in Dashboards dar, sodass Laufzeiten, Statuswerte und Systemauslastungen überprüfbar sind.

Der Benchmark-Service sammelt Leistungskennzahlen des Systems, beispielsweise Verarbeitungszeiten oder simulierte Kostenwerte. Zusätzlich werden hier ML-Modelle eingesetzt, um Prognosen auf Basis historischer Daten zu berechnen. Die Ergebnisse werden über eine REST-Schnittstelle bereitgestellt und in der Weboberfläche angezeigt. Ergänzend existieren Simulation-Services, die verschiedene Architekturszenarien abbilden, etwa AWS-, On-Premise- oder hybride Modelle. Diese Services berechnen Kostenentwicklungen, Laufzeiten oder Skalierungseffekte und übermitteln ihre Ergebnisse an den Benchmark-Service.

Redis wird als In-Memory-Komponente eingesetzt, um temporäre Daten oder Zwischenergebnisse zwischenzuspeichern. Eine dauerhafte Persistenz ist hier nicht vorgesehen. Die WebUI stellt die Benutzerschnittstelle des Systems dar. Sie wurde mit React umgesetzt und kommuniziert über REST-APIs sowie WebSocket-Verbindungen mit den Backend-Services. Über die Oberfläche können SQL-Abfragen ausgeführt, Monitoring-Daten eingesehen sowie Simulationen gestartet werden. Ergebnisse werden tabellarisch und grafisch dargestellt.

Für die Echtzeit-Aktualisierung wird eine Event-Weiterleitung zwischen RabbitMQ und der WebUI verwendet. Ein Backend-Service konsumiert relevante Nachrichten aus der Queue und sendet sie über eine WebSocket-Verbindung an das Frontend weiter. Statusänderungen oder neue Metriken erscheinen dadurch direkt in der Oberfläche. Ein periodisches Polling ist nicht erforderlich. Listing 5.2 zeigt einen Auszug der entsprechenden Implementierung.

```
// --- Backend: RabbitMQ-Event  WebSocket-Update (Node.js/Express + socket.io) ---
import { Server } from 'socket.io';
import amqp from 'amqplib';
const io = new Server(server, { /* ...CORS-Config... */ });

// RabbitMQ-Listener
amqp.connect('amqp://localhost').then(conn =>
  conn.createChannel().then(ch =>
    ch.consume('myqueue', msg => {
      const data = JSON.parse(msg.content.toString());
      io.emit('update', { timestamp: Date.now(), download_time_sec:
    data.download_time_sec });
      ch.ack(msg);
    })
  )
```

```
);

// --- Frontend: Minimaler WebSocket-Listener (z.B. in chart.js) ---
const socket = io('http://localhost:8080');
socket.on('update', payload => {
  // UI/Chart aktualisieren
  data.labels.push(payload.timestamp);
  data.datasets[0].data.push(payload.download_time_sec);
  chart.update();
});
```

Listing 5.2: Ereignisbasierte Weiterleitung von RabbitMQ-Nachrichten an die WebUI über WebSockets.
Quelle: Eigene Implementierung (VS Code, Python).

Insgesamt ergibt sich daraus eine klar strukturierte und containerisierte Microservice-Architektur. Jede Komponente erfüllt eine spezifische Rolle, und die Interaktion erfolgt über definierte Schnittstellen. Dadurch bleibt das System nachvollziehbar aufgebaut und technisch erweiterbar, ohne unnötige Komplexität im Prototyp zu erzeugen.

In Abschnitt 5.3 wird der Ablauf der Datenverarbeitung im Prototyp schrittweise beschrieben von der Bereitstellung der Rohdaten bis zur analytischen Nutzung im Lakehouse sowie zur Weiterverarbeitung in den Simulationsdiensten. Der Fokus liegt auf der technischen Abfolge der einzelnen Verarbeitungsschritte und deren Zusammenspiel. Die im Anhang abgebildete Tabelle fasst die zentralen Komponenten des Prototyps sowie deren technische Rollen und Schnittstellen zusammen Anhang A.2.

5.3 Datenfluss und Prozessabläufe

Der Prozess beginnt mit der Ablage von CSV- oder JSON-Dateien im Verzeichnis „incoming“. Dieses Verzeichnis dient als Einstiegspunkt für neue Daten. Der Service etl-inject überwacht diesen Bereich kontinuierlich. Wird eine neue Datei erkannt, prüft der Service zunächst, ob in PostgreSQL bereits eine passende Tabelle existiert. Falls nicht, wird die Tabellenstruktur dynamisch erzeugt. Anschließend werden die Daten eingelesen und in die Datenbank übertragen. Für jede verarbeitete Datei wird ein Eintrag in der Tabelle chunk_status angelegt. Dort werden unter anderem Dateiname, Zeitstempel, aktueller Status sowie spätere Exportinformationen gespeichert. Der Status wird initial auf *pending* gesetzt und im weiteren Verlauf abhängig vom Verarbeitungsergebnis aktualisiert. Abbildung 5.3 visualisiert diesen Status-Lebenszyklus als Zustandsmodell: Ein Chunk startet in *pending* und wechselt bei erfolgreichem Export in *exported* oder bei Fehlern in *error*; optional kann nach erfolgreichem Export eine Archivierung als *archived* erfolgen. Die jeweils aktuellen Zustände sowie zugehörige Metadaten werden dabei in der Tabelle `chunk_status` fortlaufend dokumentiert und ermöglichen ein nachvollziehbares Monitoring des ETL-Fortschritts. Dargestellt sind die möglichen Zustandsübergänge zwischen *pending*, *exported*, *error* sowie optional *archived*. Die Zustände werden in der Tabelle `chunk_status` dokumentiert.

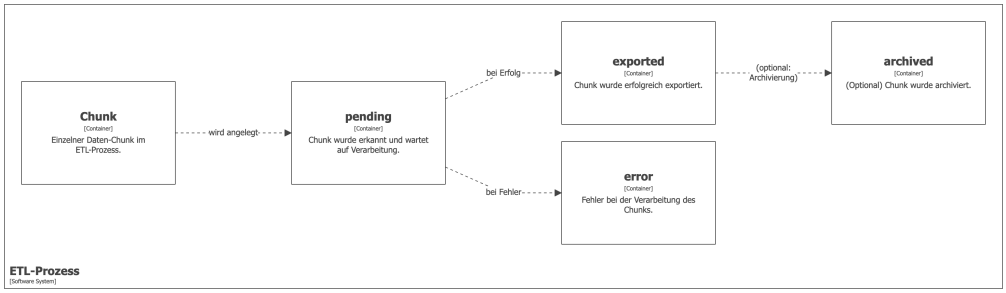


Abbildung 5.3: Status-Lebenszyklus eines Daten-Chunks innerhalb des ETL-Prozesses.
Quelle: Eigene Darstellung (DSL)

Nach erfolgreichem Einlesen einer Datei wird über eine Message-Queue ein Ereignis ausgelöst, das den nächsten Verarbeitungsschritt asynchron startet. Die Kommunikation zwischen den ETL-Komponenten erfolgt ausschließlich ereignisbasiert über persistente Queues. Dadurch entsteht eine lose Kopplung der Services, sodass keine direkte Abhängigkeit zwischen den einzelnen Verarbeitungsschritten besteht. Dieses Architekturprinzip erhöht die Skalierbarkeit und Fehlertoleranz der Pipeline.

Der vollständige Implementierungsauszug des ereignisgesteuerten Ablaufs befindet sich in Listing 5.3 dargestellt. Im Folgenden wird ersichtlich, wie etl-inject ein FILE_INGESTED-Ereignis publiziert, das von etl-export asynchron verarbeitet wird. Nach erfolgreichem Export wird ein EXPORT_DONE-Ereignis erzeugt, das wiederum von etl-register zur Registrierung der Daten im Iceberg-Katalog verarbeitet wird.

```
# --- Event-Driven ETL-Pipeline: inject  export  register

Q_EXPORT   = os.getenv("RABBITMQ_EXPORT_QUEUE", "etl.export.trigger")
Q_REGISTER = os.getenv("RABBITMQ_REGISTER_TRIGGER_QUEUE", "etl.register.trigger")

def rmq():
    return pika.BlockingConnection(
        pika.ConnectionParameters(
            host=os.getenv("RABBITMQ_HOST", "rabbitmq"),
            credentials=pika.PlainCredentials(
                os.getenv("RABBITMQ_USER", "guest"),
                os.getenv("RABBITMQ_PASSWORD", "guest"),
            ),
            virtual_host=os.getenv("RABBITMQ_VHOST", "/"),
            port=int(os.getenv("RABBITMQ_PORT", "5672")),
        )
    )

def now():
    return datetime.now(timezone.utc).isoformat(timespec="seconds").replace("+00:00", "Z")

def event(name, **data):
    return {"event": name, **data, "ts": now()}

def publish(queue, payload):
    with rmq() as c:
```

```
        ch = c.channel()
        ch.queue_declare(queue=queue, durable=True)
        ch.basic_publish(
            exchange="",
            routing_key=queue,
            body=json.dumps(payload),
            properties=pika.BasicProperties(delivery_mode=2),  # persistent
        )

def consume(queue, handler):
    c = rmq()
    ch = c.channel()
    ch.queue_declare(queue=queue, durable=True)
    ch.basic_qos(prefetch_count=1)

    def cb(ch_, method, props, body):
        try:
            handler(json.loads(body))
        finally:
            ch_.basic_ack(method.delivery_tag)

    ch.basic_consume(queue=queue, on_message_callback=cb)
    print(f"[consumer] listening on {queue}")
    ch.start_consuming()

# --- etl-inject: send FILE_INGESTED ---
def publish_file_ingested():
    e = event(
        "FILE_INGESTED",
        filename="accommodations_2026-02-28.csv",
        table="accommodations",
        chunk_id=42,
        ingest_time="2026-02-28T10:15:00Z",
    )
    publish(Q_EXPORT, e)
    print(f"[etl-inject] {e}")

# --- etl-export: receive FILE_INGESTED, send EXPORT_DONE ---
def consume_and_export():
    def handle(e):
        if e.get("event") != "FILE_INGESTED":
            return
        out = event(
            "EXPORT_DONE",
            table=e["table"],

    export_path=f"s3://lakehouse/bronze/{e['table']}/part-{e['chunk_id']:05d}.parquet",
            chunk_id=e["chunk_id"],
            export_time="2026-02-28T10:20:00Z",
        )
        publish(Q_REGISTER, out)
        print(f"[etl-export] {out}")

    consume(Q_EXPORT, handle)

# --- etl-register: receive EXPORT_DONE, emit REGISTER_DONE (logged) ---
def consume_and_register():
    def handle(e):
        if e.get("event") != "EXPORT_DONE":
```

```
            return
        done = event(
            "REGISTER_DONE",
            table=e["table"],
            trino_schema="lakehouse.bronze",
            chunk_id=e["chunk_id"],
            register_time="2026-02-28T10:21:00Z",
        )
        print(f"[etl-register] {done}")

    consume(Q_REGISTER, handle)
```

Listing 5.3: Vereinfachter Codeauszug der ereignisgesteuerten ETL-Pipeline.
Quelle: Eigene Implementierung (VS Code, Python).

Der Service etl-export verarbeitet das empfangene Ereignis und startet den Exportvorgang. Die Daten werden aus PostgreSQL ausgelesen, in das Parquet-Format konvertiert und im Object Storage MinIO gespeichert. Bei größeren Datenmengen erfolgt eine Partitionierung, beispielsweise nach Datum oder Chunk-ID. Dadurch wird die Datenstruktur im Storage übersichtlicher und spätere Abfragen können gezielter auf Teilbereiche zugreifen. Um doppelte Verarbeitung zu vermeiden, wird ein einfacher Idempotenzmechanismus eingesetzt. Vor dem Export prüft der Service anhand der Statusinformationen in PostgreSQL, ob ein Datensatz bereits exportiert wurde. Falls ein gültiger `export_path` vorhanden ist und der Status entsprechend gesetzt wurde, wird der Datensatz übersprungen. Fehlerfälle können gesondert markiert oder in ein separates Verzeichnis verschoben werden. Dadurch bleibt der Ablauf auch bei erneuter Ausführung konsistent.

Nach dem erfolgreichen Export übernimmt der Service etl-register die Registrierung im Iceberg-Katalog. Dabei wird entweder eine neue Tabelle angelegt oder eine bestehende Tabelle aktualisiert. In diesem Schritt wird insbesondere der Speicherpfad (Storage-Location) hinterlegt, unter dem die zuvor erzeugten Parquet-Dateien im Object Storage abgelegt wurden. Iceberg verwaltet anschließend die dazugehörigen Metadaten und bindet die Dateien als Teil einer Tabelle ein. Erst nach dieser Registrierung sind die Daten für die analytische Zugriffsschicht sichtbar und können beispielsweise über Trino per SQL abgefragt werden. Die Abfrage erfolgt anschließend über Trino. Trino nutzt den Iceberg-Connector, um auf die im Object Storage abgelegten Daten zuzugreifen. SQL-Abfragen können direkt gegen die registrierten Tabellen ausgeführt werden, ohne dass eine zusätzliche Datenkopie erforderlich ist.

Neben der eigentlichen Datenpipeline können über die Weboberfläche Simulationen gestartet werden. Diese betreffen beispielsweise Kostenmodelle oder Architekturszenarien. Die Simulation-Services berechnen entsprechende Kennzahlen und übermitteln die Ergebnisse an den Benchmark-Service. Dort werden Laufzeiten, Datenvolumina oder berechnete Kosten gespeichert und über eine API bereitgestellt. Parallel dazu läuft das Monitoring. Jeder Service stellt einen Health-Endpoint zur Verfügung. Prometheus ruft diese regelmäßig ab und speichert Metriken wie Laufzeiten oder Fehlerraten. Grafana visualisiert diese Informationen, sodass der Zustand des Systems nachvollziehbar bleibt.

Ein typischer Durchlauf lässt sich damit wie folgt zusammenfassen: Eine Datei wird im „`incoming`"-Verzeichnis abgelegt, `etl-inject` erkennt sie, erzeugt gegebenenfalls eine Tabelle und speichert die Daten in PostgreSQL. Ein Status-Eintrag wird erstellt und ein RabbitMQ-Event ausgelöst. `etl-export` exportiert die Daten als partitionierte Parquet-Dateien nach MinIO. Anschließend wird die Tabelle im Iceberg-Katalog registriert. Die

Daten können danach unmittelbar über Trino abgefragt werden. Währenddessen werden Status- und Monitoring-Daten kontinuierlich erfasst. Der beschriebene Ablauf zeigt, dass die Verarbeitung im Prototyp vollständig ereignisgesteuert organisiert ist. Jeder Schritt ist technisch nachvollziehbar, separat überprüfbar und über Statusinformationen dokumentiert. Damit wird die konzipierte hybride Speicher- und Zugriffsschicht nicht nur modelliert, sondern als durchgängiger Prozess umgesetzt.

5.4 Implementierung der Speicher- und Zugriffsschicht

Die Implementierung der Speicher- und Zugriffsschicht bildet den technischen Kern des Prototyps. In diesem Abschnitt wird beschrieben, wie die relationale Speicherung, der objektbasierte Lakehouse-Bereich sowie die analytische Zugriffsebene konkret miteinander verbunden wurden.

Die operative Datenspeicherung erfolgt über PostgreSQL. Für neu eingehende Dateien werden bei Bedarf dynamisch Tabellen erzeugt, etwa für unterschiedliche Datenquellen wie „genotype_data“ oder „weather_observations“. Dadurch ist es möglich, heterogene Datenformate ohne fest vorgegebenes globales Schema zu verarbeiten.

Neben den eigentlichen Rohdatentabellen existieren mehrere Metadaten- und Status-Tabellen. Eine zentrale Rolle spielt dabei die Tabelle chunk_status. Sie dokumentiert für jede Datei beziehungsweise jeden Datenabschnitt den aktuellen Verarbeitungszustand, beispielsweise „ingested“, „exported“ oder „error“. Zusätzlich werden Felder wie Zeitstempel, Dateiname und export_path gespeichert. Auf dieser Grundlage wird die Idempotenz technisch umgesetzt:

Vor einem Export wird geprüft, ob ein Datensatz bereits vollständig verarbeitet wurde. Dadurch lassen sich doppelte Verarbeitungsschritte vermeiden und der Prozess bleibt auch bei erneuter Ausführung konsistent. Im Kontext der Stat-Engine werden darüber hinaus Benchmark-Tabellen geführt, etwa genotype_stats. Diese enthalten Kennzahlen aus Simulationen, beispielsweise Datenvolumen, Laufzeiten oder berechnete Kostenwerte. Die Tabellenstruktur ist so gestaltet, dass unterschiedliche Architekturszenarien systematisch ausgewertet und miteinander verglichen werden können. Zur Unterstützung der Zugriffsschicht wird Redis als In-Memory-Komponente eingesetzt. Häufig abgefragte Statusinformationen können dort zwischengespeichert werden, wodurch die Belastung der relationalen Datenbank reduziert wird. Zusätzlich wird Redis für einfache Lock-Mechanismen genutzt. Bei parallelen ETL-Prozessen verhindert ein entsprechender Lock, dass mehrere Prozesse denselben Datenbereich gleichzeitig exportieren. Auf diese Weise wird die Konsistenz der Verarbeitung erhöht, ohne komplexe Transaktionslogik einführen zu müssen.

Der Lakehouse-Bereich wird über MinIO als S3-kompatiblen Object Storage realisiert. Exportierte Parquet-Dateien werden in einem zentralen Bucket, beispielsweise „lakehouse“, abgelegt. Innerhalb des Buckets erfolgt eine strukturierte Ablage nach Tabellenname und Partitionierungsattribut. Ein typischer Pfad kann etwa wie folgt aufgebaut sein: `lakehouse/bronze/genotype_data/genotype_data_batch-0001.parquet`

Die Partitionierung wird bereits beim Export definiert und orientiert sich an Attributen wie Datum oder Chunk-ID. Der Export in den Lakehouse-Bereich erfolgt batchweise. Die

jeweilige PostgreSQL-Tabelle wird dabei iterativ ausgelesen, in Parquet-Dateien transformiert und anschließend in das Object Storage geschrieben. Die Partitionierung erfolgt über den S3-Pfad. Nach erfolgreichem Upload wird der Exportstatus in der relationalen Tabelle aktualisiert. Dadurch kann der Exportprozess bei Bedarf erneut angestoßen werden, ohne bereits verarbeitete Daten doppelt zu übertragen. Listing 5.4 zeigt einen vereinfachten Auszug der Implementierung dieses Exportprozesses.

```
def export_table_streaming(conn, s3, table: str, export_batch_size: int, bucket: str):
    """
    Exportiert Daten aus einer PostgreSQL-Tabelle als Parquet-Batches nach MinIO/S3
    (Lakehouse-ready).
    Partitionierung erfolgt ber den S3-Pfad (z.B. bronze/genotype_data/).
    """
    exported_total = 0
    while True:
        # 1) Hole einen Batch von noch nicht exportierten Zeilen
        with conn.cursor() as cur:
            cur.execute(f"""
                SELECT * FROM {table}
                WHERE export_path IS NULL OR last_updated IS NULL
                LIMIT %s
            """, (export_batch_size,))
            rows = cur.fetchall()
            columns = [desc[0] for desc in cur.description]

        if not rows:
            break

        df = pd.DataFrame(rows, columns=columns)
        # Optional: Statusspalten nicht exportieren
        for col in ('export_path', 'last_updated'):
            if col in df.columns:
                df = df.drop(columns=[col])

        # 2) Schreibe Parquet-Datei in den Speicher
        table_pa = pa.Table.from_pandas(df, preserve_index=False)
        buf = io.BytesIO()
        pq.write_table(
            table_pa, buf,
            compression="snappy",
            write_statistics=True,
            row_group_size=500_000
        )
        buf.seek(0)

        # 3) Definiere S3-Pfad (Partitionierung nach Tabelle, optional nach Spalte)
        file_name = f"{table}_batch_{exported_total}_{exported_total + len(df) -
    1}_{datetime.utcnow().strftime('%Y%m%dT%H%M%S')}.parquet"
        object_key = f"bronze/{table}/{file_name}"

        # 4) Lade Parquet nach MinIO/S3 hoch
        s3.put_object(Bucket=bucket, Key=object_key, Body=buf.getvalue(),
    ContentType="application/octet-stream")
        print(f"[EXPORT] Parquet exportiert: s3://{bucket}/{object_key}")

        # 5) Aktualisiere Status in der Datenbank
```

```
        with conn.cursor() as cur:
            cur.execute(
                f"UPDATE {table} SET export_path = %s, last_updated = NOW() WHERE
    export_path IS NULL OR last_updated IS NULL",
                (f"s3://{bucket}/{object_key}",)
            )
            conn.commit()

        exported_total += len(df)
        if len(df) < export_batch_size:
            break

    return exported_total
```

Listing 5.4: Batchweiser Streaming-Export einer PostgreSQL-Tabelle in Parquet-Dateien nach MinIO. **Quelle:** Eigene Implementierung (VS Code, Python).

Dadurch können analytische Abfragen gezielt auf einzelne Partitionen zugreifen, ohne den gesamten Datenbestand zu lesen. Unvollständige oder fehlerhafte Exporte werden getrennt abgelegt, um die Konsistenz des Lakehouse-Bereichs nicht zu gefährden.

Für die tabellarische Verwaltung der Daten wird Apache Iceberg eingesetzt. Iceberg wird als REST-Katalog mit JDBC-Backend betrieben. Nach dem Export legt der Service etl-register eine entsprechende Tabelle im Iceberg-Katalog an oder aktualisiert sie. Dabei wird die Storage-Location des exportierten Datenpfads hinterlegt.

Die Tabellenregistrierung erfolgt automatisiert. Ordner im Object Storage werden geprüft und, sofern noch nicht im Katalog vorhanden, als Iceberg-Tabellen registriert. Existiert die Zieltabelle bereits, erfolgt keine Neuanlage. Das Schema wird aus den vorhandenen Parquet-Dateien abgeleitet. Leere oder nicht lesbare Ordner werden übersprungen. Die Tabelle wird anschließend mit festgelegten Eigenschaften (z. B. Format-Version und Kompression) im Iceberg-Katalog angelegt. Listing 5.5 zeigt den entsprechenden Implementierungsauszug.

```
def register_tables():
    tables = list_s3_folders()
    for table in tables:
        data_path = f"{WAREHOUSE_PATH}{table}/"
        table_name = f"{CATALOG}.{NAMESPACE}.{table}"
        chunk_id = table
        file_name = f"s3a://lakehouse/bronze/{table}/"
        try:
            spark.read.table(table_name).limit(1).collect()
            print(f"[INFO]  Table already exists: {table_name}")
            continue
        except Exception:
            print(f"[INFO]  Table {table_name} does not exist. Will create.")

        try:
            sample = spark.read.parquet(data_path).limit(1).collect()
        except Exception:
            print(f"[WARN]  Skipping non-parquet or unreadable folder: {table}")
            continue

        df = spark.read.parquet(data_path)
        row_count = df.count()
        if row_count == 0:
            print(f"[WARN]  Skipping empty table: {table}")
```

```
            continue

        df.createOrReplaceTempView("temp_view")
        df.writeTo(table_name) \
          .using("iceberg") \
          .tableProperty("format-version", "2") \
          .tableProperty("write.format.default", "parquet") \
          .tableProperty("write.parquet.compression-codec", "zstd") \
          .createOrReplace()

        print(f"[INFO]  Table registered: {table_name} with {row_count} rows")
        # (Optional: Metadaten-Update, Logging, Fehlerbehandlung)
```

Listing 5.5: Automatisierte Registrierung von Parquet-Ordnern als Iceberg-Tabellen via Spark.
Quelle: Eigene Implementierung (VS Code, Python)

Iceberg verwaltet anschließend die zugehörigen Metadaten und referenziert die gespeicherten Parquet-Dateien als Data Files innerhalb der Tabelle. Das Partitionierungsmuster ist im Tabellenschema definiert und wird beim Schreiben der Daten berücksichtigt, sodass Struktur und Metadaten konsistent bleiben. Die analytische Zugriffsebene wird durch Trino umgesetzt. Trino nutzt den Iceberg-Connector, um auf die im Object Storage abgelegten Tabellen zuzugreifen. Die Konfiguration erfolgt über eine Katalogdefinition, in der sowohl der Iceberg-REST-Katalog als auch das Storage-Backend referenziert sind. Neben klassischen SELECT-Abfragen sind auch föderierte Abfragen möglich. So können beispielsweise Daten aus einer Iceberg-Tabelle mit operativen Metadaten aus PostgreSQL innerhalb einer einzigen SQL-Abfrage kombiniert werden. Dadurch entsteht eine einheitliche Zugriffsschicht über unterschiedliche Speichersysteme hinweg.

Insgesamt ergibt sich daraus eine klar getrennte Speicher- und Zugriffsebene: PostgreSQL übernimmt die operative Datenhaltung und Statusverwaltung, MinIO speichert die exportierten Parquet-Dateien, Iceberg verwaltet die Tabellenmetadaten und Partitionierungsinformationen, während Trino die analytische SQL-Abfrage ermöglicht. Die einzelnen Komponenten sind über definierte Schnittstellen verbunden, ohne dass eine direkte Kopplung zwischen operativer und analytischer Speicherung erforderlich ist. Ergänzend zur technischen Implementierung wird im nächsten Abschnitt der Umgang mit Sicherheitsaspekten im Prototyp erläutert. Die konkrete technische Ausprägung dieser Architektur wird in Tabelle 5.1 anhand realer Beispielwerte aus dem Prototyp veranschaulicht.

Tabelle 5.1: Systemebenen und reale Beispielwerte aus dem PoC.
Quelle: Eigene Darstellung (Latex)

Ebene / Komponente	Technologie / Tabelle	Werte
Operative Speicherung	PostgreSQL genotype_data	genotype_id:G0835,value:GG,mock_id:7671 genotype_id:G0836,value:GG,mock_id:7671 ... (Daten aus Trino-Query: SELECT*FROMlakehouse.bronze.genotype_dataLIMIT2;)
Status/Metadaten	PostgreSQL chunk_status	chunk_id:genotype_data_20260219T092821.csv, status:exported,row_count:1000, created_at:2026-02-1909:28:21, export_path:bronze/genotype_data/genotype_data_batch_0_999_20260219T092821.parquet (Werte aus Live-PostgreSQL, z.B. chunk_status mit echten Dateinamen)

Fortsetzung auf der nächsten Seite

Ebene / Komponente	Technologie / Tabelle	Werte
ETL-Log	PostgreSQL etl_job_log	id:1, chunk_id:genotype_data_20260219T092821.csv, file_name:genotype_data_batch_0_999_20260219T092821.parquet, status:success,job_type:export, timestamp:2026-02-1909:29:02 (Werte aus Live-PostgreSQL, z. B. etl_job_log mit echten Jobs)
Benchmark-Tabellen	PostgreSQL benchmark	id:1,genotype_set_id:genotype_data,file_size_kb:12345, download_time_sec:2.3,created_at:2026-02-1909:30:11,source:mock (Werte aus Live-PostgreSQL, z. B. benchmark mit echten IDs und Zeiten)
In-Memory-Komponente	Redis (stat-engine-redis-1)	Keine Nutzdaten/Keys vorhanden. Letzter Neustart: 23.02.2026 10:49 Uhr. Version: 8.2.2 (64 Bit), Modus: standalone, Port: 6379. Module: RedisBloom (8.2.3, Konfiguration: bf-error-rate0.01, initial-size100, ...), RediSearch (8.2.5, Cluster: AUTOpartitions), RedisTimeSeries (80200), ReJSON (80201, APIs V2-V5 exportiert). Diskless Replication aktiviert. Regelmäßige RDB-Snapshots, keine Fehler im Log.
Objektspeicher	MinIO (mc, minio)	Bucket:lakehouse,Pfad:bronze/genotype_data/ Beispiel-Datei:genotype_data_batch_0_999_20260219T092821.parquet (mclsminio/lakehouse/bronze/genotype_data/)
Tabellenkatalog	Apache Iceberg / Trino	Tabellen in lakehouse.bronze: full_genetic_data,genotype_data,marker_data,result_data (SHOWTABLESINlakehouse.bronze)
Analytische Zugriffsebene	Trino + Iceberg	Trino-CLI funktioniert, Tabellen sind abfragbar: Beispiel-Query: SELECT*FROMlakehouse.bronze.genotype_dataLIMIT2; Ergebnis: "G0835","GG","7671" und "G0836","GG","7671"

5.5 Sicherheitsaspekte im Prototyp

Im Rahmen des Prototyps lag der Schwerpunkt auf der technischen Umsetzung und Integration der Systemkomponenten. Sicherheitsaspekte wurden berücksichtigt, jedoch nicht in Form eines vollständigen unternehmensweiten Sicherheitsmodells umgesetzt. Nachfolgend werden sowohl die vorhandenen Schutzmechanismen als auch bewusst nicht umgesetzte Maßnahmen beschrieben.

Eine Anbindung an ein zentrales Identity-Management-System wie beispielsweise Keycloak wurde im aktuellen Stand nicht implementiert. Es existiert keine zentrale Authentifizierungsinstanz für Benutzer oder Services. Auch ein Single-Sign-On-Mechanismus wurde nicht eingerichtet. Die Authentifizierung erfolgt im lokalen Setup nicht über einen externen Identity-Provider. Eine durchgängige tokenbasierte Absicherung der APIs wurde ebenfalls nicht umgesetzt. Zwar unterstützen die eingesetzten Backend-Technologien grundsätzlich Verfahren wie JWT oder OAuth2, diese wurden im Prototyp jedoch nicht aktiviert. Die API-Endpunkte sind innerhalb der lokalen Umgebung zugänglich, ohne dass eine verpflichtende Authentifizierung oder Autorisierung erfolgt. Ein rollenbasiertes Zugriffskonzept ist nicht implementiert; alle Services und Nutzer verfügen über identische Berechtigungen.

Auf Infrastrukturebene wurden dennoch grundlegende Schutzmechanismen berücksichtigt. Die Services sind in getrennten Docker-Netzwerken organisiert. Eine direkte Kommunikation zwischen Containern ist nur innerhalb des jeweiligen Netzwerks möglich. Externe Zugriffe erfolgen ausschließlich über explizit freigegebene Ports. Diese Struktur verhindert ungewollte direkte Verbindungen zwischen isolierten Systembereichen. Zentrale Komponenten wie PostgreSQL oder Redis sind nicht direkt über das Hostnetzwerk erreichbar.

Sie sind ausschließlich innerhalb der Docker-Netzwerke verfügbar. Eine direkte Exposition der Datenbanken nach außen ist im Standard-Setup nicht vorgesehen. Mehrere sicherheitsrelevante Maßnahmen wurden bewusst nicht umgesetzt. Dazu zählen eine zentrale Authentifizierungs- und Autorisierungslösung, verschlüsselte Kommunikation über HTTPS oder SSL, Multi-Faktor-Authentifizierung, ein feingranulares Rollen- und Rechtekonzept, Audit-Logging sowie ein dediziertes Secret- oder Key-Management. Ebenso wurde keine hochverfügbare oder cloudbasierte Sicherheitsarchitektur implementiert.

Diese Einschränkungen sind dem Charakter des PoC geschuldet. Ziel war es, die hybride Speicher- und Zugriffsschicht technisch umzusetzen und die Interaktion der Komponenten nachzuweisen. Ein vollständiges Enterprise-Sicherheitsmodell war nicht Gegenstand dieser prototypischen Entwicklung. Für eine produktive Einführung wären zentrale Authentifizierungsmechanismen, verschlüsselte Kommunikation, rollenbasierte Zugriffskontrollen sowie ein erweitertes Sicherheits- und Monitoringkonzept erforderlich. Diese Punkte werden im Evaluationskapitel als Erweiterungsmöglichkeiten diskutiert.

5.6 Einschränkungen des Prototyps

Im Rahmen des Projekteinsatzes bei der KWS war es erforderlich, den Funktionsumfang bewusst einzugrenzen und technische Prioritäten zu setzen. Die nachfolgenden Einschränkungen spiegeln diese bewussten Entscheidungen wider und verdeutlichen den prototypischen Charakter der Umsetzung.

Eine grundlegende Einschränkung betrifft die fehlende Hochverfügbarkeit. Das gesamte System wird lokal auf einem einzelnen Rechner betrieben und über Docker Compose gestartet. Komponenten wie PostgreSQL, RabbitMQ, MinIO, Trino oder Spark laufen jeweils als Einzelinstanzen. Redundante Service-Instanzen sind nicht vorgesehen, ebenso wenig ein Mechanismus zur automatischen Umschaltung im Fehlerfall. Fällt eine zentrale Komponente aus, ist das Gesamtsystem nicht mehr verfügbar.

Ein Clusterbetrieb ist ebenfalls nicht vorgesehen. Weder die Datenbank noch die Analyse- oder Compute-Komponenten sind als Multi-Node-Setup ausgelegt. Die Orchestrierung erfolgt ausschließlich über Docker Compose, ohne Einsatz von Kubernetes oder vergleichbaren Technologien. Die Architektur ist damit nicht auf verteilten Dauerbetrieb ausgelegt.

Ein produktives Identity- und Access-Management ist nicht implementiert. Es gibt keine zentrale Benutzerverwaltung und kein rollenbasiertes Zugriffskonzept. Authentifizierungs- und Autorisierungsmechanismen wurden im Rahmen des PoC , wie bereits erwähnt, nicht umgesetzt. Die Services sind innerhalb der lokalen Umgebung grundsätzlich zugänglich.

Im Bereich der Datenqualität beschränken sich die Maßnahmen auf einfache Status- und Fehlerbehandlungsmechanismen, etwa über die Tabelle chunk_status oder separate Fehlerverzeichnisse. Ein umfassendes Data-Quality-Framework mit Validierungsregeln, Qualitätsmetriken oder automatisierten Reports existiert nicht. Auch eine automatische Korrektur fehlerhafter Datensätze ist nicht vorgesehen.

Ein dedizierter Data Catalog wurde nicht integriert. Die Verwaltung der Tabellen erfolgt ausschließlich über den Iceberg-Katalog, der primär technische Metadaten verwaltet. Funktionen wie Datenlineage, Business-Glossare oder eine erweiterte Metadatensuche sind nicht Bestandteil des Prototyps. Das System ist zudem nicht mandantenfähig. Es existiert keine

logische oder physische Trennung von Daten für unterschiedliche Organisationseinheiten oder Nutzergruppen. Der Prototyp ist für einen einzelnen Nutzungskontext ausgelegt.

Ein strukturiertes Lifecycle-Management wurde ebenfalls nicht umgesetzt. Nach der Ingestion und dem Export verbleiben die Daten dauerhaft im System. Mechanismen zur Archivierung, Retention oder automatischen Löschung sind nicht implementiert. Hinsichtlich Performance wurden keine systematischen Last- oder Stresstests durchgeführt. Die Umgebung ist für Entwicklungs- und Demonstrationszwecke ausgelegt. Zwar werden Metriken über Prometheus und Grafana erfasst, eine automatische Skalierung oder Lastverteilung ist jedoch nicht vorhanden. Die Ressourcenverteilung erfolgt statisch über Docker Compose. Eine horizontale oder vertikale Skalierung im laufenden Betrieb ist nicht vorgesehen. Die Leistungsfähigkeit des Systems ist somit durch die Hardware des Entwicklerrechners begrenzt.

Der Prototyp konzentriert sich bewusst auf die technische Integrationsfähigkeit und die Umsetzung der architektonischen Konzepte. Aspekte wie Hochverfügbarkeit, produktive Sicherheitsmechanismen, umfassende Data-Governance-Strukturen oder skalierbarer Dauerbetrieb sind nicht Bestandteil dieser prototypischen Realisierung. Sie stellen aber mögliche Erweiterungen für eine zukünftige produktive Implementierung dar.

5.7 Test

Zur Überprüfung der prototypischen Implementierung wurden funktionale und technische Tests durchgeführt. Ziel war es, die korrekte Umsetzung der ETL-Pipeline, der Speicher- und Zugriffsschicht sowie der Simulations- und Monitoring-Komponenten nachzuweisen. Im Fokus stand die Funktionsfähigkeit im Rahmen des PoC, nicht eine produktive Lastvalidierung.

Die ETL-Pipeline wurde durch das Einspielen von CSV- und JSON-Testdateien in das Verzeichnis „incoming“ geprüft. Dabei wurde kontrolliert, ob die Dateien korrekt erkannt, in PostgreSQL geladen, nach MinIO exportiert und über etl-register im Iceberg-Katalog eingebunden wurden. Besonderes Augenmerk lag auf der Statusverwaltung in der Tabelle chunk_status und den korrekten Statusübergängen. Zur Überprüfung der Idempotenz wurde dieselbe Datei mehrfach eingespielt. Bereits verarbeitete Daten-Chunks wurden anhand der Statusinformationen erkannt und nicht erneut exportiert. Doppelte Verarbeitung trat im Testbetrieb nicht auf.

Die Zugriffsschicht wurde über SQL-Abfragen in Trino validiert. Ergebnisse wie Datensatzanzahlen oder Aggregationen wurden mit den ursprünglichen Rohdaten in PostgreSQL verglichen, um die korrekte Transformation und Speicherung zu bestätigen.

Im Bereich der Stat-Engine wurden Benchmark-Messungen zu Verarbeitungszeit, Datenvolumen und Durchsatz durchgeführt und in Tabellen wie genotype_stats gespeichert. Simulationen, etwa zu Cloud-Kosten oder ML-Analysen, wurden über WebUI und API gestartet und mit realen Daten oder Referenzwerten verglichen. Fehlerfälle wurden gezielt erzeugt, indem fehlerhafte Dateien eingespielt wurden. Diese wurden korrekt markiert, in ein separates Verzeichnis verschoben und konnten nach Korrektur erneut verarbeitet werden. Inkonsistente Zustände oder Endlosschleifen traten nicht auf.

Während der Tests wurden Laufzeiten und Ressourcennutzung über Prometheus und Grafana beobachtet. Engpässe bei größeren Datenmengen entsprachen den erwartbaren Grenzen einer lokal betriebenen Umgebung. Auch die Monitoring-Komponenten wurden durch simulierte Service-Unterbrechungen validiert. Die Tests zeigen, dass die Pipeline, das Status- und Fehlerhandling, die Idempotenzmechanismen sowie die analytische Abfrage über Trino im Rahmen des Prototyps funktionsfähig sind. Einschränkungen bestehen bei Last, Skalierung und produktiven Anforderungen, die nicht Gegenstand dieser Validierung waren.

6 Darstellung und Bewertung der Ergebnisse

6.1 Prototypische Validierung der Lösung

Das Ziel der prototypischen Validierung war es zu prüfen, ob der entwickelte hybride Data-Lakehouse-Ansatz technisch wie vorgesehen funktioniert. Hierzu wurden die zentralen Verarbeitungsschritte der End-to-End-Pipeline systematisch überprüft und dokumentiert.

Zunächst wurde die Dateningestion validiert. Rohdaten in Form von CSV-, JSON- und Parquet-Dateien wurden automatisiert durch den etl-inject-Service in PostgreSQL übernommen. Anschließend erfolgte durch den etl-export-Service der Export in den S3-kompatiblen Objektspeicher MinIO. Die Daten wurden spaltenorientiert im Parquet-Format gespeichert und über den Iceberg REST-Katalog (JDBC-Backend mit HadoopFileIO) als Iceberg-Tabellen eingebunden. Die Speicherung und Metadatenverwaltung im Lakehouse wurde durch die Prüfung der erzeugten Iceberg-Tabellen sowie der zugehörigen Metadatenstrukturen nachvollzogen.

Im Rahmen der Validierung wurde geprüft, ob Partitionierungsinformationen korrekt übernommen und konsistent im Katalog hinterlegt sind. Die analytische Nutzung erfolgte über SQL-Abfragen in Trino sowie ergänzend über Apache Spark. Durchgeführt wurden vollständige Tabellenabfragen, Aggregationen, Row-Counts sowie partitionierte Zugriffe. Die Abbildung 6.1 zeigt exemplarisch die Ausführung und Überwachung dieser Abfragen. Anhand der von der Trino-Engine bereitgestellten Query-Ausführungsstatistiken lassen sich Aggregations- und Filteroperationen sowie Laufzeit- und Ressourcenkennzahlen (z. B. CPU-Zeit, Speicherverbrauch und verarbeitete Datenmenge) direkt auswerten.

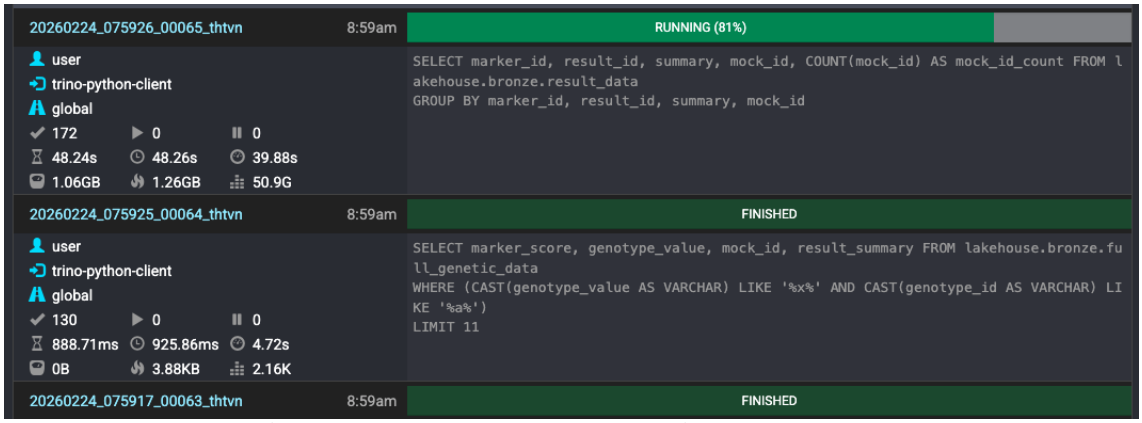


Abbildung 6.1: Ausführung und Monitoring von SQL-Abfragen über Trino im Rahmen der prototypischen Validierung.
Quelle: Eigene Implementierung (Trino Dashboard)

Die Ergebnisse wurden mit den ursprünglichen Quelldaten in PostgreSQL abgeglichen, um Datenintegrität und Vollständigkeit im Rahmen der Tests zu überprüfen. Zur Beobachtung

der Verarbeitungsprozesse wurden Status- und Metadaten-Tabellen eingesetzt, insbesondere zur Nachverfolgung von Export-Chunks. Ergänzend kamen Health-Endpoints sowie ein Monitoring-Setup auf Basis von Prometheus und Grafana zum Einsatz. Darüber hinaus wurden einfache Smoke-Tests und Health-Checks durchgeführt, um die Erreichbarkeit der Services sowie die Konsistenz der Datenbestände zu prüfen. Die Pipeline (Ingestion → Export → Registrierung → Abfrage) wurde mehrfach mit synthetisch generierten Testdatensätzen durchlaufen, um die Funktionsfähigkeit und Stabilität der Architektur zu validieren. Die Ergebnisse waren in den durchgeführten Tests nachvollziehbar und wiederholbar. Es konnten keine Hinweise auf Datenverluste oder inkonsistente Metadatenzustände festgestellt werden. Die im Rahmen des Monitorings erhobenen Fehlermetriken sind exemplarisch in Abbildung 6.2 als Screenshot dargestellt. Die konstante Null-Linie zeigt, dass im betrachteten Zeitraum keine Ingestion-Fehler aufgetreten sind.



Abbildung 6.2: Visualisierung der Metrik `etl_ingestion_errors_gauge` in Grafana während des Validierungszeitraums.
Quelle: Eigene Implementierung (Grafana Dashboard)

Die Überprüfung bezog sich dabei auf die technische Integrität der Datenübertragung, beispielsweise durch den Abgleich von Datensatzanzahlen und einfachen Aggregationen. Eine darüberhinausgehende fachliche oder inhaltliche Datenqualitätsprüfung war nicht Bestandteil der prototypischen Validierung.

Die prototypische Validierung zeigt, dass die zentralen Kernfunktionen insbesondere die automatisierte Ingestion, der strukturierte Export in das Lakehouse, die Iceberg-Registrierung sowie die SQL-basierte Abfrage technisch funktionsfähig implementiert sind. Anwendungsfälle wie Ad-hoc-Analysen, partitionierte Abfragen und Statusüberwachung konnten im Testbetrieb nachvollzogen werden. Nicht Gegenstand der Validierung waren produktive Sicherheitsmechanismen, Hochverfügbarkeits- oder Disaster-Recovery-Konzepte. Diese Aspekte wurden bewusst ausgeklammert, da der Fokus auf der technischen Integrationsfähigkeit der hybriden Speicher- und Zugriffsschicht lag.

Insgesamt zeigt der Prototyp, dass die entwickelte Lösung technisch umsetzbar ist zumindest unter den Bedingungen des Testaufbaus. Damit wird die Forschungsfrage auf der Ebene der prototypischen Umsetzung beantwortet, ohne den Anspruch einer produktionsreifen Systemvalidierung zu erheben.

6.2 Kostensimulationsmodell: Ergebnisse und Annahmen

Zur wirtschaftlichen Einordnung des entwickelten hybriden Data-Lakehouse-Ansatzes wurde im Rahmen des Projekts eine eigene Kostensimulationsumgebung („Stat-Engine“) aufgebaut. Ziel war es, die langfristigen Betriebskosten unterschiedlicher Architekturvarianten

insbesondere Cloud, Hybrid und On-Premises unter vergleichbaren Annahmen modellhaft gegenüberzustellen.

Das Simulationsmodell erlaubt die Simulation von Compute-, Storage- und Transferkosten auf Basis definierter Szenarien. Die Berechnung erfolgt parametrisiert in Abhängigkeit von Datenvolumen, Zugriffshäufigkeit, Workload-Gewichtung sowie der Verteilung von Speicher- und Rechenanteilen zwischen Cloud- und On-Prem-Komponenten.

Die hybride Kostensimulation wird über eine parametrisierte Berechnungsfunktion umgesetzt. Die einzelnen Kostenkomponenten werden getrennt berechnet und anschließend zu Gesamtkosten zusammengeführt. Speicher- und Rechenanteile werden zwischen Cloud- und On-Prem-Komponenten prozentual aufgeteilt. Laufzeitabhängige Kosten werden entsprechend der Simulationsdauer skaliert, während infrastrukturelle Fixkosten separat berücksichtigt werden. Listing 6.1 zeigt einen vereinfachten Auszug der Implementierung dieser Kostenberechnung.

```
def simulate_hybrid_lakehouse_costs(total_gb, queries_per_row):
    """
    Simuliert die Gesamtkosten fr eine Hybrid-Lakehouse-Architektur (Cloud + OnPrem).
    Parameter:
        total_gb: Gesamtspeicherbedarf in GB
        queries_per_row: Anzahl der Abfragen pro Zeile (Workload)
    Modellvariablen:
        cloud_pct, onprem_pct: Verteilung Cloud/OnPrem (z.B. 70/30)
        growth: (optional, fr Zeitreihen)
        tier_rate_hot: OnPrem-Storage-Tarif (EUR/TB/Tag)
        compute_cost_per_server, num_servers: OnPrem-Compute
        DBU/Server-Kosten, Energie, Abschreibung, Wartung, Lizenz, Transfer
    Gibt zurck:
        Dictionary mit Kostenkomponenten und Gesamtkosten
    """
    # Parameterisierung
    cloud_pct = 0.7
    onprem_pct = 0.3
    period_days = 1

    # Storage-Kosten
    aws_storage_gb = total_gb * cloud_pct
    aws_storage_cost = aws_storage_gb * (0.023 / 30) * period_days  # EUR/GB/Tag
    onprem_storage_gb = total_gb * onprem_pct
    onprem_storage_tb = onprem_storage_gb / 1024
    tier_rate_hot = 0.1  # EUR/TB/Tag
    onprem_storage_cost = onprem_storage_tb * tier_rate_hot * period_days

    # Compute-Kosten (OnPrem)
    compute_cost_per_server = 600 / 30  # EUR/Server/Tag
    num_servers = 2
    download_time_per_query_sec = 4 * 60 + 22.853
    total_download_time_sec = download_time_per_query_sec * queries_per_row
    total_query_hours = total_download_time_sec / 3600
    max_server_hours = num_servers * period_days * 24
    utilization = min(1.0, total_query_hours / max_server_hours)
    base_computer_cost = num_servers * compute_cost_per_server * period_days
    variable_compute_cost = base_computer_cost * utilization
    compute_cost = base_computer_cost + variable_compute_cost

    # Energie, Abschreibung, Wartung, Lizenz, Overhead
```

```
    power_kw_per_server = 0.5
    price_per_kwh = 0.30
    energy_cost = power_kw_per_server * 24 * price_per_kwh * num_servers * period_days
    extra_energy_cost = energy_cost * utilization
    energy_cost = energy_cost + extra_energy_cost
    hardware_cost_total = 30000
    lifespan_days = 5 * 365
    depreciation_per_day = hardware_cost_total / lifespan_days
    depreciation_cost = depreciation_per_day * period_days
    internal_maintenance_cost_per_day = 15.0
    internal_maintenance_cost = internal_maintenance_cost_per_day * num_servers *
    period_days
    cooling_overhead_cost = energy_cost * 0.3
    backup_cost = onprem_storage_cost * 0.05
    software_license_cost_per_server = 10.0
    software_license_cost = software_license_cost_per_server * num_servers * period_days

    # Gesamtkosten
    cost_components = {
        'Storage': aws_storage_cost + onprem_storage_cost,
        'Compute': compute_cost,
        'Energy': energy_cost,
        'Depreciation': depreciation_cost,
        'Maintenance': internal_maintenance_cost,
        'Cooling/Facility': cooling_overhead_cost,
        'Backup/DR': backup_cost,
        'Software License': software_license_cost
    }
    cost_components['Total'] = sum(cost_components.values())
    return cost_components
```

Listing 6.1: Parametrische Kostenberechnung einer Hybrid-Lakehouse-Architektur.
Quelle: Eigene Implementierung (VS Code, Python)

Neben statischen Betrachtungen unterstützt das Modell zeitliche Projektionen über frei wählbare Zeiträume (z. B. zehn Jahre). Damit können Kostenverläufe unter konstanten Annahmen berechnet und verglichen werden. Die verwendeten Preisparameter basieren auf dokumentierten Referenzwerten, darunter Storage-Tiers, Query-Kosten pro verarbeitetem Datenvolumen, Databricks-DBU-Sätze, Transferkosten sowie kalkulatorische On-Prem-Betriebskosten. Diese Werte sind als Modellvariablen hinterlegt und können angepasst werden. Zusätzlich berücksichtigt das Modell Simulationsvariablen wie Datenwachstumsraten, Zugriffsmuster, Speicherverteilungen (`cloud_pct` / `onprem_pct`) sowie Transferanteile zwischen Systemen. Auf dieser Basis lassen sich unterschiedliche Nutzungsszenarien berechnen, etwa analyseintensive Phasen oder stärker archivorientierte Workloads. Die Abbildung 6.3 zeigt einen Auszug des Kostensimulationsmodells mit aggregierten Gesamtkosten und deren zeitlichem Verlauf bei identischer Parameterkonfiguration.

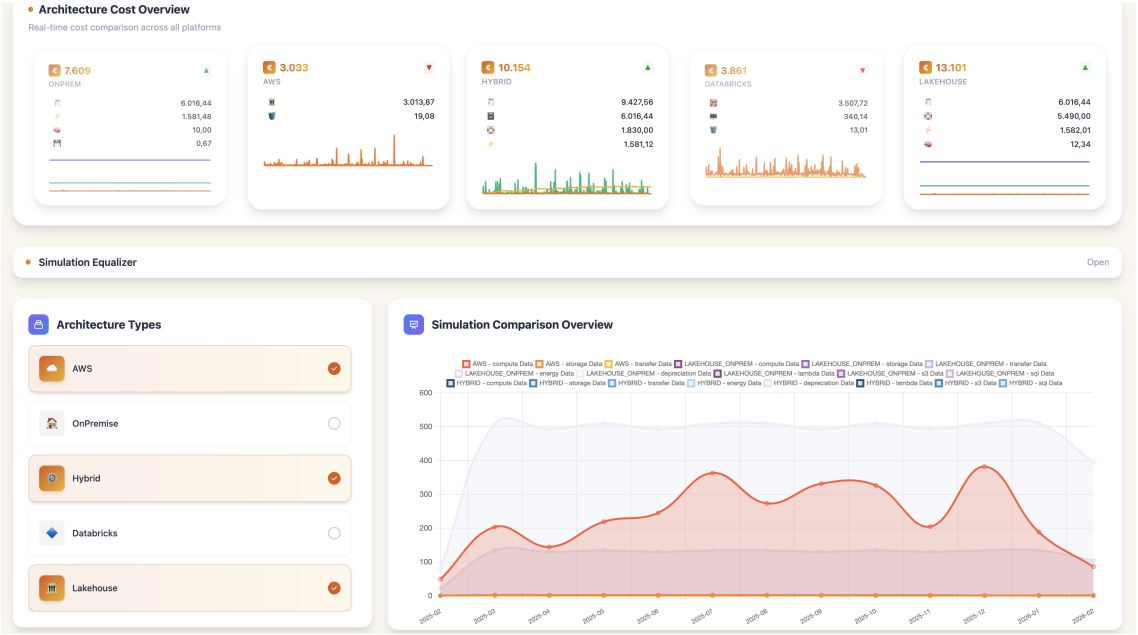


Abbildung 6.3: Dashboard des Kostensimulationsmodells der entwickelten Stat-Engine mit Vergleich ausgewählter Architekturvarianten (AWS, Hybrid, Lakehouse).
Quelle: Eigene Darstellung (Kostensimulationsmodell im UI-Dashboard)

Die Simulation berechnet modellhafte absolute Kostenwerte für die jeweiligen Architekturvarianten auf Basis definierter Parameter. Eine relative Kostenentwicklung wird nicht als eigenständige Kennzahl ausgegeben, sondern ergibt sich aus dem Vergleich dieser absoluten Werte über verschiedene Zeitpunkte und Szenarien hinweg. Die Aussagekraft liegt somit im strukturierten Gegenüberstellen der Varianten unter identischen Annahmen. Dabei ist zu beachten, dass sich das Simulationsmodell noch in einer prototypischen Entwicklungsphase befindet. Die zugrundeliegende Datenbasis ist begrenzt, und nicht alle preisrelevanten Faktoren etwa individuelle Rabattmodelle, vertragsspezifische Konditionen oder organisationsinterne Sondervereinbarungen konnten einbezogen werden. Die verwendeten Preisparameter sind statisch modelliert; dynamische Marktpreisänderungen oder stark schwankende Lastprofile wurden nicht abgebildet. Eine vollständige Abstimmung einzelner Kostenannahmen mit internen KWS-Fachbereichen war im Projektzeitraum eingeschränkt gegeben.

6.3 Bewertung des Prototyps anhand definierter Kriterien

Die Bewertung des Prototyps orientiert sich an den in Kapitel 4 definierten Kriterien und basiert auf den im Rahmen der Implementierung sowie der prototypischen Validierung gewonnenen Erkenntnissen. Im Hinblick auf die Performance wurde die Abfrage- und Verarbeitungsschicht praktisch überprüft. SQL-basierte Analysen über Trino sowie ergänzende Ausführungen über Apache Spark wurden auf Parquet- und Iceberg-Tabellen durchgeführt. In den begleitenden Auswertungen sind konkrete Query-Laufzeiten, Row-Counts und Aggregationsergebnisse dokumentiert. Durch die spaltenorientierte Speicherung im Parquet-Format und die gezielte Partitionierung der Tabellen konnte im Testbetrieb eine effiziente Verarbeitung größerer Datenmengen beobachtet werden. Die ETL-Prozesse wurden mehrfach mit synthetisch generierten Testdatensätzen durchlaufen und mittels

Health-Checks validiert. Die Abläufe erwiesen sich als stabil und wiederholbar. Einschränkungen zeigten sich insbesondere bei speicherintensiven Voll-Load-Szenarien in Spark. Bei der Verarbeitung größerer Datenmengen stieg der Ressourcenbedarf deutlich an, was in der lokalen Testumgebung zu längeren Laufzeiten führte. Diese Beobachtung ist auf die begrenzte Hardwareausstattung des Entwicklerrechners zurückzuführen und verdeutlicht die Abhängigkeit der Performance von verfügbaren Infrastrukturressourcen.

Bezüglich der Skalierbarkeit ist die Architektur vollständig containerisiert und modular aufgebaut. Die einzelnen Komponenten, etwa ETL-Services, Objektspeicher, Katalog, Analyse Engines und Monitoring sind voneinander getrennt implementiert. Konzeptionell erlaubt dieser Aufbau eine horizontale Erweiterung einzelner Dienste. Eine tatsächliche Skalierung unter produktionsnaher Last wurde im Rahmen des PoC jedoch nicht durchgeführt. Die Skalierbarkeit ist daher primär architektonisch angelegt und abhängig von den jeweiligen Infrastrukturressourcen. Die wirtschaftliche Einordnung erfolgt auf Basis des entwickelten Kostensimulationsmodells. Dieses ermöglicht den strukturierten Vergleich unterschiedlicher Betriebsmodelle unter identischen Annahmen. Die Simulation berechnet modellhafte absolute Kostenwerte, aus denen sich durch Vergleich unterschiedliche Kostenverläufe ableiten lassen. Die Aussagekraft ist im Rahmen der definierten Parameter gegeben, bleibt jedoch abhängig von den zugrunde gelegten Preisannahmen und Vereinfachungen des Modells.

Hinsichtlich der Komplexität zeigt sich, dass der Prototyp modular aufgebaut ist und einer klaren Trennung zwischen Ingestion-, Speicher-, Katalog-, Analyse- und Monitoring-Komponenten folgt. Die Nutzung standardisierter Technologien erleichtert die Erweiterbarkeit. Gleichzeitig führt die verteilte Struktur zu einer erhöhten betrieblichen Komplexität, da Orchestrierung, Monitoring und Ressourcenmanagement entsprechendes Fachwissen erfordern.

Auch die Integrationsfähigkeit wurde im prototypischen Rahmen technisch demonstriert. Die Architektur basiert auf offenen Schnittstellen wie SQL, REST und Messaging, wodurch eine Anbindung an Analyse- oder BI-Werkzeuge grundsätzlich möglich ist. Eine vollständige Integration in bestehende Unternehmenssysteme war jedoch nicht Bestandteil des Projekts.

Zusammenfassend erfüllt der Prototyp die definierten Kriterien in wesentlichen Punkten, insbesondere im Hinblick auf die modulare Struktur und die technische Umsetzbarkeit einer hybriden Speicher- und Zugriffsschicht. Einschränkungen bestehen vor allem in der fehlenden produktionsnahen Lastvalidierung sowie in der erhöhten Komplexität verteilter Systeme. Die Ergebnisse zeigen, dass die gewählte Architektur technisch realisierbar ist, wobei weiterführende Maßnahmen für einen produktiven Einsatz erforderlich wären.

6.4 Identifizierte Optimierungspotenziale

Auf Basis der prototypischen Implementierung sowie der zuvor dargestellten Ergebnisse lassen sich mehrere Weiterentwicklungsmöglichkeiten ableiten. Diese ergeben sich aus dem prototypischen Charakter der Umsetzung und dem begrenzten Projektzeitraum.

Im Bereich der Datenverarbeitung besteht Potenzial zur Verfeinerung der Partitionierungsstrategie. Die bisherige Partitionierung orientiert sich an grundlegenden Kriterien. Eine differenziertere oder dynamisch angepasste Partitionierung – etwa nach Zeitintervallen oder

fachlichen Attributen – könnte die Selektivität von Abfragen erhöhen und potenziell zu kürzeren Laufzeiten führen. Derzeit werden keine zusätzlichen Caching-Mechanismen auf Anwendungsebene eingesetzt. Der gezielte Einsatz von Query-Caching oder In-Memory-Caching für wiederkehrende Abfragen könnte insbesondere bei häufig genutzten Reports die Systemlast reduzieren. Neben Trino und Apache Spark wurden keine alternativen Query-Engines vergleichend untersucht. Für bestimmte Workloads könnten andere Engines unterschiedliche Optimierungsansätze bieten. Eine systematische Gegenüberstellung verschiedener Abfrage-Engines stellt daher ein mögliches Erweiterungsfeld dar.

Architektonisch wäre eine klarere Trennung von Batch- und interaktiven Workloads denkbar. Der aktuelle Aufbau nutzt eine gemeinsame Infrastruktur. Eine separate Ressourcenbereitstellung könnte unterschiedliche Lastprofile stabiler abbilden. Im Bereich der Automatisierung bestehen ebenfalls Erweiterungsmöglichkeiten. Während zentrale ETL-Prozesse automatisiert ablaufen, könnten zusätzliche Mechanismen wie eventbasierte Workflows oder erweiterte Orchestrierungslogiken implementiert werden. Auch ein Ausbau der Überwachungs- und Reportingfunktionen wäre denkbar.

Im Hinblick auf den produktiven Betrieb wären darüber hinaus weiterführende Sicherheitsmaßnahmen erforderlich, darunter rollenbasierte Zugriffskonzepte, verschlüsselte Kommunikation sowie ein strukturiertes Secrets-Management. Ebenso wäre ein erweitertes Data-Governance-Konzept mit Metadatenmanagement, Lineage-Tracking und klar definierten Aufbewahrungsregeln zu berücksichtigen. Das entwickelte Kostensimulationsmodell bietet eine Grundlage für strategische Szenarienvergleiche. Für einen operativen Einsatz könnte es um Mechanismen zur kontinuierlichen Kostenüberwachung ergänzt werden.

Die genannten Punkte stellen mögliche Weiterentwicklungen dar, die über den Rahmen des PoC hinausgehen. Der Prototyp dient dem Nachweis der technischen Machbarkeit; weiterführende Optimierungen würden vor allem Aspekte des langfristigen Betriebs betreffen.

6.5 Gültigkeit und Abgrenzung der Ergebnisse

Die Aussagekraft der in dieser Abschlussrbeit erzielten Ergebnisse ist im Kontext eines prototypischen Aufbaus zu betrachten. Sowohl die technische Validierung als auch die wirtschaftliche Bewertung beziehen sich ausdrücklich auf ein PoC und nicht auf einen produktiven Dauerbetrieb unter realen Unternehmensbedingungen. Die Ergebnisse sind für Architekturen übertragbar, die auf vergleichbaren technologischen Prinzipien basieren, insbesondere auf containerisierten, modular aufgebauten Systemen mit S3-kompatiblem Objektspeicher, einem Lakehouse-Format wie Apache Iceberg sowie SQL-basierten Query-Engines wie Trino oder Spark. Die getesteten Workloads – darunter Tabellen mit mehreren Millionen Datensätzen, partitionierte Parquet-Strukturen sowie typische Aggregationsabfragen – entsprechen praxisnahen Szenarien analytischer Datenumgebungen mittlerer Größenordnung. Die technische Machbarkeit sowie die Wiederholbarkeit der ETL-Prozesse konnten unter diesen Rahmenbedingungen nachvollziehbar nachgewiesen werden. Aussagen über Performance und Stabilität beziehen sich jedoch auf die getestete Infrastruktur und die eingesetzten Ressourcen der lokalen Entwicklungsumgebung.

Die prototypische Implementierung ist nicht für einen produktiven 24/7 Betrieb mit geschäftskritischen Daten ausgelegt. Umfassende Sicherheitsmechanismen, Hochverfügbarkeitskonzepte oder Disaster-Recovery-Strategien wurden bewusst nicht umgesetzt. Ebenso

erfolgte keine vollständige Lastvalidierung unter realen Produktionsbedingungen. Auch die wirtschaftliche Bewertung über das entwickelte Kostensimulationsmodell basiert auf definierten Modellannahmen und dokumentierten Preisparametern. Dynamische Preisentwicklungen, individuelle Rabattvereinbarungen oder unternehmensspezifische Vertragskon-ditionen wurden nicht berücksichtigt.

Die Simulation erlaubt daher keine exakten absoluten Kostenprognosen, sondern dient dem strukturierten Vergleich unterschiedlicher Architekturvarianten unter identischen Annahmen. Die unmittelbare Übertragbarkeit auf andere Unternehmen ist nur eingeschränkt möglich, da bestehende IT-Landschaften, regulatorische Anforderungen, Datenvolumen und Nutzungsmuster stark variieren können. Übertragbar sind jedoch grundlegende Architekturprinzipien wie der modulare Aufbau, die Entkopplung von Speicherung und Abfrage durch ein offenes Lakehouse-Format sowie die Integration von Monitoring- und Statusmechanismen.

Der eigene Beitrag im Projekt konzentrierte sich auf die Konzeption und prototypische Umsetzung der hybriden Speicher- und Zugriffsschicht sowie auf die Entwicklung der Kostensimulationslogik. Unternehmensspezifische Rahmenbedingungen, strategische Vorgaben und bestehende Infrastrukturkomponenten wurden berücksichtigt, lagen jedoch außerhalb des direkten Implementierungsumfangs dieser Arbeit.

Zusammenfassend sind die Ergebnisse im Rahmen des gewählten PoC-Ansatzes methodisch nachvollziehbar und technisch belastbar. Ihre Aussagekraft bezieht sich auf die untersuchten Szenarien und Annahmen; eine produktive Umsetzung würde weitergehende technische, organisatorische und sicherheitsrelevante Maßnahmen erfordern.

6.6 Beantwortung der Forschungsfrage

Auf Grundlage der prototypischen Implementierung sowie der in den vorangegangenen Abschnitten dargestellten technischen und wirtschaftlichen Evaluation lässt sich die zentrale Forschungsfrage wie folgt beantworten:

Ein Data Lakehouse kann um eine hybride Speicher- und Zugriffsschicht erweitert werden, indem operative Datenspeicherung, objektbasierter Storage und analytische Zugriffsebene technisch entkoppelt und über definierte Schnittstellen miteinander verbunden werden. Im Rahmen dieser Arbeit wurde dies durch die Kombination aus PostgreSQL als operativer Datenbasis, MinIO als S3-kompatiblem Objektspeicher, Apache Iceberg zur Verwaltung tabellarischer Metadaten sowie Trino und Spark als analytische Abfrage-Engines umgesetzt. Die End-to-End-Verarbeitung von Ingestion über Export und Registrierung bis hin zur SQL-basierten Analyse wurde mehrfach reproduzierbar durchgeführt. Damit konnte die technische Machbarkeit einer solchen hybriden Erweiterung nachgewiesen werden.

Bezüglich der Kosten im Vergleich zu alternativen Lösungsansätzen zeigt die entwickelte Simulation, dass sich reine Cloud-, reine On-Premises- und hybride Modelle unter identischen Annahmen systematisch gegenüberstellen lassen. Die Simulation berechnet absolute Kostenwerte für verschiedene Architekturvarianten; die Bewertung erfolgt durch deren strukturierten Vergleich. Dabei wird deutlich, dass hybride Modelle insbesondere bei steigenden Datenvolumina oder differenzierten Zugriffsmustern wirtschaftliche Vorteile gegenüber

rein cloudbasierten Ansätzen bieten können, während bei geringem Datenvolumen oder konstanten Workloads auch einfache Architekturen ausreichend sein können.

Im Vergleich zu historisch gewachsenen, überwiegend statischen Infrastrukturmodellen zeigt sich, dass eine stärkere Trennung von Speicher- und Verarbeitungsschicht eine flexiblere Ressourcenallokation ermöglicht. Während klassische On-Prem-Strukturen häufig langfristig gebundene Ressourcen erfordern, erlaubt eine hybride Architektur eine differenziertere Verteilung von Daten zwischen kostenintensiven und kostengünstigeren Speicherklassen. Die Kostenentwicklung hängt dabei wesentlich von Datenwachstum, Zugriffshäufigkeit und Transferanteilen ab. Eine pauschal kostengünstigste Lösung lässt sich nicht benennen. Unter den im Projekt betrachteten Szenarien zeigt sich jedoch, dass hybride Modelle für Organisationen mit heterogenen Workloads und wachsendem Datenbestand ein ausgewogenes Verhältnis zwischen Skalierbarkeit und Kostenkontrolle ermöglichen können. Die konkrete wirtschaftlich sinnvollste Variante ist stets abhängig von unternehmensspezifischen Parametern.

Hinsichtlich der technologischen Anforderungen wurde deutlich, dass eine solche Erweiterung eine modular aufgebaute, containerisierte Architektur, standardisierte Schnittstellen (z.B. REST und SQL), ein offenes Lakehouse-Format zur Entkopplung von Speicherung und Abfrage sowie ein strukturiertes Monitoring- und Metadatenkonzept voraussetzt. Grenzen ergeben sich aus verfügbaren Infrastrukturressourcen, Sicherheitsanforderungen, organisatorischen Rahmenbedingungen sowie dem Integrationsaufwand in bestehende Systemlandschaften.

Zusammenfassend zeigt die Arbeit, dass eine hybride Speicher- und Zugriffsschicht technisch realisierbar ist und sowohl funktional als auch wirtschaftlich strukturiert bewertet werden kann. Die Forschungsfragen können im Rahmen des gewählten PoC-Ansatzes auf technischer und konzeptioneller Ebene positiv beantwortet werden, wobei produktive Umsetzungsszenarien zusätzliche infrastrukturelle und organisatorische Maßnahmen erfordern würden.

7 Zusammenfassung

Diese Abschlussarbeit hatte das Ziel zu untersuchen, wie ein Data-Lakehouse-Ansatz um eine hybride Speicher- und Zugriffsschicht erweitert werden kann, um sowohl Cloud- als auch On-Premises-Daten in einer gemeinsamen Architektur effizient und wirtschaftlich zu verwalten. Ausgangspunkt war die Fragestellung, wie eine solche Erweiterung technisch realisierbar ist, welche Kostenwirkungen sich ergeben und welche technologischen Voraussetzungen sowie Grenzen dabei zu berücksichtigen sind.

Im Rahmen eines sechsmonatigen Projekts bei der KWS SAAT SE & Co. KGaA wurde eine prototypische Architektur konzipiert und in einer isolierten Projektumgebung technisch aufgebaut. Die vorliegende Abschlussarbeit analysiert diesen Prototyp systematisch und ordnet die gewonnenen Erkenntnisse wissenschaftlich ein. Die technische Ausgestaltung des Prototyps zeigte, dass sich operative Datenspeicherung, objektbasierter Storage und analytische Zugriffsschicht voneinander entkoppeln und dennoch konsistent integrieren lassen. Die End-to-End-Verarbeitung von der Datenaufnahme über Export und Registrierung bis hin zur SQL-basierten Analyse konnte reproduzierbar durchgeführt werden.

Ergänzend zur technischen Validierung wurde ein Kostensimulationsmodell entwickelt, mit der unterschiedliche Architekturvarianten (Cloud, Hybrid und On-Premises) unter identischen Annahmen vergleichend bewertet werden können. Die Simulation liefert keine exakten absoluten Kostenvorhersagen, erlaubt jedoch einen strukturierten Vergleich der Kostenentwicklung in Abhängigkeit von Datenvolumen, Zugriffsmustern und Speicherverteilung. Die Ergebnisse zeigen, dass hybride Modelle insbesondere bei heterogenen Workloads und wachsenden Datenbeständen eine ausgewogene Balance zwischen Skalierbarkeit und Kostenkontrolle ermöglichen können. Eine pauschal günstigste Lösung existiert nicht; die Wirtschaftlichkeit ist stets kontext- und workloadabhängig.

Aus architektonischer Sicht wurde deutlich, dass eine modulare, containerisierte Struktur mit klar definierten Schnittstellen eine tragfähige Grundlage für hybride Datenlandschaften bildet. Die Entkopplung von Speicherung und Abfrage über ein offenes Lakehouse-Format schafft Flexibilität bei der Wahl von Analysewerkzeugen und Infrastrukturkomponenten. Gleichzeitig erhöht eine solche Architektur die betriebliche Komplexität, da Monitoring, Ressourcenmanagement und Integrationsaufwand sorgfältig geplant werden müssen.

Im Verlauf der Arbeit hat sich gezeigt, dass der gewählte prototypische Ansatz sowohl Vorteile als auch Grenzen mit sich bringt. Positiv hervorzuheben ist die strukturierte Nachvollziehbarkeit der End-to-End-Pipeline, die reproduzierbare Verarbeitung realer und simulierter Datensätze sowie die Möglichkeit, technische und wirtschaftliche Fragestellungen in einem konsistenten Modell zusammenzuführen. Einschränkungen ergaben sich insbesondere aus dem bewusst gewählten PoC-Rahmen, den begrenzten Ressourcen der Testumgebung sowie vereinfachenden Annahmen innerhalb der Kostensimulation. Diese Aspekte stellen keine konzeptionellen Schwächen dar, sondern markieren den Übergangspunkt zwischen prototypischer Validierung und produktiver Systemgestaltung. Die

Ergebnisse sind im Kontext des Unternehmensumfelds einzuordnen, in dem steigende Datenvolumina, zunehmende Analyseanforderungen und heterogene Systemlandschaften zu beobachten sind.

Für Unternehmen wie die KWS SAAT SE & Co. KGaA kann eine hybride Lakehouse-Architektur eine strukturierte Grundlage bieten, um bestehende On-Premises-Strukturen schrittweise mit cloudnahen Komponenten zu verbinden, ohne sich frühzeitig vollständig auf ein einzelnes Betriebsmodell festzulegen. Der entwickelte Prototyp stellt dabei keine fertige Produktlösung dar, sondern eine belastbare Entscheidungs- und Diskussionsgrundlage für strategische Architekturfragen.

Ein zentraler Mehrwert der Arbeit liegt in der Verbindung von technischer Machbarkeitsprüfung und wirtschaftlicher Bewertung innerhalb eines konsistenten Modells. Die prototypische Validierung zeigt, dass auch mit begrenzten Ressourcen eine strukturierte Analyse komplexer Architekturentscheidungen möglich ist. Damit wird deutlich, dass prototypische Implementierungen nicht nur technische Experimente darstellen, sondern als fundierte Entscheidungsbasis für weiterführende Investitions- und Transformationsprozesse dienen können.

Ausblickend ergeben sich mehrere Weiterentwicklungspotenziale. Technisch könnten Partitionierungsstrategien, Ressourcenallokation und Workload-Trennung weiter optimiert werden. Eine produktive Umsetzung würde zudem Themen wie Hochverfügbarkeit, Sicherheit, Governance, Mandantenfähigkeit sowie automatisiertes Lifecycle-Management erfordern.

Im Bereich der Kostensimulation besteht Potenzial zur Integration dynamischer Preisentwicklungen, unternehmensspezifischer Vertragsmodelle oder realer Betriebskennzahlen. Auch die stärkere Verzahnung mit Data-Governance- und Metadatenframeworks bietet Ansatzpunkte für zukünftige Arbeiten.

Zusammenfassend lässt sich festhalten, dass hybride Data-Lakehouse-Architekturen einen pragmatischen Ansatz zur Integration heterogener Datenlandschaften darstellen können. Sie ermöglichen eine flexible Kombination aus On-Premises- und Cloud-Komponenten und schaffen eine gemeinsame Zugriffsebene für analytische Workloads. Die vorliegende Arbeit zeigt, dass eine solche Architektur technisch umsetzbar und wirtschaftlich strukturiert bewertbar ist. Gleichzeitig verdeutlicht sie, dass jede konkrete Ausgestaltung von den spezifischen organisatorischen, technischen und wirtschaftlichen Rahmenbedingungen abhängt.

Der wesentliche Erkenntnisgewinn dieser Arbeit besteht somit weniger in der Identifikation einer universellen „besten“ Lösung, sondern in der methodischen Herangehensweise: Durch eine prototypische Validierung in Verbindung mit einer transparenten Kostenmodellierung können komplexe Architekturentscheidungen nachvollziehbar analysiert und fundiert vorbereitet werden. Über den konkreten Anwendungsfall hinaus zeigt die Arbeit, dass prototypische Architekturvalidierung in Kombination mit transparenter Kostenmodellierung ein geeignetes methodisches Instrument darstellt, um komplexe Transformationsentscheidungen in datenintensiven Organisationen strukturiert vorzubereiten.

Quellenverzeichnis

[AAA2016] Alnoukari, M, I Alsmadi und A Al*Zoubi* (2016). Comparative Study of Data Warehouses Modeling Approaches: Inmon, Kimball and Data Vault. https://www.researchgate.net/publication/312486486_Comparative_study_of_data_warehouses_modeling_approaches_Inmon_Kimball_and_Data_Vault (abgerufen am 07. 02. 2026).

[A2026] Amazon Web Services (2026a). Amazon Aurora Architecture. Official documentation. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/Concepts.AuroraArchitecture.html (abgerufen am 19. 02. 2026).

[A2026] Amazon Web Services (2026b). Amazon Redshift Pricing. Pricing information. https://aws.amazon.com/redshift/pricing/ (abgerufen am 18. 02. 2026).

[A2026] Amazon Web Services (2026c). AWS CLI Command Reference. Official documentation. https://docs.aws.amazon.com/cli/latest/reference/ (abgerufen am 19. 02. 2026).

[A2026] Apache Software Foundation (2026a). Apache Flink Documentation – Architecture. Official documentation. https://nightlies.apache.org/flink/flink-docs-stable/docs/concepts/flink-architecture/ (abgerufen am 01. 02. 2026).

[A2026] Apache Software Foundation (2026b). Apache Iceberg Documentation – Overview. https://iceberg.apache.org/docs/latest/ (abgerufen am 19. 02. 2026).

[A2026] Apache Software Foundation (2026c). Apache Kafka Documentation – Architecture. Official documentation. https://kafka.apache.org/documentation/#design (abgerufen am 01. 02. 2026).

[A2026] Apache Software Foundation (2026d). Apache NiFi Overview (Features and Architecture). Official documentation. https://nifi.apache.org/docs/nifi-docs/html/overview.html (abgerufen am 19. 02. 2026).

[A2026] Apache Software Foundation (2026e). Apache Spark Documentation. Official documentation. https://spark.apache.org/docs/latest/ (abgerufen am 19. 02. 2026).

[A2026] Apache Software Foundation (2026f). Iceberg Table Specification. https://iceberg.apache.org/spec/ (abgerufen am 19. 02. 2026).

[A2026] Apache Software Foundation (2026g). REST Catalog Specification – Apache Iceberg. https://iceberg.apache.org/rest-catalog-spec/ (abgerufen am 19. 02. 2026).

[A2021] Armbrust, Michael u. a. (2021a). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. In: Proceedings of the VLDB Endowment 13.12, S. 3411–3424. https://www.vldb.org/pvldb/vol13/p3411-armbrust.pdf (abgerufen am 19. 02. 2026).

[A2021] Armbrust, Michael u. a. (2021b). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In: CIDR 2021 (Conference on Innovative Data Systems Research). https://www.cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf (abgerufen am 08. 02. 2026).

[A2026] Atlantic Council (2026). Digital Sovereignty: Europe's Declaration of Independence. https://www.atlanticcouncil.org/in-depth-research-reports/report/digital-sovereignty-europes-declaration-of-independence/ (abgerufen am 23. 01. 2026).

[B2023] Boston Consulting Group (2023). A New Architecture to Manage Data Costs and Complexity. https://www.bcg.com/publications/2023/new-data-architectures-can-help-manage-data-costs-and-complexity (abgerufen am 07. 02. 2026).

[B2026] Bundesnetzagentur (2026). EU Data Act – Informationen und Hintergründe. https://www.bundesnetzagentur.de/DE/Fachthemen/Digitales/DataAct/Hintergrund/start.html (abgerufen am 07. 02. 2026).

[C2026] Celery Project (2026). Celery Documentation. Official documentation. https://docs.celeryq.dev/en/stable/ (abgerufen am 01. 02. 2026).

[CML2014] Chen, Min, Shiwen Mao und Yunhao Liu (2014). Big Data: A Survey. In: Mobile Networks and Applications 19.2, S. 171–209. DOI: 10.1007/s11036-013-0489-0. https://link.springer.com/article/10.1007/s11036-013-0489-0 (abgerufen am 07. 02. 2026).

[D2026] Datadog Inc. (2026). Datadog – Product Overview. Product overview. https://www.datadoghq.com/product/ (abgerufen am 01. 02. 2026).

[DS2023] Dittmar, Carsten und Peter Schulz (2023). Architekturen und Technologien für den Data Lake. In: Künstliche Intelligenz und Data Science in Theorie und Praxis. Von Algorithmen und Methoden zur praktischen Umsetzung in Unternehmen. Hrsg. von Andreas Gillhuber, Göran Kauermann und Wolfgang Hauner. Berlin, Heidelberg: Springer Spektrum, S. 157–166. ISBN: 978-3-662-66277-9.

[D2026] Django REST Framework (2026). DRF Documentation. Official documentation. https://www.django-rest-framework.org/ (abgerufen am 19. 02. 2026).

[D2026] Django Software Foundation (2026). Django Deployment and ASGI Support. Official documentation. https://docs.djangoproject.com/en/stable/howto/deployment/asgi/ (abgerufen am 19. 02. 2026).

[E2022] Europäische Kommission (23. Feb. 2022). Proposal for a Regulation of the European Parliament and of the Council on harmonised rules on fair access to and use of data (Data Act). COM(2022) 68 final vom 23. Februar 2022. https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=COM:2022:68:FIN (abgerufen am 07. 02. 2026).

[E2023] Europäisches Parlament und Rat der Europäischen Union (13. Dez. 2023). Verordnung (EU) 2023/2854 über harmonisierte Regeln für fairen Zugang zu und Nutzung von Daten (Data Act). https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32023R2854 (abgerufen am 07. 02. 2026).

[F2011] Farkisch, Klaus (2011). Data-Warehouse-Systeme kompakt – Aufbau, Architektur, Grundfunktionen. Berlin Heidelberg: Springer. ISBN: 978-3-642-21532-2.

[FM2011] Fette, Ian und Alexey Melnikov (2011). The WebSocket Protocol. Techn. Ber. RFC 6455. IETF. https://datatracker.ietf.org/doc/html/rfc6455 (abgerufen am 01. 02. 2026).

[G2026] Google Cloud (2026a). Architectural Overview of Pub/Sub. Architecture documentation. https://docs.cloud.google.com/pubsub/architecture (abgerufen am 19. 02. 2026).

[G2026] Google Cloud (2026b). Cloud SQL Overview. Official documentation. https://cloud.google.com/sql/docs (abgerufen am 19. 02. 2026).

[G2026] Google Cloud (2026c). Service Infrastructure – Overview. Architecture documentation. https://docs.cloud.google.com/service-infrastructure/docs/overview (abgerufen am 19. 02. 2026).

[H2012] Hardt, Dick (2012). The OAuth 2.0 Authorization Framework. RFC 6749, Internet Engineering Task Force (IETF). https://datatracker.ietf.org/doc/html/rfc6749 (abgerufen am 19. 02. 2026).

[I2025] IBM (2025). Event-Driven Architecture. Overview article. https://www.ibm.com/topics/event-driven-architecture (abgerufen am 18. 02. 2026).

[ILS2021] Inmon, Bill, Mary Levins und Ranjeet Srivastava (2021). Building the Data Lakehouse. 1. Aufl. Basking Ridge, NJ: Technics Publications. ISBN: 978-1-634622-966-9.

[J2026] Jaeger Authors (2026). Jaeger Architecture. Official documentation. https://www.jaegertracing.io/docs/latest/architecture/ (abgerufen am 01. 02. 2026).

[J2022] Janssen, N. E. (2022). Examining the Value of the Data Lakehouse Architecture. Master's thesis. University of Twente.

https://essay.utwente.nl/92801/1/Janssen_MA_EEMCS.pdf (abgerufen am 31. 01. 2026).

[J2022] Janssen, Niels E. (2022). Examining the Value of the Data Lakehouse Architecture. Magisterarb. University of Twente. https://essay.utwente.nl/92801/1/Janssen_MA_EEMCS.pdf (abgerufen am 19. 02. 2026).

[K2026] KWS SAAT SE & Co. KGaA (2026). Unternehmenswebsite. https://www.kws.com (abgerufen am 07. 02. 2026).

[M2026] MariaDB Foundation (2026a). Galera Cluster. Official documentation. https://mariadb.com/kb/en/galera-cluster/ (abgerufen am 19. 02. 2026).

[M2026] MariaDB Foundation (2026b). MariaDB Documentation. Official documentation. https://mariadb.com/kb/en/documentation/ (abgerufen am 19. 02. 2026).

[M2026] Memcached Project (2026). Memcached Documentation. Official documentation. https://memcached.org/ (abgerufen am 19. 02. 2026).

[M2022] Menon, Pradeep (2022). Data Lakehouse in Action: Architecting a Modern and Scalable Data Analytics Platform. Birmingham: Packt Publishing Ltd. ISBN: 978-1-80181-593-2.

[M2025] Microsoft (2025a). Azure Control Plane and Data Plane. Architecture documentation. https://learn.microsoft.com/en-us/azure/azure-resource-manager/management/control-plane-and-data-plane (abgerufen am 19. 02. 2026).

[M2025] Microsoft (2025b). Extract, Transform, and Load (ETL). Technical documentation. https://learn.microsoft.com/en-us/azure/architecture/data-guide/relational-data/etl (abgerufen am 18. 02. 2026).

[M2026] Microsoft (2026a). Cache-Aside Pattern. Architecture pattern. https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside (abgerufen am 19. 02. 2026).

[M2026] Microsoft (2026b). Single Page Applications (SPA). Technical documentation. https://learn.microsoft.com/en-us/aspnet/core/client-side/spa/ (abgerufen am 19. 02. 2026).

[M2026] Microsoft Corporation (2026a). BI Solution Architecture in the Center of Excellence – Power BI. Technical documentation. https://learn.microsoft.com/en-us/power-bi/guidance/center-of-excellence-business-intelligence-solution-architecture (abgerufen am 19. 02. 2026).

[M2026] Microsoft Corporation (2026b). SQL Server Technical Documentation. Official documentation. https://learn.microsoft.com/en-us/sql/sql-server/ (abgerufen am 19. 02. 2026).

[M2026] MinIO Inc. (2026). MinIO Object Storage Overview. Official documentation. https://min.io/docs/minio/linux/index.html (abgerufen am 19. 02. 2026).

[M2026] Mozilla Developer Network (2026). REST APIs. Technical overview. https://developer.mozilla.org/en-US/docs/Glossary/REST (abgerufen am 19. 02. 2026).

[N2020] National Institute of Standards and Technology (2020). Zero Trust Architecture. Techn. Ber. Special Publication 800-207. NIST. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf (abgerufen am 19. 02. 2026).

[N2026] Node.js Foundation (2026). Node.js – Event Loop and Asynchronous Programming. Technical documentation. https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/ (abgerufen am 19. 02. 2026).

[O2026] OAuth2 Proxy Authors (2026). OAuth2 Proxy Documentation. Official documentation. https://oauth2-proxy.github.io/oauth2-proxy/ (abgerufen am 19. 02. 2026).

[O2026] Okta/Auth0 (2026). Auth0 Documentation. Official documentation. https://auth0.com/docs (abgerufen am 19. 02. 2026).

[O2026] Open Policy Agent Authors (2026). OPA Documentation. Official documentation. https://www.openpolicyagent.org/docs/ (abgerufen am 19. 02. 2026).

[O2026] OpenJS Foundation (2026). Express.js Documentation. Official documentation. https://expressjs.com/ (abgerufen am 19. 02. 2026).

[O2025] Oracle Corporation (2025). Database Concepts. Technical documentation. https://docs.oracle.com/cd/F39414_01/cncpt/database-concepts.pdf (abgerufen am 19. 02. 2026).

[O2026] Oracle Corporation (2026a). Database Licensing Information User Manual. Licensing documentation. https://docs.oracle.com/en/database/oracle/oracle-database/21/dblic/database-licensing-information-user-manual.pdf (abgerufen am 19. 02. 2026).

[O2026] Oracle Corporation (2026b). MySQL Reference Manual. Official documentation. https://dev.mysql.com/doc/ (abgerufen am 19. 02. 2026).

[O2026] Oracle Corporation (2026c). Oracle Real Application Clusters (RAC). Official documentation. https://www.oracle.com/database/real-application-clusters/ (abgerufen am 19. 02. 2026).

[P2026] Pallets Project (2026). Flask Documentation. Official documentation. https://flask.palletsprojects.com/ (abgerufen am 19. 02. 2026).

[P2026] PostgreSQL Global Development Group (2026). PostgreSQL Documentation. Official documentation. https://www.postgresql.org/docs/ (abgerufen am 19. 02. 2026).

[P2026] Presto Foundation (2026). PrestoDB Documentation. Official documentation. https://prestodb.io/docs/current/ (abgerufen am 01. 02. 2026).

[PNM2022] Priebe, Torsten, Sebastian Neumaier und Stefan Markus (2022). Von Data Warehouse bis Data Mesh: Ein Wegweiser durch den Dschungel analytischer Datenarchitekturen. arXiv: 2212.03612. https://arxiv.org/pdf/2212.03612 (abgerufen am 07. 02. 2026).

[P2026] Prometheus Authors (2026). Understanding and Using the Multi-Target Exporter Pattern. Official documentation. https://prometheus.io/docs/guides/multi-target-exporter/ (abgerufen am 01. 02. 2026).

[P2026] Python Software Foundation (2026a). ASGI – Asynchronous Server Gateway Interface. Technical specification. https://asgi.readthedocs.io/en/latest/ (abgerufen am 19. 02. 2026).

[P2026] Python Software Foundation (2026b). WSGI – Web Server Gateway Interface. Technical specification. https://peps.python.org/pep-3333/ (abgerufen am 19. 02. 2026).

[Q2026] Qlik (Talend) (2026). Talend Data Integration. Product overview. https://www.qlik.com/us/products/talend-data-integration (abgerufen am 19. 02. 2026).

[R2026] Ramírez, Sebastián (2026). FastAPI Documentation. Official documentation. https://fastapi.tiangolo.com/ (abgerufen am 19. 02. 2026).

[R2026] Red Hat Inc. (2026). Keycloak Documentation. Official documentation. https://www.keycloak.org/documentation (abgerufen am 19. 02. 2026).

[R2026] Redis Ltd. (2026). Redis Documentation. Official documentation. https://redis.io/docs/ (abgerufen am 19. 02. 2026).

[RGR2018] Reinsel, David, John Gantz und John Rydning (2018). The Digitization of the World – From Edge to Core. IDC White Paper (Data Age 2025). https://www.seagate.com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper.pdf (abgerufen am 01. 02. 2026).

[S2014] Sakimura, Nat u. a. (2014). OpenID Connect Core 1.0. OpenID Foundation Specification. https://openid.net/specs/openid-connect-core-1_0.html (abgerufen am 19. 02. 2026).

[S2026] Salesforce (Tableau) (2026). Tableau Platform Architecture. Technical documentation. https://help.tableau.com/current/blueprint/en-gb/bp_server_architecture.htm (abgerufen am 19. 02. 2026).

[S2025] Serra, James (2025). Datenarchitekturen. Modern Data Warehouse, Data Fabric, Data Lakehouse und Data Mesh richtig einsetzen. Übers. von

Frank Langenau. 1. Aufl. Heidelberg: dpunkt.verlag. ISBN: 978-3-96009-254-4.

[S2026] Snowflake Inc. (2026a). Snowflake Architecture Overview. Official documentation. https://docs.snowflake.com/en/user-guide/intro-key-concepts (abgerufen am 01. 02. 2026).

[S2026] Snowflake Inc. (2026b). Snowflake Pricing Overview. Pricing information. https://www.snowflake.com/pricing/ (abgerufen am 18. 02. 2026).

[T2021] Ting, Jason u. a. (2021). Apache Iceberg: A Table Format for Huge Analytic Datasets. In: Proceedings of the VLDB Endowment 14.13, S. 3328–3340. https://www.vldb.org/pvldb/vol14/p3328-ting.pdf (abgerufen am 19. 02. 2026).

[T2026] Trino Software Foundation (2026a). Architecture and Concepts. https://trino.io/docs/current/overview/concepts.html (abgerufen am 01. 02. 2026).

[T2026] Trino Software Foundation (2026b). Iceberg Connector Documentation. https://trino.io/docs/current/connector/iceberg.html (abgerufen am 01. 02. 2026).

[V2025] VMware Inc. (2025). RabbitMQ – Overview. Project overview. https://www.rabbitmq.com/ (abgerufen am 18. 02. 2026).

Abkürzungsverzeichnis

AMQP Advanced Message Queuing Protocol

API Application Programming Interface

ASGI Asynchronous Server Gateway Interface

AWS Amazon Web Services

CARG Compound Annual Growth Rate

CSV Comma-Separated Values

DSL Domain Specific Lenguage

ETL Extract, Transform, Load

HTTPS HyperText Transfer Protocol Secure

JDBC Java Database Connectivity

JSON JavaScript Object Notation

JWT JSON Web Token

ML Machine Learning

OLAP Online Analytical Processing

OLTP Online Transaction Processing

OPA Open Policy Agent

PoC Proof of Concept

REST Representational State Transfer

S3 Simple Storage Service

SaaS Software as a Service

SQL Structured Query Language

URL Uniform Resource Locator

WebUI Web User Interface

Anhang

A.1 Bewertung möglicher Technologien

Bewertung möglicher Technologien

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
Tabellenformate für analytische Daten	**Apache Iceberg** **Delta Lake** **Apache Hudi**	Alle drei Technologien unterstützen ACID-Transaktionen und gewährleisten konsistente Datenänderungen. **Apache Iceberg** bietet ein besonders effektives Metadata-, File- und Partition-Pruning und dies wirkt sich positiv auf Query-Latenzen bei großen analytischen Tabellen aus.	**Apache Iceberg:** Geeignet für sehr große Tabellen, effiziente Verwaltung umfangreicher Metadaten, stabil bei langfristigen Datenhistorien. **Apache Hudi:** Unterstützt wachsende Datenbestände, besonders geeignet für kontinuierliche Updates und inkrementelle Verarbeitung. **Delta Lake:** Gute Skalierung bei großen Datenmengen, stärkere Bindung an das Spark-Ökosystem.	**Apache Iceberg:** Cloud- und On-Prem-neutral, kompatibel mit verschiedenen Object-Storage-Lösungen, keine feste Anbieterbindung. **Apache Hudi:** Ebenfalls cloudneutral, engere Kopplung an Processing-Frameworks. **Delta Lake:** Primär im Databricks-Umfeld verbreitet, außerhalb eingeschränkter flexibel.	**Apache Iceberg:** Native Unterstützung für Trino, Integration mit Spark und Flink, geeignet für modulare Lakehouse-Architekturen. **Apache Hudi:** Fokus auf Streaming- und Ingestion-Prozesse, weniger ausgerichtet auf reine analytische Query-Engines. **Delta Lake:** Starke Spark-Integration, eingeschränkte Unterstützung in Trino-Umgebungen.	**Apache Iceberg:** Apache-Open-Source-Projekt, aktive Community, geringe Anbieterabhängigkeit. **Apache Hudi:** Apache-Projekt, aktive Weiterentwicklung, teilweise stärkere Framework-Abhängigkeit. **Delta Lake:** Open-Source (Apache 2.0), Erweiterungen stark mit Databricks verbunden, mögliches Lock-in-Risiko.	Auf Grundlage der festgelegten Bewertungsdimensionen zeigt **Apache Iceberg** die größte Übereinstimmung mit den formulierten Anforderungen und eignet sich besonders für eine engine-unabhängige und hybride Architektur.
Objektbasierte Speicherlösungen (Object Storage)	**MinIO** **Amazon S3** **Ceph**	Hohe Durchsatzraten bei allen Lösungen. **MinIO:** optimiert für lokale und private Umgebungen. **Amazon S3:** hohe Performance im Cloud-Betrieb. **Ceph:** leistungsfähig, jedoch konfigurationsabhängig.	**Amazon S3:** nahezu unbegrenzte Skalierung in AWS. **MinIO:** horizontale Skalierung in Cluster-/Kubernetes-Umgebungen. **Ceph:** hohe Skalierbarkeit, jedoch komplexerer Betrieb.	**MinIO:** vollständig On-Prem-fähig, S3-kompatibel. **Ceph:** On-Prem-fähig. **Amazon S3:** cloudgebunden (AWS-Infrastruktur).	**MinIO:** nahtlose Integration mit Iceberg, Trino, Spark und ETL-Services. **Amazon S3:** funktional kompatibel, Cloud-Anbindung erforderlich. **Ceph:** integrierbar, höherer Administrationsaufwand.	**MinIO:** Open Source. **Ceph:** Open Source. **Amazon S3:** proprietär, Vendor-Lock-in.	Auf Grundlage der Bewertungskriterien zeigt **MinIO** die größte Übereinstimmung mit den architektonischen Anforderungen und stellt die geeignetste Speicherlösung für das hybride Lakehouse dar.
Query Engines & Analytics	**Trino** **Apache Spark SQL** **Presto** **Dremio** **klassische Cloud-Data-Warehouse-Systeme** Zusätzliche Architekturkomponenten: **Query-Engine-Service** **Query-Stats-Service**	**Trino**: optimiert für interaktive SQL-Abfragen, geringe Query-Latenz durch verteilte In-Memory-Verarbeitung. **Presto**: ähnliches Performance-Profil, geringere Weiterentwicklungsdynamik. **Apache Spark SQL**: stark bei Batch- und Transformations-Workloads, höhere Latenzen bei Ad-hoc-Abfragen. **Dremio**: hohe Performance durch zusätzliche Optimierungsmechanismen (z. B. Reflections), plattformabhängig. **klassische Cloud-Data-Warehouse-Systeme**: hohe Performance, jedoch externe Plattformbindung.	Alle Engines unterstützen horizontale Skalierung. **Trino**: effizient bei hoher Nutzerkonkurrenz (High Concurrency). **Apache Spark SQL**: sehr skalierbar für große Batch-Jobs, ressourcenintensiver. **Dremio**: clusterbasierte Skalierung, zusätzlicher Betriebsaufwand. **klassische Cloud-Data-Warehouse-Systeme**: automatische Skalierung, kostenabhängig.	**Trino**: cloudneutral, On-Prem und Cloud einsetzbar. **Apache Spark SQL**: ebenfalls cloudneutral. **Dremio**: hybridfähig, jedoch plattformzentriert. **klassische Cloud-Data-Warehouse-Systeme**: überwiegend cloudgebunden.	**Trino**: native Konnektoren für Iceberg, Object Storage und relationale Datenbanken (z. B. PostgreSQL), unterstützt föderierte Abfragen. **Query-Engine-Service**: kapselt Abfrageausführung und stellt einheitliche Zugriffsschicht bereit. **Query-Stats-Service**: erfasst Laufzeiten, Ressourcennutzung und Datenvolumen zur Analyse und Optimierung. **Apache Spark SQL**: tiefe Integration in ETL- und Processing-Pipelines. **Dremio**: unterstützt Iceberg, ergänzt jedoch zusätzliche semantische Verwaltungsschicht.	**Trino**: aktives Open-Source-Projekt, breite Community, geringes Lock-in-Risiko. **Apache Spark SQL**: etabliert und offen, jedoch eng mit dem Spark-Ökosystem verbunden. **Dremio**: Community-Edition verfügbar, zentrale Funktionen proprietär. **klassische Cloud-Data-Warehouse-Systeme**: proprietär, Vendor-Lock-in und Lizenzabhängigkeit.	**Trino** erfüllt die definierten Anforderungen insgesamt am stimmigsten und kombiniert geringe Latenz, hohe Skalierbarkeit, native Iceberg-Integration sowie eine offene Architektur ohne starke Anbieterbindung.

Fortsetzung auf der nächsten Seite

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
ETL- und Processing-Frameworks	**Apache Spark** **Apache Flink** **Apache Airflow** **ETL-/Sync-Pipeline**	**Apache Spark**: geeignet für umfangreiche Batch-Transformationen mit hoher Parallelisierbarkeit. **Apache Flink**: spezialisiert auf Streaming- und Echtzeitverarbeitung. **Apache Airflow**: Workflow-Orchestrierung ohne eigene Datenverarbeitung. **ETL-/Sync-Pipeline**: inkrementelle, kontrollierte Verarbeitung mit klarer Service-Struktur.	**Apache Spark** und **Apache Flink**: verteilte Skalierung für datenintensive Workloads. **Apache Airflow**: Skalierung auf Orchestrierungsebene. **ETL-/Sync-Pipeline**: modular und serviceorientiert skalierbar.	Alle Frameworks containerisierbar und flexibel einsetzbar. **Apache Airflow**: zusätzliche Infrastruktur erforderlich.	**Apache Spark**: tiefe Integration in Lakehouse-Umgebungen. **Apache Flink**: Streaming-Integration. **Apache Airflow**: Steuerung verteilter ETL-Tasks. **ETL-/Sync-Pipeline**: gezielte Integration in relationale Quellsysteme und Object Storage.	**Apache Spark**, **Apache Flink**, **Apache Airflow**: etablierte Open-Source-Projekte. **ETL-/Sync-Pipeline**: vollständig unter eigener Kontrolle.	Die **eigene Python-basierte ETL-/Sync-Pipeline** stellt im Projektkontext die integrationsnahe und strukturell konsistente Lösung dar, während spezialisierte Frameworks bei großskaligen Workloads Vorteile bieten.
Messaging & Orchestrierung	**RabbitMQ** **Apache Kafka** **Celery** **Apache Airflow** **Apache Flink** **REST-Kommunikation** **cron-basierte Orchestrierung**	**RabbitMQ**: ereignisbasierte, entkoppelte Service-Kommunikation mit geringer Latenz. **Apache Kafka**: hochskalierbare Event-Streaming-Plattform. **Celery**: Task-Queue-Erweiterung mit Retry- und Scheduling-Mechanismen. **Apache Airflow**: zeitbasierte Workflow-Orchestrierung. **Apache Flink**: Streaming-Verarbeitung, für reine Steuerung funktional überdimensioniert. **synchrone REST-Kommunikation**: direkte Kopplung zwischen Services. **cron-basierte Orchestrierung**: einfache zeitgesteuerte Abläufe.	**RabbitMQ** und **Apache Kafka**: horizontal skalierbar. **Celery**: workerbasierte Skalierung. REST- und cron-Ansätze: begrenzte Entkopplung.	Alle Technologien containerisierbar betreibbar. **Apache Kafka** und **Apache Flink**: höherer Infrastrukturaufwand.	**RabbitMQ**: Integration in Microservice-Architekturen. **Celery**: Erweiterung um Task-Management. **Apache Airflow**: ergänzende Orchestrierungsebene.	Open-Source-basierte Ansätze ohne proprietäre Bindung. REST- oder Tool-basierte Varianten können Governance-Risiken erhöhen.	**RabbitMQ** stellt im Kontext serviceorientierter Kommunikation die ausgewogenste Lösung dar.
Monitoring & Observability (Standard-Stack)	**Prometheus** **Grafana** **Blackbox Exporter** **Elastic Stack (ELK)** **Datadog** **Jaeger**	**Prometheus**: effizientes Zeitreihen-Monitoring. **Grafana**: Visualisierung und Dashboarding technischer Metriken. **Blackbox Exporter**: aktive Verfügbarkeitsprüfung externer Endpunkte. **Elastic Stack (ELK)**: Log-Analyse mit höherem Ressourcenbedarf. **Datadog**: SaaS-basiertes Monitoring. **Jaeger**: Distributed Tracing.	**Prometheus**: skalierbar über verteilte Architektur. **Grafana**: zustandslos betreibbar und horizontal skalierbar. **ELK**: clusterfähig. **Datadog**: Skalierung durch SaaS-Modell des Anbieters.	**Prometheus**, **Grafana**, **Blackbox Exporter**: plattformunabhängig einsetzbar. **Datadog**: cloudgebunden.	Integration in containerisierte Microservice-Architekturen. Monitoring von ETL-, Query- und API-Komponenten. **Jaeger**: Ergänzung für Tracing-Szenarien.	**Prometheus** und **Grafana**: Open Source. **Datadog**: proprietär.	Die Kombination aus **Prometheus** und **Grafana** bildet die technisch geeignete Grundlage für das Monitoring der Plattform.

Fortsetzung auf der nächsten Seite

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
API und Web	API-Schicht: **FastAPI** **Flask** **Django REST Framework** **Express.js** Web-Schicht: **Modulare WebUI** **Monolithische Webanwendungen** **BI-Tools (Tableau, Power BI)**	**FastAPI**: asynchrone Verarbeitung (ASGI), geringe Latenz bei hoher Parallelität. **Flask**: synchrones Request-Modell, eingeschränkte Performance bei hoher Last. **Django REST Framework**: umfangreiche Funktionalität, höherer Overhead. **Express.js**: hohe Performance, separates Node.js-Ökosystem. **Modulare WebUI**: leichtgewichtig, direkte Service-Kommunikation. **BI-Tools (Tableau, Power BI)**: performante Visualisierung, zusätzliche Plattformschicht.	**FastAPI**: containerisiert horizontal skalierbar. **Express.js**: ebenfalls skalierbar. **Django REST Framework**: skalierbar, jedoch schwergewichtig. **Modulare WebUI**: getrennte Services unabhängig skalierbar. **Monolithische Webanwendungen**: eingeschränkte Komponenten-Skalierung. **BI-Tools (Tableau, Power BI)**: skalierbar, plattformabhängig.	**FastAPI**: vollständig cloudneutral. **Modulare WebUI**: On-Prem und Cloud betreibbar. **Flask**, **Django REST Framework**, **Express.js**: ebenfalls cloudneutral. **BI-Tools (Tableau, Power BI)**: häufig cloudzentriert oder lizenzgebunden.	**FastAPI**: direkte Integration in Python-basierte ETL-Services, Simulationen und Monitoring. Unterstützung serviceorientierter Architektur und REST-basierter Kommunikation. **Modulare WebUI**: modulare Anbindung an unterschiedliche Backend-Services. **BI-Tools (Tableau, Power BI)**: eingeschränkte Integrationskontrolle, externe Plattformabhängigkeit.	**FastAPI**: modernes Open-Source-Framework mit aktiver Community. **Flask**: etabliertes Open-Source-Projekt. **Django REST Framework**: Open Source. **Express.js**: Open Source. **BI-Tools (Tableau, Power BI)**: proprietär, Vendor-Lock-in.	**FastAPI** erfüllt die definierten Kriterien hinsichtlich Performance, horizontaler Skalierbarkeit, Hybridfähigkeit, Integrationsfähigkeit und Anbieterunabhängigkeit am konsistentesten.
Automatisierung & CLI	**MinIO Client (mc)** **Cleanup-/Housekeeping-Service** Alternativen: **AWS CLI** **Shell-Skripte** **Web-Verwaltung**	**MinIO Client (mc)**: direkter, effizienter Zugriff auf Object Storage; geringer Overhead. **Shell-Skripte**: performant, jedoch anfällig bei komplexer Logik. **Web-Verwaltung**: zusätzlicher Interaktionsaufwand.	**Cleanup-/Housekeeping-Service**: ereignisgesteuerte und zeitbasierte Automatisierung; unabhängig skalierbar. CLI-Integration in Container möglich. **Web-Verwaltung**: nicht skalierbar.	**MinIO Client (mc)**: S3-kompatibel, cloudneutral. **AWS CLI**: primär AWS-gebunden. **Shell-Skripte**: plattformneutral.	**MinIO Client (mc)**: direkte Integration in ETL-Container und CI/CD-Pipelines. **Cleanup-/Housekeeping-Service**: eingebunden in Microservice-Architektur. **Web-Verwaltung**: keine systemische Integration.	**MinIO Client (mc)**: Open Source. **Cleanup-/Housekeeping-Service**: unter eigener Kontrolle. **AWS CLI**: cloudgebunden. **Shell-Skripte**: offen, wartungsintensiv.	Die Kombination aus **MinIO Client (mc)** und **Cleanup-/Housekeeping-Service** erfüllt die definierten Kriterien hinsichtlich Performance, Skalierbarkeit und Anbieterunabhängigkeit am konsistentesten.
Security & Authentication	**Keycloak** **Auth0** **OAuth2 Proxy** **OPA (Policy-basierte Autorisierung)** **Individuelle Authentifizierung**	**Keycloak**: tokenbasierte Authentifizierung (OAuth2 / OpenID Connect); JWT ermöglicht lokale Validierung. **OAuth2 Proxy**: delegiert Authentifizierung an externen Identity Provider; geringer Overhead. **Auth0**: performante SaaS-basierte Identity-Plattform, jedoch externe Netzwerklatenz. **OPA**: geringer Overhead durch Policy-Auswertung. **Individuelle Authentifizierung**: Performance abhängig von Eigenimplementierung.	**Keycloak**: containerisierbar und horizontal skalierbar. **Auth0**: automatische Skalierung als SaaS-Dienst. **OAuth2 Proxy**: skalierbar als leichtgewichtiger Reverse-Proxy. **OPA**: leichtgewichtig skalierbar. **Individuelle Authentifizierung**: hoher Wartungsaufwand.	**Keycloak**: vollständig cloudneutral, On-Prem und Cloud einsetzbar. **Auth0**: cloudbasierter Dienst. **OAuth2 Proxy**: abhängig vom verwendeten Identity Provider. **OPA**: cloudneutral.	**Keycloak**: Integration in **FastAPI**, **WebUI** und Backend-Services; zentrale Rollen- und Claim-Verwaltung über standardisierte Protokolle. **OAuth2 Proxy**: Integration als Reverse-Proxy vor Webdiensten möglich. **Auth0**: Integration über OAuth2 / OpenID Connect APIs. **OPA**: Policy-as-Code zur Autorisierung. **Individuelle Authentifizierung**: individuelle Integration erforderlich.	**Keycloak**: etabliertes Open-Source-Projekt. **OPA**: Apache Open Source. **OAuth2 Proxy**: Open Source. **Auth0**: proprietärer SaaS-Dienst. **Individuelle Authentifizierung**: keine Anbieterbindung, jedoch erhöhte Sicherheits- und Wartungsrisiken.	**Keycloak** erfüllt die definierten Kriterien hinsichtlich Performance, Skalierbarkeit, Hybridfähigkeit und Anbieterunabhängigkeit am konsistentesten.

Fortsetzung auf der nächsten Seite

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
Relationale Datenhaltung & Metadatenverwaltung	**PostgreSQL** **MySQL / MariaDB** **Oracle** **Microsoft SQL Server** **Amazon Aurora / Cloud SQL**	**PostgreSQL**: stabile Performance für relationale Workloads und Metadatenverwaltung; geeignet für Status-Tracking und Benchmark-Daten. **MySQL / MariaDB**: vergleichbare Eigenschaften im relationalen Umfeld. **Oracle** und **Microsoft SQL Server**: leistungsfähig für Enterprise-Workloads, jedoch mit komplexerem Betriebsmodell. **Amazon Aurora / Cloud SQL**: verwaltete Cloud-Datenbanken mit externer Infrastrukturabhängigkeit.	**PostgreSQL**: ausreichend skalierbar für Control-Plane-Daten mit moderatem Volumen. **MySQL / MariaDB**: vergleichbare Skalierbarkeit im relationalen Umfeld. **Oracle** und **Microsoft SQL Server**: hohe Skalierbarkeit im Enterprise-Bereich. **Amazon Aurora / Cloud SQL**: automatische Skalierung durch verwaltete Cloud-Dienste.	**PostgreSQL** und **MySQL / MariaDB**: flexibel in On-Premises- und Cloud-Umgebungen einsetzbar. **Oracle** und **Microsoft SQL Server**: sowohl On-Premises als auch als Cloud-Dienste verfügbar. **Amazon Aurora / Cloud SQL**: cloudnative Varianten mit Plattformbindung.	**PostgreSQL**: Integration in Microservices und ETL-Komponenten. **MySQL / MariaDB**: ähnliche Integrationsmöglichkeiten in Microservice-Architekturen. **Oracle / Microsoft SQL Server**: Integration über Enterprise-Datenbank-Ökosysteme. Klare Trennung von Control-Plane-Metadaten (**PostgreSQL**) und Data-Plane-Metadaten (**Apache Iceberg**).	**PostgreSQL**: etabliertes Open-Source-System mit breiter Community-Unterstützung. **MySQL / MariaDB**: Open-Source-Varianten; MariaDB mit stärkerer Community-Ausrichtung. **Oracle**: proprietäres Enterprise-System. **Microsoft SQL Server**: proprietäres System, eng an das Microsoft-Ökosystem gekoppelt. **Amazon Aurora / Cloud SQL**: verwaltete Cloud-Datenbanken mit Plattformabhängigkeit (AWS/GCP).	**PostgreSQL** stellt im Projektkontext die konsistente Metadatenbasis dar.
ETL-Services als Microservice-Ansatz	**ETL-Microservices** **ETL-Prozess** **Apache Airflow** **Apache NiFi** **Talend**	**ETL-Microservices**: Parallelisierung einzelner ETL-Stufen möglich; Reduktion potenzieller Engpässe. **monolithischer ETL-Prozess**: zentrale Verarbeitung mit potenziellen Bottlenecks. **Apache Airflow**: orchestriert Tasks, optimiert jedoch nicht die Datenverarbeitung selbst. **Apache NiFi / Talend**: leistungsfähig, jedoch zusätzlicher Betriebs- und Laufzeitaufwand.	**ETL-Microservices**: gezielte horizontale Skalierung pro Verarbeitungsschritt. **monolithischer ETL-Prozess**: Skalierung nur als Gesamteinheit. **Apache Airflow**: Workflow-Skalierung, jedoch keine direkte datenbezogene Skalierung. **Apache NiFi / Talend**: skalierbar, abhängig vom Betriebsmodell.	Containerisierte **ETL-Microservices**: identisch in Cloud- und On-Prem-Umgebungen betreibbar; unterstützt hybride Deployments. **Apache Airflow**: containerisierbar und sowohl in Cloud- als auch On-Prem-Umgebungen einsetzbar. **Apache NiFi / Talend**: zusätzliche Infrastruktur erforderlich. **monolithischer ETL-Prozess**: weniger flexibel in verteilten Umgebungen.	**ETL-Microservices**: Integration über ereignisbasierte Kommunikation (z. B. Messaging); klare Schnittstellen pro Service. Nutzung von **PostgreSQL** für Status- und Metadatenverwaltung sowie Persistenz im Object Storage und Registrierung in der Lakehouse-Schicht. **monolithischer ETL-Prozess**: enge Kopplung der Verarbeitungsschritte; eingeschränkte Integrationsflexibilität. **Apache Airflow**: Integration über Workflow-Orchestrierung und externe Task-Ausführung. **Apache NiFi**: integrierte Datenflussplattform mit eigener Pipeline- und Integrationslogik. **Talend**: Integration über proprietäre ETL-Frameworks und Connectoren.	**ETL-Microservices**: offene Technologien ohne Plattformbindung. **Apache Airflow**: Open-Source-Projekt mit breiter Community-Unterstützung. **Apache NiFi**: Open Source, jedoch zusätzlicher Betriebsaufwand. **Talend**: kommerziell, Lizenz- und Plattformabhängigkeit. **monolithischer ETL-Prozess**: abhängig von eingesetzten Frameworks oder Eigenimplementierung.	Der **Microservice-basierte ETL-Ansatz** erfüllt die definierten Kriterien hinsichtlich Performance, gezielter Skalierbarkeit, Hybridfähigkeit und Anbieterunabhängigkeit am konsistentesten.

Fortsetzung auf der nächsten Seite

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
Servicebasierte Query-Zugriffsschicht	**Query-Service und Query-Stats-Service** **Direkter Zugriff auf die Query-Engine (Trino)** **BI-Tool als Query-Abstraktionsschicht** **Reines Query-Logging ohne Service-Schicht**	**Dedizierter Query- und Query-Stats-Service**: geringe zusätzliche Latenz durch vorgeschaltete API-Schicht. Zentrale Steuerung von Query-Routing, Parametrisierung und Limitierung. Konsistente Abfrageausführung im Multi-Client-Betrieb. **Direkter Zugriff auf Trino**: minimale Latenz, jedoch keine zentrale Steuerung. **BI-Tool als Query Abstraktionsschicht**: Performance abhängig von Caching-Mechanismen. **Reines Query-Logging**: keine strukturierte Performance-Analyse.	**Dedizierter Query- und Query-Stats-Service**: unabhängige horizontale Skalierung der Services möglich; Entkopplung von Query-Cluster und API-Last. **Direkter Zugriff auf Trino**: Skalierung primär über den **Trino**-Cluster. **BI-Tool als Query Abstraktionsschicht**: Skalierung abhängig vom Tool- und Lizenzmodell. **Reines Query-Logging**: keine gezielte Skalierung der Analysekomponenten.	API-basierte **dedizierte Query- und Query-Stats-Services**: cloudneutral betreibbar; identische Deployment-Logik in Cloud- und On-Prem-Umgebungen. **Direkter Zugriff auf Trino**: abhängig von der Infrastruktur des Query-Clusters. **BI-Tool als Query Abstraktionsschicht**: häufig plattform- oder cloudgebunden. **Reines Query-Logging**: abhängig von eingesetzter Logging-Infrastruktur.	**Dedizierte Query- und Query-Stats-Services**: zentrale Schnittstelle für **WebUI** und Benchmarking-Komponenten. Strukturierte Speicherung von Query-Metriken und Ausführungsstatistiken. Einheitliche API statt heterogener Client-Zugriffe. Unterstützung konsistenter Authentifizierungs- und Autorisierungsprüfung. **Direkter Zugriff auf Trino**: heterogene Client-Zugriffe ohne zentrale Steuerung. **BI-Tool als Query Abstraktionsschicht**: Integration primär über das jeweilige Tool-Ökosystem. **Reines Query-Logging**: Integration ausschließlich über Logging- oder Monitoring-Systeme.	**Dedizierte Services**: offene Schnittstellen (SQL/HTTP, REST, tokenbasierte Authentifizierung). **Direkter Zugriff auf Trino**: offene Query-Engine ohne proprietäre Abstraktionsschicht. **BI-Tool als Query Abstraktionsschicht**: potenzieller Vendor-Lock-in durch proprietäre Analyseplattformen. **Reines Query-Logging**: abhängig von eingesetzten Logging- und Monitoring-Systemen.	Der **dedizierte Service-Ansatz** erfüllt die definierten Kriterien hinsichtlich Performance, gezielter Skalierbarkeit, Hybridfähigkeit und Anbieterunabhängigkeit am konsistentesten.
Monitoring-Service (projektspezifische Observability-Schicht)	**Dedizierter Monitoring-Service** **Prometheus-Exporter** **ELK-Stack** **SaaS-Monitoring-Lösungen**	**Dedizierter Monitoring-Service**: zentrale Aggregation, Korrelation und Weiterverarbeitung technischer und fachlicher Metriken aus ETL-, Query- und Simulationsdiensten. Entkopplung der Auswertungslogik von Kernservices. Asynchrone Verarbeitung und strukturierte Persistenz von Status- und Messdaten. **Prometheus-Exporter**: reine Rohdatenerfassung ohne aggregierende Logik. **ELK-Stack**: primär auf Log-Analyse fokussiert. **SaaS-Monitoring-Lösungen**: Integration über externe Monitoring-Plattformen.	**Dedizierter Monitoring-Service**: unabhängig horizontal skalierbar; Entlastung der Kernsysteme. **Prometheus-Exporter**: Skalierung über Prometheus-Server und Scraping-Architektur. **ELK-Stack**: skalierbar über Cluster-Architektur. **SaaS-Monitoring-Lösungen**: Skalierung über externe Plattforminfrastruktur.	**Dedizierter Monitoring-Service**: containerisiert und plattformunabhängig betreibbar. **Prometheus-Exporter** und **ELK-Stack**: sowohl in Cloud- als auch On-Prem-Umgebungen einsetzbar. **SaaS-Monitoring-Lösungen**: plattformgebunden.	Konsolidierung von ETL-, Query- und Simulationsmetriken. Anbindung an **Prometheus** als Datenquelle. Bereitstellung strukturierter Kennzahlen für Dashboard- und Benchmark-Komponenten. **Prometheus-Exporter**: Integration über Metrik-Endpunkte. **ELK-Stack**: Integration über Log-Forwarding. **SaaS-Lösungen**: Integration über Agenten oder APIs.	**Dedizierter Monitoring-Service**: basiert auf offenen Schnittstellen. **Prometheus-Exporter** und **ELK-Stack**: etablierte Open-Source-Lösungen. **SaaS-Monitoring-Lösungen**: potenzieller Vendor-Lock-in.	Der **dedizierte Monitoring-Service** stellt im Projektkontext die strukturell konsistenteste Lösung zur Aggregation und Auswertung projektspezifischer Metriken dar.
Dashboard- und Visualisierungsschicht	**Dashboard (SPA)** **Power BI, Tableau** **Monitoring-Dashboards (Grafana)**	**Webbasiertes Dashboard**: clientseitige Darstellung mit API-basierter Datenabfrage. Unterstützung interaktiver und teilweise echtzeitnaher Auswertungen. **BI-Plattformen (Power BI, Tableau)**: leistungsfähige Visualisierung mit zusätzlicher Plattformebene. **Grafana**: optimiert für Monitoring- und Zeitreihendaten.	**Webbasiertes Dashboard**: zustandslos und horizontal skalierbar. **BI-Plattformen**: Skalierung abhängig vom Betriebsmodell. **Grafana**: skalierbar über Dashboard-Server und Datenquellen.	**Webbasiertes Dashboard**: containerisiert und flexibel in Cloud- und On-Prem-Umgebungen betreibbar. **BI-Plattformen**: häufig plattform- oder cloudgebunden. **Grafana**: sowohl in Cloud- als auch On-Prem-Umgebungen einsetzbar.	Zentrale UI für Simulation, Monitoring und Analyse. Anbindung ausschließlich über definierte Services. **BI-Plattformen**: Integration primär über proprietäre Datenquellen-Connectoren. **Grafana**: Integration über Monitoring- und Zeitreihen-Datenquellen.	**Webbasiertes Dashboard**: offene Webtechnologien ohne Lizenzbindung. **Grafana**: Open-Source-Grundversion verfügbar. **BI-Plattformen**: proprietär.	Das **webbasierte Dashboard** bildet im Projektkontext die konsistente Präsentationsschicht der Plattform.

Fortsetzung auf der nächsten Seite

Schichten	Technologie	Performance	Skalierbarkeit	Cloud-/On-Prem-Kompatibilität	Integration in bestehende Systeme	Open-Source-Reifegrad / Vendor-Unabhängigkeit	Bewertung
Caching & Zustandsverwaltung	**Redis** **Memcached** **In-App-Caching** **Relationale Datenbank als Cache**	**Redis**: speicherbasierter Zugriff mit sehr geringer Latenz; unterstützt erweiterte Datenstrukturen. **Memcached**: speicherbasierter Key-Value-Cache mit sehr geringer Latenz. **In-App-Caching**: sehr geringe Zugriffszeiten, jedoch nur lokal pro Instanz verfügbar. **Relationale Datenbank als Cache**: zusätzlicher Transaktions-Overhead.	**Redis**: verteilbar und für Microservice-Architekturen geeignet. **Memcached**: horizontal skalierbar über verteilte Cache-Knoten. **In-App-Caching**: keine globale Konsistenz zwischen Serviceinstanzen. **Relationale Datenbank als Cache**: nicht für hochfrequente Cache-Zugriffe optimiert.	**Redis** und **Memcached**: containerisiert sowohl in Cloud- als auch On-Prem-Umgebungen betreibbar. **In-App-Caching**: unabhängig von Infrastruktur, jedoch an einzelne Serviceinstanzen gebunden. **Relationale Datenbank als Cache**: abhängig von Datenbankinfrastruktur.	**Redis**: zentrale Cache-Schicht für kurzlebige Zustände und serviceübergreifende Koordination. **Memcached**: vergleichbare Integration als verteilter Key-Value-Cache. **In-App-Caching**: nur innerhalb einzelner Serviceinstanzen nutzbar. **Relationale Datenbank als Cache**: Integration über Datenbankzugriffe.	**Redis** und **Memcached**: etablierte Open-Source-Lösungen. **In-App-Caching**: abhängig vom verwendeten Framework. **Relationale Datenbanken**: teilweise proprietär.	**Redis** bildet im Projektkontext die geeignete verteilte Cache-Schicht der Plattform.
Data-Access-Service & Benchmark-Service	**Dedizierte Zugriff- und Benchmark-Services.** **Direkter Datenbankzugriff.** **Monolithisches Backend.** **Externe Benchmark-Tools.**	**Dedizierte Services**: serverseitige Filterung und Aggregation; kontrollierte Zugriffspfade und reduzierte Datenübertragung. **Direkter Datenbankzugriff**: potenziell geringe Latenz, jedoch fehlende zentrale Abfragekontrolle. **Monolithisches Backend**: Performance abhängig von zentraler Implementierung. **Externe Benchmark-Tools**: zusätzliche Laufzeit- und Integrationsschicht.	**Dedizierte Services**: unabhängige Skalierung von Zugriff und Benchmarking. **Direkter Datenbankzugriff**: Skalierung primär über Datenbankressourcen. **Monolithisches Backend**: Skalierung nur als Gesamtsystem. **Externe Benchmark-Tools**: abhängig vom Tool- und Betriebsmodell.	**Dedizierte Services**: containerisierbar und cloudneutral. **Direkter Datenbankzugriff**: abhängig von Infrastruktur und Sicherheitsrichtlinien. **Monolithisches Backend**: weniger flexibel bei modularen Deployments. **Externe Benchmark-Tools**: zusätzliche Plattformebene.	**Dedizierte Services**: zentrale Schnittstelle für **WebUI**, Monitoring und Simulation; klare Trennung von Fachlogik und Persistenz. **Direkter Datenbankzugriff**: heterogene Client-Zugriffe ohne einheitliche API. **Monolithisches Backend**: enge Kopplung von Logik und Datenzugriff. **Externe Benchmark-Tools**: Integration meist über Connectoren oder Exporte.	**Dedizierte Services**: offene Schnittstellen ohne Plattformbindung. **Direkter Zugriff** / **Monolith**: Abhängigkeit vom verwendeten Stack. **Externe Benchmark-Tools**: potenzieller Vendor-Lock-in.	Der **Service-Ansatz** ermöglicht kontrollierte Zugriffspfade und reproduzierbares Benchmarking.
Simulation- und ML-Services	**Spezialisierte Simulatoren je Architekturtyp.** **ML-Service zur Prognose.** **Monolithischer Simulator.** **Externe Simulationstools.**	Parallele Ausführung spezialisierter Simulatoren ermöglicht unabhängige Verarbeitung verschiedener Architekturszenarien. Reduzierung von Laufzeiten durch Entkopplung der Modelle. **Monolithischer Simulator**: eingeschränkte Parallelisierung und potenzielle Ressourcenkonflikte. **Externe Simulationstools**: Performance abhängig von Plattform- und Infrastrukturmodell.	Separate Skalierung je Simulator-Service entsprechend Modellkomplexität. **ML-Service**: unabhängig skalierbar für Trainings- und Prognosephasen. **Monolithischer Simulator**: Skalierung nur als Gesamtsystem. **Externe Tools**: Skalierung abhängig vom jeweiligen Tool.	Containerisierte Simulator-Services cloudneutral in Cloud- und On-Prem-Umgebungen betreibbar. Simulation von hybriden Szenarien ohne Infrastrukturbindung. **Externe Simulationstools**: häufig plattform- oder lizenzgebunden.	Anbindung an **Benchmark-Service** und **Monitoring-Service**. Vergleich von simulierten und real gemessenen Werten. Bereitstellung strukturierter Ergebnisse für Dashboard und Analyse. **ML-Service**: Nutzung historischer Benchmark-Daten zur Prognose.	Eigenentwickelte Services auf offenen Technologien. Transparente Modelllogik ohne proprietäre Bindung. **Externe Simulationstools**: potenzieller Vendor-Lock-in.	Der **Multi-Simulator-Ansatz** mit ergänzendem **ML-Service** erfüllt die Kriterien hinsichtlich Performance, Skalierbarkeit, Hybridfähigkeit und Integrationsfähigkeit am konsistentesten.

Quelle: Eigene Darstellung (Latex)
Insbesondere auf Grundlage offizieller Dokumentationen und Herstellerinformationen sowie Whitepapers und wissenschaftlicher Literatur. (Vgl. [T2026]; [T2026]; [P2026]; [S2026]; [A2026]; [S2026]; [A2026]; [M2025]; [V2025]; [I2025]; [A2026]; [C2026]; [P2026]; [J2026]; [D2026]; [R2026]; [P2026]; [P2026]; [P2026]; [D2026]; [D2026]; [O2026]; [N2026]; [M2026]; [M2026]; [M2026]; [S2026]; [H2012]; [S2014]; [R2026]; [O2026]; [O2026]; [O2026]; [N2020]; [P2026]; [O2026]; [M2026]; [M2026]; [O2025]; [O2026]; [O2026]; [M2026]; [A2026]; [G2026]; [M2025]; [G2026]; [G2026]; [R2026]; [M2026]; [M2026]; [M2026]; [A2026]; [A2026]; [A2026]; [Q2026]; [FM2011]; [A2021]; [T2021]; [J2022]; [A2026]; [A2026]; [A2026]).

A.2 Systemkomponenten und Schnittstellen

Systemkomponenten und Schnittstellen der containerisierten PoC-Umgebung
Quelle: Eigene Darstellung (Latex)

Komponente	Zweck	Technische Rolle	Wichtige Schnittstellen
etl-inject	Überwacht eingehende Rohdaten, erkennt neue Dateien, lädt sie nach PostgreSQL und triggert Folgeprozesse	Data Ingestion, Tabellenanlage, Status-Tracking	Dateisystem, PostgreSQL, RabbitMQ
etl-export	Exportiert Daten aus PostgreSQL nach MinIO (Parquet), partitioniert große Tabellen	ETL-Export, Datenpartitionierung, Status-Update	PostgreSQL, MinIO, RabbitMQ
etl-register	Registriert neue Parquet-Daten im Iceberg-Katalog für analytische Abfragen	Metadaten-Management, Lakehouse-Integration	Iceberg REST, RabbitMQ
RabbitMQ	Vermittelt Nachrichten zwischen ETL-Services, entkoppelt Verarbeitungsschritte	Messaging, Trigger, Service-Entkopplung	AMQP, REST (Management UI)
PostgreSQL	Speichert Rohdaten, Metadaten und Statusinformationen	Relationale Datenbank, Persistenz, Metadatenhaltung	SQL, ETL-Services
MinIO	Speichert exportierte Daten im Parquet-Format	Object Storage, S3-kompatibel, Lakehouse-Backend	S3-API, ETL-Export, Trino, Spark
Trino	Führt analytische SQL-Abfragen auf Lakehouse-Daten aus	SQL Query Engine, Analytics	Iceberg-Connector, MinIO, WebUI
Apache Iceberg	Katalogisiert und verwaltet Tabellen im Lakehouse	Table Catalog, Metadaten-Management	REST-API, Trino, etl-register
Spark	Batch-Processing und Analyse großer Datenmengen	Compute Engine, Datenverarbeitung	S3-API, Iceberg, MinIO
Benchmark-Service	Erfasst Systemmetriken, berechnet ML-Prognosen und simuliert Kosten	Monitoring, ML, Simulation, REST-API	REST, WebUI, Simulation-Services
Simulation-Services	Simulieren verschiedene Betriebsmodelle (z. B. AWS, On-Prem, Hybrid)	Kosten-/Laufzeitsimulation, Szenarienanalyse	REST, Benchmark-Service
Redis	Temporärer Speicher für Zwischenergebnisse	In-Memory Cache, schnelle Datenablage	Key-Value-API, Backend-Services
WebUI	Benutzeroberfläche für Abfragen, Monitoring und Steuerung	Frontend, Visualisierung, User-Interaktion	REST-API, WebSocket, Trino, Benchmark
Prometheus	Sammelt Metriken von Services	Monitoring, Metrik-Sammlung	HTTP (Metrics Endpoints)
Grafana	Visualisiert Systemmetriken und Dashboards	Monitoring, Dashboarding	Prometheus, WebUI

A.3 KI-Verzeichnis

Bei der Erstellung dieser Arbeit wurde das KI-Tool ChatGPT (OpenAI) in begrenztem Umfang unterstützend genutzt.

Die Nutzung erfolgte vor allem als Gesprächspartner zur Reflexion von Ideen und zur gedanklichen Strukturierung einzelner Themenbereiche. Teilweise wurde das Tool auch zur sprachlichen Präzisierung einzelner Textstellen verwendet. Darüber hinaus wurde ChatGPT gelegentlich als Ideengeber bei Fragen zur Erstellung und Formatierung von LaTeX-Tabellen sowie bei Querverweisen und allgemeinen Formatierungsfragen eingesetzt.

Vorschläge aus der KI-gestützten Unterstützung wurden lediglich als Reflexionshilfe genutzt und nicht ungeprüft übernommen. Alle Inhalte dieser Arbeit wurden eigenständig formuliert, überprüft und wissenschaftlich eingeordnet.

Die Nutzung des Tools erfolgte verantwortungsvoll im Sinne der Empfehlungen der Hochschule für Technik und Wirtschaft Berlin.

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe, und dass die Arbeit in gleicher oder ähnlicher Form in noch keiner anderen Prüfung vorgelegen hat. Sämtliche wörtlichen oder sinngemäßen Übernahmen und Zitate, sowie alle Abschnitte, die mithilfe von KI-basierten Tools entworfen, verfasst und/oder bearbeitet wurden, sind kenntlich gemacht und nachgewiesen.

Angaben zur Verwendung KI-basierter Hilfsmittel

Im Anhang meiner Arbeit habe ich sämtliche KI-basierte Hilfsmittel angegeben. Diese sind mit Produktnamen in einem KI-Verzeichnis ausgewiesen. Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung der Prüfer / die Prüferin explizit schriftlich ausgeschlossen hat. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate. Ich versichere zudem, dass in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt.

Ort: Berlin

Datum: 17.03.2026

Unterschrift:

