Design of an Open-Source Data Lakehouse Architecture for Software Development Analytics

MASTER THESIS

Dominic Rouven Fischer

Submitted on 30 April 2025

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Julian Hirsch, M. Sc.
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 30 April 2025

License

Erlangen, 30 April 2025

Abstract

Software engineering involves the use of many tools that inherently generate valuable data. Analyzing this data is challenging because it must be retrieved from various data sources and enriched to provide analytical insights. While specific problems within the software development process have been extensively studied, comparatively less research has focused on building a scalable platform to support software development analytics. This thesis explores the applicability of a modern data lakehouse architecture for software development analytics.

Following a design science approach, a modular, scalable, and extensible data lakehouse architecture for software development analytics, based on Apache Spark, Delta Lake, and S3, was developed. The solution builds upon an existing system and extends it into a complete data lakehouse following the Medallion architecture, with a structured Data Vault-based Silver layer and a Star Schema-based Gold layer. Although the developed prototype focused on implementing a single-tenant solution, the thesis discusses in detail how the system can be extended to multi-tenancy. The prototype enhances the existing system of a small development team by adding a Data Vault-based Silver layer, orchestration via Apache Airflow, and an S3-compatible object store. The prototype's applicability was successfully verified using development data from GitHub, GitLab, and Jira. The developed software architecture was evaluated through a structured walkthrough with an experienced software architect, and documented using the industry-proven arc42 architecture documentation template.

Contents

List of Figures

List of Tables

Acronyms

ACID Atomicity, Consistency, Isolation, Durability

ADR Architecture Decision Record

AI Artificial Intelligence

API Application Programming Interface

AWS Amazon Web Services

BI Business Intelligence

CaaS Container as a Service

CD Continuous Delivery

CI Continuous Integration

CLI Command Line Interface

CPU Central Processing Unit

CRUD Create, Read, Update, Delete

DAG Directed Acyclic Graph

DBMS Database Management System

DevOps Development and Operations

ETL Extract Transform Load

GPU Graphics Processing Unit

HDFS Hadoop Distributed File System

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

I/O Input/Output

JAR Java Archive

JSON JavaScript Object Notation

ML Machine Learning

MQ Message Queue

ODBC Open Database Connectivity

OLAP Online Analytical Processing

ORC Optimized Row Columnar

PaaS Platform as a Service

REST Representational State Transfer

SOAP Simple Object Access Protocol

SQL Structured Query Language

UI User Interface

UUID Universally Unique Identifier

VM Virtual Machine

XML Extensible Markup Language

YAML Yet Another Markup Language

3NF Third Normal Form

1 Introduction

1.1 Motivation

Software engineering generates large amounts of valuable data through the use of various tools for different aspects, such as repositories, ticketing systems, testing frameworks, and communication channels. This data can be structured, semi-structured or unstructured depending on its source. Correlating and analyzing this heterogeneous data can yield significant insights into the software development process.

However, aggregating data across these various data sources is technically challenging. Software development data needs to be ingested regularly, stored and correlated to gain valuable insights. The traditional approach of using a data warehouse can be challenging given the presence of semi-structured and unstructured data. On the other hand, storing the data in a data lake lacks data quality and user-friendliness, considering the schema-on-read nature of data lakes. To leverage the advantages of both approaches, the data lakehouse architecture (Armbrust et al., 2021) has seen increased adoption, combining the flexibility and scalability of a data lake to handle various types of data with the structured, ACID-compliant processing capabilities of traditional data warehouses.

Implementing a data lakehouse appears to be a promising approach for aggregating software development data from various sources. Yet, most approaches regarding implementing data lakehouses focus on single-tenant solutions that are used within a single organization. Multi-tenancy increases complexity as flexibility becomes essential and processing data of multiple tenants on a shared platform raises security and privacy considerations. Conversely, stakeholders can benefit from multi-tenancy through reduced infrastructure costs and improved maintainability of the application.

This thesis aims to design a data lakehouse architecture for software development analytics, addressing the need for a centralized, scalable analytics platform to gain insights into software development data. In particular, the solution is designed for single-tenancy, but architectural considerations regarding an exten-

sion to a multi-tenant system are discussed. The solution's applicability is shown by a prototypical implementation focusing on a single-tenant deployment and demonstrated using data from repositories and ticketing systems. To facilitate its adoption and support further research, the solution is designed exclusively using open-source components.

1.2 Research Questions

After outlining the motivation, four research questions are defined. This thesis aims to design a solution to aggregate and analyze software development data of various data sources. Therefore, the first research question focuses on identifying the core components required for such a system.

1. What are the core logical components required for a system that aggregates and analyzes software development data?

Before a software architecture is created, it is necessary to identify the challenges associated with these logical components. This includes addressing functionalities such as data ingestion, data integration and configuration handling.

2. Which challenges do the logical components of such a system need to address?

After having identified core logical components and having discussed their complexities, the aim is to develop a single-tenant-architecture, which should be a foundation for subsequent enhancements to multi-tenancy.

3. How would a suitable single-tenant data lakehouse architecture for aggregating and analyzing development data look like?

Building on the developed single-tenant data lakehouse architecture, the next step is to extend its functionality to multi-tenancy as this extension inherits additional complexity that must be tackled.

4. How can the single-tenant architecture be enhanced to support the aggregation and analysis of development data from multiple tenants?

1.3 Thesis Structure

As a method of choice to address the defined research questions, the design science research method for information systems as described by Peffers et al. (2007) is adopted. Accordingly, this thesis is structured as follows:

Chapter 2 covers the problem definition. It introduces the fundamentals of gaining knowledge from development data, discusses the inherent challenges in data

ingestion and integration, and examines the concepts of data warehousing, data lakes, and data lakehouses in detail. The chapter concludes with a presentation of the system's business context and an overview of related work.

Chapter 3 defines the objectives of the solution given the problem definition. After working out the artifact's functional requirements, quality requirements and applicable architecture constraints are presented.

Chapter 4 comprises the design of the proposed solution. Firstly, the solution's technical context is introduced. Secondly, key design decisions that shape the solution's software architecture are discussed using Architecture Decision Record (ADR)s. Next, building upon the previous sections, the derived solution design is presented from multiple viewpoints.

Chapter 5 targets the implementation of the designed solution of chapter 4. It is stated, which technologies are used, how the prototype is implemented and how the application can be configured.

Chapter 6 demonstrates that the developed prototype works as expected by presenting its execution using software development data.

Chapter 7 covers the evaluation of the developed artifact that was conducted as a structured walkthrough.

Chapter 8 concludes this thesis by summarizing main contributions, discussing the limitations of the proposed solution and providing recommendations for future research.

2 Problem Definition

Initially, this chapter introduces the fundamentals of gaining knowledge from software development data and which challenges in data ingestion and integration are inherited. Subsequently, the concepts of data warehousing, data lakes and data lakehouses are discussed in detail. The chapter concludes with a presentation of the business context of the system that should be designed and an overview of related work.

2.1 Software Development Analytics

Artificial Intelligence (AI) driven software development has been adopted in the industry and has been a huge field of interest in research in recent years. Areas of application for AI in the software development process range from code generation and bug detection to project management and testing. See Ajiga et al. (2024) for a recent review on the adoption of AI in software development in the industry.

Buse and Zimmermann (2010) coined the term *software analytics* for approaches where insights into the software development process are gained by development data analysis to support decision-making. The term *software development analytics* is also commonly used in the literature, as highlighted by another relevant paper (R. P. L. Buse & T. Zimmermann, 2012). Hassan and Xie (2010) proposed the term *software intelligence* for this domain. Even though the term software intelligence has not been able to establish itself, it is notable because it refers to the relationship between software development analytics and business intelligence. Business Intelligence (BI) can be defined as systems that use data gathering, data storage, knowledge management and analysis to gain business-sensitive information on a company's or competitor's business (Negash & Gray, 2008). BI hereby assists in planning and decision-making and enhances the timeliness and quality of inputs within the decision-making process.

The analysis of software development data is of interest for many actors. These stakeholders have different interests. Data and analysis needs of managers and developers were extensively surveyed by R. P. L. Buse and T. Zimmermann

(2012). Franch (2022) pointed out the relationships of relevant stakeholders with software analytics tools. Relevant stakeholders are (Franch, 2022; R. P. L. Buse & T. Zimmermann, 2012):

- **Software Developers:** Developers have a strong interest in software quality and in improving their software development processes. Many metrics for software quality have been introduced for diverse aspects, such as software complexity, testability, or maintainability. Development tools for static program analysis facilitate software engineering work by analyzing source code in the development phase.
- **(Project) Managers:** For good decision-making processes, managers need project insights. Managers are interested in trends of key-performance indicators, alerting of sudden changes that affect a project's performance and simulation and forecasting of how a project will continue to keep the project ongoing and in time. A case study by Rique et al. (2023) examines the use of data-driven decision-making in one software organization, identifying key use cases, influencing factors, and organizational challenges.
- **Requirements Engineers:** Requirements engineers are interested in identifying system misbehavior, such as detecting missing functional or quality requirements.
- **Domain Experts:** Domain experts design a quality model that links metrics gathered from the development data to indicators presented to stakeholders, such as defining the components that make up dashboard indicators.
- **Data Scientists:** Data scientists configure analytics software by connecting relevant data sources and implementing metrics and indicators that were determined by the domain experts.
- **Other Stakeholders:** Most studies deal with the analysis of development data with regard to developer needs and management decisions. However, analyzing development data is also relevant for other areas such as accounting. Possible use cases include demand planning, working time recording or transfer pricing.

Development data is distributed and can be collected from diverse sources. In software development analytics, not only source code repositories are of interest, but also other data sources, such as ticketing systems, documentation and communication channels. Software development analytics tasks differ in scope. Development data of various sources can be analyzed separately, but important information is often gained by correlation. Therefore, software development analytics can be applied on various scales. A small scope is the correlation of development data of a single entity (e.g., a single developer). Broader analytics tasks

involve multiple entities, such as the analysis of all projects within a team, of an organization or across organizational boundaries.

A prerequisite for analyzing data from multiple entities is that the data to be analyzed is available to the analytics platform. *Open source* grants the right to freely copy, distribute and modify a program and view its source code (Perens et al., 1999). Hence, open source development facilitates software development analytics and is a beneficial basis for the analysis of cross-organizational development processes. *Inner source* practices support software development analytics within organizations. According to Capraro and Riehle (2016), inner source describes the usage of open source development practices within organizations. Organizations adopt open source methodologies, such as open communication, open development artifacts, the creation of communities, reusability and participatory development of software. However, the created software product is often still proprietary licensed. (Capraro & Riehle, 2016)

2.2 Challenges in Data Ingestion and Data Integration

As Negash and Gray (2008) stated, BI systems involve data gathering, data storage, knowledge management and analysis. Creating a system to analyze software development data requires ingesting the data from various sources into a common data storage. Subsequently, the data needs to be integrated if data correlation should be achieved, as data integration eases further data analysis.

2.2.1 Data Ingestion

Software development data is heterogeneous. A solution to analyze software development data must support data ingestion to retrieve data from multiple external data sources. Some data sources produce data that needs to be ingested in batches, such as the entire historical data of a newly added data source (e.g., historical Git issues). In other cases, data to be ingested is of a streaming type that requires fast real-time processing (e.g., integrating the latest Git commits to allow live-tracking of a project's success).

Marz (2011) described in a popular blog post the batch/real-time architecture that was later coined lambda architecture. In a lambda architecture, the processing of batch and streaming data is separated. A batch processing system is responsible for computing all but the most current data of the last few hours. A real-time processing system is responsible for the most current, newly available data. Query results in lambda architectures are calculated by querying both batch and real-time views and combining the results. (Marz, 2011)

Kreps (2014) found as major disadvantage of the Lambda architecture that both batch and real-time processing systems must produce the same results, leading to high complexity and maintainability issues. Therefore, the Kappa architecture was introduced. In contrast, in a Kappa architecture, all data types are ingested as a stream. If batch processing becomes necessary, it is derived by replaying events that have already been processed. This necessitates keeping events as long as a reprocessing may be required, e.g., for 30 days by defining a retention time on them. (Kreps, 2014)

While both approaches seem feasible to process software development data, separate endpoints for batch and streaming data seem beneficial for this project due to its simplicity. Query processing of real-time data can be implemented if required, but this complexity is not further discussed in this thesis as it is often sufficient to analyze historical data. This thesis only addresses the analysis of historical data, neglecting real-time data analysis.

2.2.2 Data Integration

Analyzing development data also has challenges in terms of data integration. Multiple data sources have different data models. Even if data sources share entities with a common meaning, these entities often have different data models. For example, the entity `commit` has a different data model in GitHub and GitLab. Data models may differ in the existence or absence of tables or columns. Also, data types, constraints and the way in which relationships are defined can differ. To correlate data from multiple data sources, data of each data source must be transformed from its original data model to a unified data model. This involves multiple transformation steps, such as converting data types like dates and mapping keys. During this whole process, transformation steps must maintain data quality.

Furthermore, data from multiple data sources can also differ in its semantic meaning. The semantic model of data defines what the data means (for its stakeholders). In some cases, the data model does not contain all information that is required to correctly interpret the data. Consider two tables about costs from two different data sources. Without knowing the currency in which the data is presented in each of these two tables, it is possible to transform the data and combine it into a unified model, but it is not possible to interpret the correlated data correctly.

2.3 From Data Warehouses to Data Lakehouses

Analyzing software development data requires a system that supports data ingestion and integration and is able to provide the curated data. This section dives

deeper into data ingestion and integration approaches, explaining the foundations that are required for a system that is able to flexibly handle different types of data.

2.3.1 Data Warehouses

In the 1990s, data warehouses came up as foundational technology to support further analytical tasks. Data warehouses provide an integrated view on data, including summaries on the data from various data sources, historical data and metadata (Inmon, 1996). Typically, a data warehouse consists of multiple layers of data. The first stage contains detailed operational data, used and modified by applications on a day-to-day basis, often by high-performance transaction processing. Data warehousing is usually conducted with structured data. This data from various sources is integrated into a common data layer, called the atomic or data warehouse level. This data warehouse layer contains historical data from multiple applications, but only parts of the data are relevant to different stakeholders. Therefore, abstractions for several stakeholders are needed. A data mart is a departmental abstraction of the data in a data warehouse. Data marts often consist of original data and derived data, such as indicators. (Inmon, 2005)

Operational data from different applications has several schemes. To pass from the operational layer to the data warehouse layer, the data of each data source needs to be integrated into the data warehouse by Extract Transform Load (ETL) processes. This is referred to as schema-on-write approach. (Inmon, 2005)

Maintaining a data warehouse can be challenging. Scaling requires both computing infrastructure and storage to be sufficiently provisioned. As this can be a time-consuming task and data warehouses do not decrease in size when they are underutilized, they tend to be over-provisioned and thereby costly. In recent years, cloud data warehouses came up, where infrastructure and storage are handled by the cloud computing provider. This allows for flexible scaling up and down based on the data volumes to handle, making cost benefits possible. They also offer massively parallel processing (MPP) to achieve better performance for complex analytical queries. (Rehman et al., 2018)

2.3.2 Data Lakes

The emergence of big data brings further challenges, as data volumes become to large and complex to be processed by conventional applications and data warehousing technology in a reasonable amount of time. Data volumes foster the need for batch processing, stream processing or hybrid approaches for efficient processing. Big data is characterized by the *5V's* as described by Ishwarappa and Anuradha (2015):

- **Volume:** Companies encounter difficulties in processing the large volumes of data with their limited resources, so a significant part remains untouched.
- **Velocity:** Data is generated with increasing speed, making fast processing of large data volumes necessary.
- **Variety:** Data is not always structured and easily storable in a relational database by a pre-defined schema. Semi- or unstructured data increases the complexity of storage, data integration and analysis.
- **Veracity:** Dirty or corrupted entries are likely to appear in large data volumes. The accuracy of the analysis is influenced by the veracity of the source data.
- **Value:** Storing big data is costly and only valuable if it can be turned into actionable insights, so that businesses see a return on their investment.

Data lakes came up as technology to address the increasing demands of big data (Miloslavskaya & Tolstoy, 2016). The term *data lake* was first mentioned in a blog post by Dixon (2010) as a solution to deal with large amounts of homogeneous data, including semi-structured and unstructured data, by dispensing with data integration when writing. Data warehouses typically store structured data using schemas in relational databases, whereas data lakes store raw data in cost-effective file-based systems, often utilizing open formats like Apache Parquet (The Apache Software Foundation, 2024d) or Apache ORC (The Apache Software Foundation, 2024c). (Armbrust et al., 2021) This eases storing the data into the data lake, but at the cost of harder analysis for readers. Readers need to analyze the data structure, face data quality issues and gain information by applying a scheme on it. This is considered as schema-on-read approach.

Two common data lake architectures were proposed in the literature. In *zone architectures*, data is distributed to multiple zones based on the extent to which the data was processed. For data ingestion purposes, data is stored in its raw format into the raw zone first. Subsequently, it is further processed and stored additionally in other zones, such as zones with a common data format, zones for cleansed data, or zones like data marts. So, the data in data lakes with zone architectures can be read in its raw format or in a pre-processed state. (Giebler et al., 2019)

In *pond architectures*, data is distributed to different ponds, but unlike in zone architectures, it is only accessible from one pond at a time. At first, it is stored in a raw data pond. Then, the data is further transformed to other data ponds based on its attributes, e.g., textual data flows from the raw pond to textual data ponds. Unused data and data that is not able to be processed to other ponds remains in the raw data pond. When data is no longer required, it can be archived in an archive data pond. Pond architectures facilitate data analysis as data can

be accessed in an already pre-processed state. However, this convenience comes at the expense of the loss of the original raw data format. (Giebler et al., 2019)

Data lakes are a good choice for scalable handling of large data volumes. However, the lack of a schema hardens data discovery. Furthermore, data lakes do not fully support traditional Database Management System (DBMS) features such as Atomicity, Consistency, Isolation, Durability (ACID) transaction processing, auditing, data versioning and caching. If insufficient metadata is provided, data lakes tend to become *data swamps* (Harby & Zulkernine, 2022).

2.3.3 Data Lakehouses

To leverage the advantages of both architectures Armbrust et al. (2021) proposed data lakehouses as an architecture of choice for big data management. Data lakehouses combine the flexibility of data lakes to handle large amounts of data with the structured, clean and integrated view on data of data warehouses. A data lakehouse can be defined as "a data management system based on low-cost and directly-accessible storage that also provides traditional analytical DBMS management and performance features such as ACID transactions, data versioning, auditing, indexing, caching, and query optimization." (Armbrust et al., 2021)

In the following, key aspects of implementing a data lakehouse are discussed.

Data storage

Data can be stored in distributed storage systems. One possibility is the use of a cloud object storage such as Amazon S3 (Amazon Web Services, 2024) and storing the data in an open file format (e.g., Apache Parquet or Apache ORC). Therefore, multiple nodes from different computing environments can access the same storage data and perform separate computations concurrently (e.g., perform application-based queries on a few Central Processing Unit (CPU) nodes while also calculating a complex Machine Learning (ML) task on a Graphics Processing Unit (GPU) cluster). For this reason, data lakehouses are also well-suited for cloud computing environments. Another option is the implementation of a data lakehouse based on an on-premises storage systems such as Hadoop Distributed File System (HDFS). (Armbrust et al., 2021) There exist two commonly adopted strategies for updating data in data lakehouse storage systems. In data lakehouses that adopt the copy-on-write strategy, files with records to be updated are eagerly rewritten. This supports a higher read performance, but at the cost of slower writes. The merge-on-read strategy writes out updates to separate files and defers the merge to query time resulting in faster writes but slower reads. (Paras et al., 2023)

Metadata layer:

To support traditional DBMS operations like transaction processing, data lakehouses require a metadata layer on top of the data storage layer. There exist multiple storage frameworks based on log-structured tables to implement the metadata layer of data lakehouses. The most popular implementations include Delta Lake (Armbrust et al., 2020), Apache Hudi (The Apache Software Foundation, 2024a) and Apache Iceberg (The Apache Software Foundation, 2024b). A comparison of them is given in Paras et al. (2023).

Delta Lake was initially developed by Databricks, a large data lakehouse platform provider. In Delta Lake, metadata information about objects that are part of the Delta table is stored in a write-ahead log to support ACID operations. Hereby, the write-ahead log is stored next to the Parquet files containing the objects in the same cloud object storage. This possibility for transactional processing allows data lakehouses to support data management capabilities like UPSERT, DELETE, MERGE, audit logging and data layout optimization as in traditional data warehouses, while still supporting time travel and data lake features such as efficient streaming and schema evolution. (Armbrust et al., 2020)

Performance

Data lakehouse implementations have different trade-offs, especially based on the technology used for log-structured tables and as data storage. Camacho-Rodríguez et al. (2024) developed a benchmarking approach and metrics to assess the performance of data lakehouses, which are well-suited for horizontal scalability in cloud computing environments. However, data has to be transferred over the network from the storage layer to computing nodes for every single computation, which requires huge network bandwidth and can become a bottleneck for calculation performance. Query performance challenges of queries against data lakehouses in cloud computing environments are discussed in Weintraub (2023). To tackle Structured Query Language (SQL) performance problems, Armbrust et al. (2021) proposes holding statistics for each Parquet file for data skipping, caching files from the object storage in memory or on disk of processing nodes and data layout optimizations such as ordering the records.

Support for Advanced Analytics

Advanced data analytics methods such as ML require large amounts of data, but ML frameworks face performance problems in retrieving data from traditional data warehousing architectures. As Lewis et al. (2025) point out in their recent work on Input/Output (I/O) behavior, due to the weak compatibility of data warehouses and ML frameworks, data is often loaded via Representational State Transfer (REST), Simple Object Access Protocol (SOAP), or Open Database

Connectivity (ODBC) interfaces with high latency, which has negative performance impacts for large datasets.

In contrast, data lakehouse architectures and data lakes offer the advantage that data is stored file-based in distributed storage systems. ML frameworks support open columnar file formats such as Apache Parquet, which are commonly used in data lakehouses. Providing ML frameworks with direct file-based access to the data has a positive impact on performance. On the other hand, data lakehouses still offer data management features that are comparable to functionality of traditional data warehouses. (Armbrust et al., 2021)

2.3.4 Comparison

In a recent survey, Harby and Zulkernine (2025) compared common open-source implementations of a data warehouse, data lake and data lakehouse. They found that data lakehouses have higher response times for simple queries, but outperform data warehouses and data lakes in complex queries with real-world data. On top of that, in contrast to the other two architectures, data lakehouses allow both time travel and schema evolution to provide accurate results for multiple points in time. (Harby & Zulkernine, 2025)

To summarize the previous sections, a comparison of key differences between the three architectures is shown in Table 2.1.

Characteristic	**Traditional Data Warehouse**	**Data Lake**	**Data Lakehouse**
Type of data	structured	structured, semi-structured and unstructured	structured, semi-structured and unstructured
Usability	easy data analysis for readers	complex data analysis as schema is missing, support for ML and advanced analytics	easy data analysis for readers, support for ML and advanced analytics
Data storage	relational or structured and columnar-based	fast, low-cost file storage based on open-formats	fast, low-cost file storage based on open-formats
Data integration	schema-on-write	schema-on-read	schema-on-read and schema-on-write possible (schema implementation after data storage, but before user access)
Transactions	ACID-compliant	no ACID-compliant transaction processing	ACID-compliant
Data quality	high	low	high
Scaling	vertical	horizontal	horizontal

Table 2.1: Comparison of key characteristics of data warehouses, data lakes and data lakehouses based on Harby and Zulkernine (2025)

2.3.5 Two-Tier Architecture

Before the emergence of data lakehouses, two-tier architectures were commonly conducted in the industry. Firstly, large data volumes are handled in a data lake, followed by a data warehouse downstream. So, data is first ingested into a data lake with ETL processes, followed by other ETL processes for integrating it into the data warehouse. Utilizing data warehouses for aggregated data has the advantage that the strict data integrity rules of relational databases are applied on data that needs high data quality. Furthermore, traditional BI tools that typically rely on data warehouse implementations can easily be integrated. However, Armbrust et al. (2021) state that these architectures typically have reliability and data staleness issues, a limited support for advanced analytical tasks such as ML and their total costs are high due to redundancy. (Armbrust et al., 2021)

2.4 Business Context

This thesis aims to design a reliable and scalable solution that is able to analyze software development from various sources, including repositories, ticketing systems and communication channels. It explores the adoption of the modern data lakehouse architecture to software development analytics. This section introduces the topic further by describing the scope of the system whose software architecture is discussed in this paper and explaining its business context.

The main purpose of the application is to gain knowledge from development data of various sources by correlation. The system is designed to be used by a small team that especially applies software development analytics in the following domains (Hirsch et al., 2025):

- **Productivity Measurement:** By analyzing software development data, which is produced by tools used in the software development process, the productivity of teams should be measured. This allows project managers to monitor the health of on-going software development projects.
- **Transfer Pricing:** If software is developed in different organizational units across tax boundaries (e.g., in multiple countries), source code must be legally priced and fairly taxed. With software development analytics, the transfer prices of software engineering work of multi-national organizations should be determined.
- **Impact Assessment:** Source code generates (monetary) value. The monetary impact of the work of software development teams should be assessed by correlating the created value with lines of code produced by teams.

Figure 2.1 depicts the business context of the system. The application is connected to various external data sources that provide software development data.

This data is ingested and integrated into the data lakehouse, which serves as the central analytical storage that unifies software development data from multiple data sources. Data ingestion and data integration processes can be partially configured. Configuration options include modifying data ingestion intervals, adjusting specific data integration rules and managing credentials that are required to access external data sources.

Different user roles interact with the system. Developers and project managers use the solution mainly to access calculated analytical insights such as metrics by the usage of provided dashboards or reports. In contrast, data scientists are interested in working with the data and controlling the process in which analytical insights are derived. Administrators are responsible for setting up access to external data sources by configuring routes and credentials.

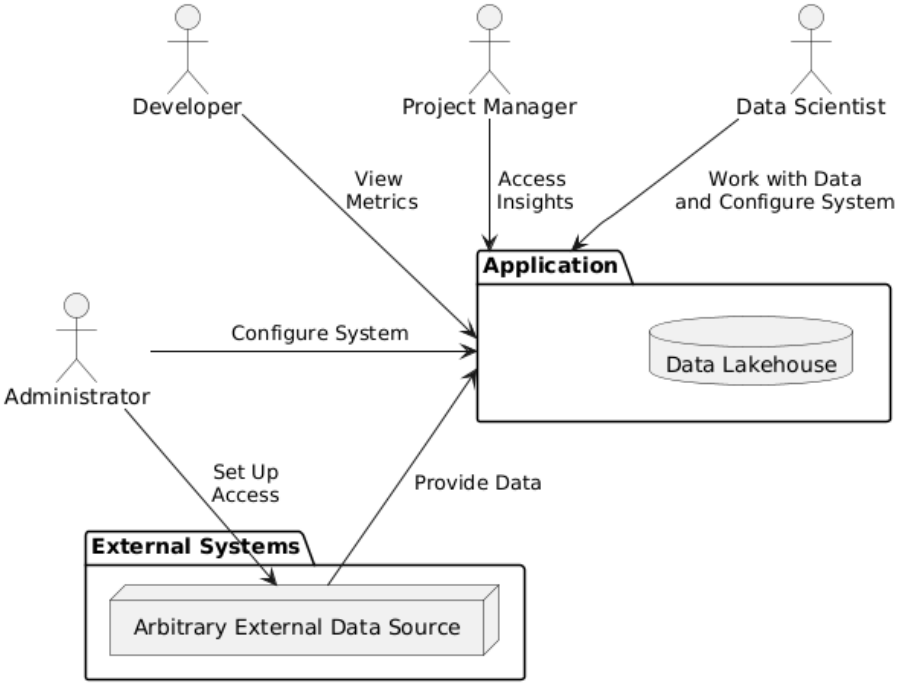


Figure 2.1: Business Context of the System

2.5 Related Work

Most research in the field of software development analytics focuses on investigating specific problems within the software development process. In 2014, a study by Begel and Zimmermann (2014) from Microsoft identified questions that data science applied to the domain of software development can try to answer. A later study conducted by Huijgens et al. (2020) at ING in 2020 investigated which of these questions also apply for other organizations. Wang et al. (2023)

examined the use of ML in analyzing software engineering activities through a systematic literature review of papers published between 2009 and 2020. They investigated to which software engineering tasks traditional ML or Deep Learning approaches have been applied to, in order to identify trends in the use of ML in this field. ML techniques have been applied to software engineering activity in various domains, including requirements engineering (e.g., requirements detection and classification), software implementation (e.g., code generation) and project management (e.g., software cost estimation). (Wang et al., 2023) For example, a study by Karna et al. (2019) applied a data mining approach based on data from 27 software projects to estimate the effort of software development activity.

While investigating specific problems within the software development process has gained much attention in the scientific community, comparatively less research has focused on building a platform to support software development analytics. An early approach was conducted by Czerwonka et al. (2013), who described the development of CODEMINE, a software development analytics platform at Microsoft. Scheibel et al. (2024) aimed to integrate visual software analytics directly into the GitHub platform by using GitHub Actions and the GitHub Application Programming Interface (API) to analyze and visualize insights about each commit at the data source.

The Atlassian Data Lake is a platform based on Amazon Web Services (AWS) S3 and Databricks that integrates various sources, primarily from the Atlassian ecosystem, including products such as Jira and Confluence. On top of that, it supports connections to a collection of other external data sources, such as commonly used relational databases, Snowflake and Databricks. However, the Atlassian Data Lake builds upon its own data silo and is a commercial, closed-source solution whose source code is not publicly available. (Atlassian, 2025; Friedman et al., 2023)

GrimoireLab was designed as a free, open-source toolset that aims to support common tasks involved in software development analytics, such as collecting, curating and visualizing software development data. It currently supports a wide range of commonly used data sources, including GitHub, GitLab, Jira, Confluence, Telegram and others. It also provides an API to retrieve the data, which was ingested to the Elasticsearch of GrimoireLab, to use the data within other applications. Data integration in Grimoirelab involves standardizing fields into a unified data model and optionally identifying different contributor identities across multiple datasets and merging these identities. (Dueñas et al., 2021) A study by Gonzalez-Barahona et al. (2022) demonstrates how GrimoireLab can be used to gain insights into the health of software development projects by integrating diverse data sources and creating dashboards and reports. They identified different stakeholders and created customized metrics and visualizations tailored to each stakeholder group (Gonzalez-Barahona et al., 2022).

3 Objective Definition

The following chapter defines the objectives of the solution given the problem definition. After working out the artifact's functional requirements, its quality requirements and the limitations under which the architecture must be designed are discussed.

3.1 Functional Requirements

This section defines the key functional requirements that the solution must fulfill to satisfy stakeholder needs. Section 2.4 already introduced the business context of the solution. The system has various stakeholders, including project managers, data scientists, developers and administrators. These users are either interested in working with the system or in retrieving the calculated analytical results.

The solution that is designed in this thesis builds upon an existing system by a small development team (Hirsch et al., 2025). This existing system successfully implements a functional data lake for software development data, realized using Python, Apache Spark and Delta Lake. It features data ingestion pipelines in the form of Python scripts to retrieve data from various external data sources, including GitLab, GitHub and Jira. The collected data is stored in files locally on the machine running the Spark jobs. The already implemented data integration steps involve multiple cleaning steps of the ingested data. Based on the curated datasets, the system is already capable of generating valuable analytical insights.

However, the focus of this thesis is to design a target software architecture based on the data lakehouse architecture that can be used for software development analytics in arbitrary domains and extends this foundational system. The paper discusses the software architecture of such a system on various scales, so that it can be used in small analytical projects, but can also be extended to handle large data volumes and solve analytical tasks in large organizations.

3.1.1 Main Functional Requirements

Such a general system to process and analyze software development data must provide the following main functionality:

FR1: Data Ingestion

The system must be able to ingest data from multiple data sources, including source code repositories (GitHub, GitLab), ticketing systems (Jira), documentation sites (Confluence, Sharepoint, Websites) and communication channels (E-Mail, Microsoft Teams, Slack). Multiple organizations also work with different systems and may require integrating their own custom data sources. Consequently, the solution must be designed with the flexibility to integrate new data sources. It must be able to handle multiple serialization formats and it should be also adaptable to work with mechanisms like batch processing or data streaming. On top of that, the ingestion functionality of each data source must include handling of difficulties such as rate limits or temporary unavailability of data sources.

FR2: Extract, Transform and Normalize Data

To support further analysis steps raw data of various sources needs to be cleaned, normalized and transformed into a unified structure. This involves converting serialization formats, unifying source specific formats such as dates and dropping not relevant information (e.g., tables) for each data source.

FR3: Correlate Data from Different Sources

The system must be able to identify related entities, that share a common semantic meaning, across different data sources and link these entities together to provide advanced insights through data correlation. This allows to perform analytics on the data of a person from multiple data sources.

FR4: Gain Business-Oriented Insights

Furthermore, stakeholders require to gain useful insights into the software development process. Different stakeholders are interested in different insights of multiple domains, such as work-time calculations, productivity metrics, and financial impact assessments. The system must be able to perform multiple data correlations separately and offer the flexibility to implement supplementary correlations.

FR5: Present Insights to Stakeholders

The system must offer the possibility to present the gained analytical insights to stakeholders. Therefore, it needs to provide the gained analytical insights through an easily usable interface. This API can be used by a custom user interface to visualize results with dashboards or to create reports. This API also allows organizations to integrate the results of the analytical platform into other used BI tools.

3.1.2 Multi-Tenancy Support

Some organizations require an On-premises deployment of the analytical system in their infrastructure. In this scenario, the application is operated within or by the organization. Other organizations may want to benefit of a cloud solution without the burden of operating the system. Such a cloud solution has cost benefits with multi-tenancy support. While the original system of the development team is based on a single-tenant solution that can only analyze software development for one tenant, this paper also discusses how such a system can be designed to be multi-tenant. Multi-tenancy introduces significant additional complexity. As a single-tenant deployment for a specific tenant within an organization is often sufficient, multi-tenancy is not seen as a strict requirement but rather as a promising extension of the solution.

Multi-Tenancy support especially necessitates the following additional functionality:

FR6: Data Isolation per Tenant

The system works with sensitive data of multiple tenants that needs to be processed securely. Data of different tenants needs to be isolated from the data of other tenants and must not be combined or correlated.

FR7: Security and Access Control

The solution must ensure that a tenant can only access data which belongs to the tenant. A tenant must not be able to access data of other tenants.

3.2 Quality Requirements

After having discussed key functional requirements, this section summarizes major quality goals that are relevant for the solution. Quality goals are divided according to the nine quality characteristics, which define a model of the quality of a product, as defined in the norm ISO/IEC 25010:2023(en) (International Organization for Standardization, 2023).

QG1: Functional Suitability

The system must provide realistic insights into the software development processes. Therefore, the system must ensure correct data ingestion, integration and analytics processes to gather meaningful insights. The solution should allow to possibly revert incorrect inserts (data ingestion).

QG2: Performance Efficiency

The end user should be able to see business insights fast even if this requires pre-calculation. The system should be horizontally scalable and able to handle large volumes of data. Data ingestion and integration should work in a reasonable amount of time and can be scheduled.

QG3: Reliability

Reports should provide consistency. Business insights to the end user on dashboards can be eventually consistent. It is sufficient to calculate resource-intensive evaluations (e.g., once a day) and display the pre-calculation to users.

QG4: Compatibility

New data sources can be added easily and the schema of existing data sources can be evolved.

QG5: Flexibility

Tenants can configure which data sources and to some extent which parts of a data source are to be used. The system should be able to run in cloud computing environments, but also OnPremises installations on servers should be possible.

QG6: Security

Credentials and sensitive data should be processed securely. Data of a tenant should be protected from the access of other tenants.

QG7: Maintainability

The system should be easily maintainable with standardized API generation and containerized deployments.

QG8: Interaction Capability

The user interface should be easily usable and self-explanatory.

QG9: Safety

It is unlikely that the solution poses a risk to humans or the environment. Therefore, the quality characteristic safety is not a primary concern.

3.3 Architecture Constraints

During the design of the solution, the following architecture constraints have to be acknowledged.

The development team aims to extend their existing solution by incorporating the architecture designed in this paper. This introduces the organizational constraint that the solution should be implementable by a team of a small size. Furthermore, the source code has to remain in a private repository and cannot be publicly released.

Since the application has already been partially developed, technical constraints also apply. The existing solution was developed using Python, Apache Spark and Delta Lake. These technologies should also be employed to realize the data lakehouse for software development analytics discussed in this thesis. To facilitate the solution's adoption, all third-party software used must be available under a permissive license. This approach also helps to avoid costs and mitigate potential legal problems.

The solution should be developed independently of the operating system. It must be executable on all commonly used operating systems (Linux, Windows, macOS). An installation on a single machine should be supported to simplify software development. However, to prepare it to be executed in cloud-native environments, it should especially be able to be deployed in containers on Linux/Ubuntu basis. To achieve containerized deployments Docker Compose (Docker Inc., 2025) is used to run all components of the application across multiple containers.

4 Solution Design

After having defined functional and quality requirements as well as architecture constraints for the solution, this chapter presents the design of the proposed system. It first describes the technical context of the solution and introduces high-level system layers. The introduction of the technical scope of the artifact serves as a basis for key architectural decisions, which are discussed in section 4.2. Finally, section 4.3 derives the overall solution strategy based on these key design decisions of the previous section.

4.1 Technical Context

Section 2.4 has already introduced the business context of the system. This section provides an overview of the system's technical context, including an introduction to its main components.

According to the principle of separation of concerns, the solution follows a structured and modular approach to support replaceability, maintainability and scalability. The system consists of multiple layers:

- **Frontend:** A frontend visualizes insights into the development process and allows users to interact with the system.
- **Backend Services Layer:** A backend services layer is responsible for providing access to analytical insights (Analytics Service) and functionality for configuration management (Configuration Service).
- **Data Management Layer:** This layer consists of Spark jobs responsible for ingestion and integration of software development data into the data lakehouse.
- **Storage Layer:** Stores the ingested and processed data within a data lakehouse.

Figure 4.1 comprises the technical context of the system. The Frontend communicates with the Analytics Service to retrieve analytical insights gained from

software development data. Configuration functionality to customize the data lakehouse is provided by the Configuration Service. Apache Spark jobs of the data management layer are responsible for data integration and ingest data from external systems. This technical context provides an architectural overview that serves as a basis for key design decisions that are detailed in the following section.

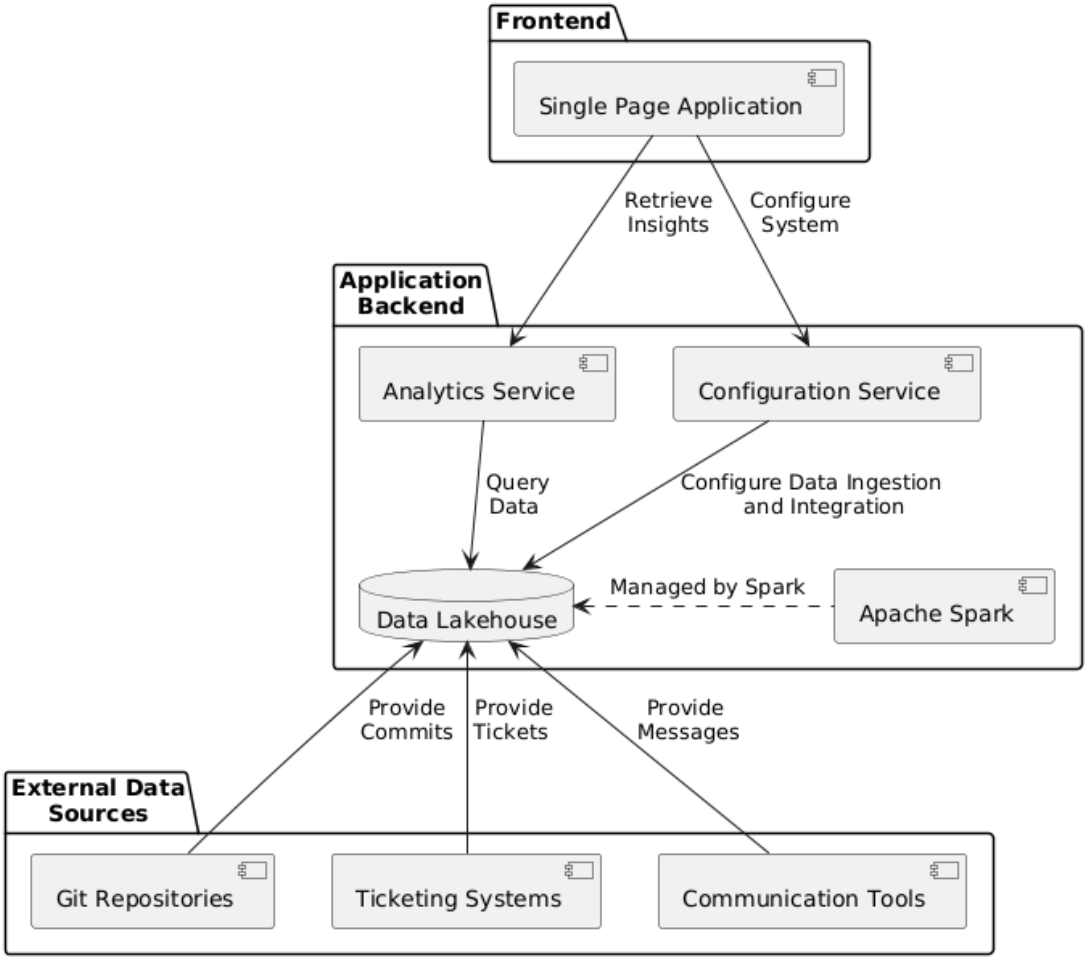


Figure 4.1: Technical Context of the System

The Medallion Architecture:

The data management layer of the data lakehouse requires a data structure. A commonly used approach to tackle the complexity of combining multiple data sources, is to improve data structure and quality incrementally as data flows through multiple layers. As previously mentioned in 2.3.2, data in data lakes and data lakehouses is often logically divided based on the extend to which data has been processed, where each layer has a certain objective. There exists various data design patterns (zone architectures) that propose different logical layers.

The so called Medallion architecture is a three-layered zone architecture which is often adopted in data lakehouses (Databricks Blog, 2022). According to the Medallion architecture a data lakehouse is logically divided into three separate data layers. At first, data is ingested and stored without further integration into the so called Bronze layer which contains raw data. Metadata such as identifiers or the loading time is optionally added. Secondly data is cleansed, normalized and merged to the Silver layer providing a broader view on the data of different domains. The Gold layer offers the highest level of data quality. Data in the Gold layer is typically either stored in a traditional Star Schema or in another consumer-friendly format and is provided consumption-ready. A schematic view of the Medallion architecture is depicted in figure 4.2. (Haelen & Davis, 2024)

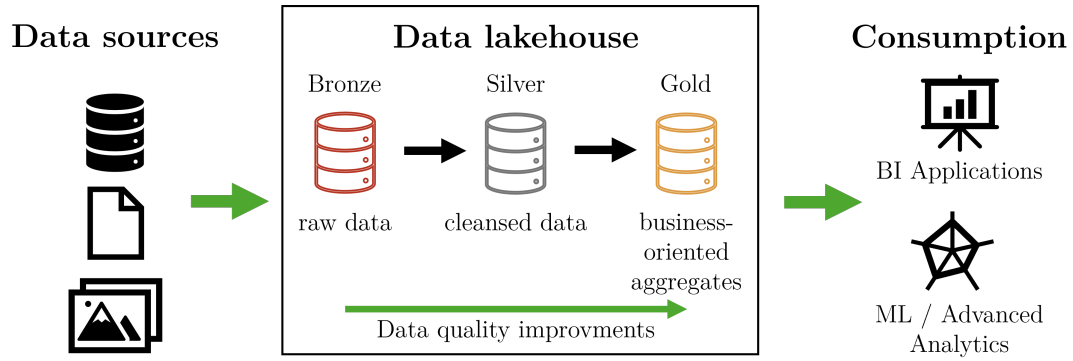


Figure 4.2: The Medallion Architecture (based on (Haelen & Davis, 2024))

The Medallion architecture solves the problem that some users are only interested in aggregates and do not need to view cleansed data from all data sources. The Gold layer offers a simple user-friendly view on these aggregates. The three-layered Medallion architecture as foundation of the data lakehouse is adopted in this solution, because a dedicated layer with a user-friendly data model is seen as beneficial.

4.2 Key Design Decisions

After having introduced key requirements in chapter 3 and explaining the technical context of the system in the previous section, this section dives deeper into the actual solution design. Key design decisions of the software architecture are discussed that have a major impact on the internal structure of the solution and how the system behaves.

In the industry often ADR's are adopted as structuring format to systematically document key architecture decisions. ADR's are composed of a title, the context in which the decision was embedded, the decision itself, the status of the decision (e.g., accepted, outdated) and the resulting consequences (Nygard, 2011). ADR's

are also used to document key architecture decisions in this thesis. As deviation from standard ADR's, the decision status is neglected in this thesis as it is not further relevant. For each decision, considered alternatives are presented and their key advantages and disadvantages are summarized in tabular form.

4.2.1 Physical Data Storage

Context

Data Lakehouse Storage Frameworks store information in specialized data storage formats, such as Apache Parquet or Optimized Row Columnar (ORC). Consequently, data lakehouses require a scalable storage system in which these files can be stored physically. The modern data lakehouse often uses a low-cost object store such as an Amazon S3-compatible system as physical storage as suggested by Armbrust et al. (2021). But also OnPremises deployments based on HDFS are possible. This architecture decision discusses the impact of the choice of the physical data storage and compares the two storage systems.

Alternatives

Cloud object stores: Object stores hold objects that can be characterized as logical collection of bytes including some metadata describing the object, methods for accessing the object and security policies for access control. Objects can be seen as a combination of files and blocks, where files represent a higher-level storage abstraction, including security and access control, and metadata. However, file server contention has a negative impact on performance. Blocks provide fast and scalable access to shared data, but offer insufficient security without a file server for access authorization. (Mesnier et al., 2003) Popular cloud object stores include Amazon S3 (Amazon Web Services, 2024) and Azure Blob Storage (Microsoft Azure, 2025). As the Amazon S3 interface is widely supported by many tools and libraries, even Google's Cloud Storage Extensible Markup Language (XML) API offers compatibility to S3 API-using tools partially (Google Cloud, 2025).

Most cloud object storage implementations are realized as key-value stores with no cross-key consistency guarantees. This makes it difficult to obtain fast, mutable table storage and thereby hard to achieve data warehousing functionality. To realize data lakehouses in cloud object stores data can be stored in file formats such as Apache Parquet and ORC. But still, achieving high-performance for large selective queries can be hard. File formats such as Apache Parquet contain footers with min/max statistics. These footers allow skipping Parquet files without reading the entire file for better read performance. However, each object/Parquet file needs to be retrieved from the object store with latency between client and object store provider, what has a major negative impact on performance.

This issue is addressed by data lakehouse storage frameworks such as Delta Lake. Delta Lakes core idea is to store metadata in a write-ahead log in the cloud object store. This metadata includes the information which objects are part of which table, so that not every file needs to be accessed over the network, addressing the latency problem. (Armbrust et al., 2020)

The most important advantages and disadvantages of using a cloud-object store are discussed in table 4.1. A detailed review of the benefits of cloud object stores can be found in the work of Zagan and Danubianu (2021).

Advantages	**Disadvantages**
• Widely adopted cloud-native solution that is supported by many libraries including Delta Lake. • In terms of cost, S3 has cost benefits in comparison to HDFS and especially a relational database system (Jamal et al., 2021). • In contrast to local deployments where additional hardware would have to be provisioned, cloud object stores can be easily expanded as the cloud object store provider manages this complexity. From a user point of view, storage capacity in cloud object stores is unlimited and scales on demand. (Zagan & Danubianu, 2021) • Cloud object stores offer data security mechanisms according to the shared responsibility security model between users and the cloud provider (Zagan & Danubianu, 2021). • Many cloud object store solutions adopt the S3 interface and are S3-compatible. Some can also be deployed locally and are available as ready-to-run Docker Images.	• Efficient usage of cloud object stores requires using a storage framework that is optimized for cloud object store usage such as Delta Lake. So the decision for a physical data storage has a major impact on other parts of the architectural design. • Data is stored physically at another location and not at the machine where the execution takes places. So query performance is negatively impacted as data has to be transmitted over the network with latency.

Table 4.1: Advantages and Disadvantages of Choosing a Cloud Object Store as Physical Storage of the Data Lakehouse

Hadoop Distributed File System (HDFS): HDFS (Shvachko et al., 2010; The Apache Software Foundation, 2025a) is the file system component of Hadoop (The Apache Software Foundation, 2025a) and offers the functionality of a fault-tolerant distributed file system that aims to efficiently handle large datasets with high throughput access to data (The Apache Software Foundation, 2022). Data in Hadoop can be partitioned across many hosts. Operations on the data of multiple hosts can be performed in parallel, while performance advantages are tried to be achieved with data locality. Computations are tried to be executed by the node that also holds the data on which computations are performed. This reduces necessary network bandwidth, latency and allows to process large datasets with increased throughput. (Shvachko et al., 2010)

HDFS follows a master/slave architecture. Metadata is stored on a dedicated server, the so called `NameNode`, that is responsible for file system namespace operations such as file opening and closing. `DataNodes` hold the application data and perform read and write requests. Data is replicated to multiple DataNodes to achieve fault-tolerance. (The Apache Software Foundation, 2022)

Advantages and Disadvantages of the usage of HDFS are given in table 4.2.

Advantages	**Disadvantages**
• Performance advantages due to the use of data locality. Jamal et al. (2021) compared the performance of S3, HDFS and rational database system (MySQL) storage for real-time big-data applications running with a Spark cluster on AWS in terms of latency, network load and cost. They found that the best latency performance for a mixture of analytical and complex queries can be achieved with Spark and HDFS. For this reason Jamal et al. (2021) suggest using an HDFS-based architecture for real-time big-data applications. • Highly scalable system, that can be distributed across many hosts.	• HDFS is less cost-effective in comparison to S3. Databricks (2017) claim that S3 storage is ten times cheaper compared to HDFS, but with HDFS a six times higher read throughput per node can be achieved due to data locality. Accordingly, Databricks (2017) claims that S3 is two times better than HDFS in performance per dollar for data lakehouses. The study of Jamal et al. (2021) also sees cost-benefits for S3, but they are significantly lower than Databricks (2017) claims. • Operating HDFS clusters is rather complex, especially in comparison to S3 deployments.

Table 4.2: Advantages and Disadvantages of Choosing HDFS as Physical Storage of the Data Lakehouse

Decision

While a S3-based deployment seems to be more simple and offers a good trade-off between performance and cost, a HDFS deployment is favorable if there are strict-latency and performance requirements. It is also noteworthy that not every cloud object store is compatible with every Data Lakehouse Storage Framework, so compatibility has to be checked.

The solution adopts an S3-based approach, due to cost-benefits, the simplicity of this approach, simple local dockerized deployments and because Delta Lake is designed to be used with S3. For the development phase of the solution, a dockerized deployment is seen as beneficial in contrast to having to maintain machines with HDFS installations.

Consequences

- **Technical Impact:** The adoption of an S3-compatible object store enables cloud-native deployments. Either an S3 solution of a cloud provider or an on-premises installation on a server or cluster can be used. Most of the S3 compatible object store solutions also offer Kubernetes based or dockerized deployments. Furthermore, an architectural dependency on a storage framework is introduced. The used storage framework must support S3, which Delta Lake does. However, storage frameworks such as Delta Lake require that the S3 compatible object store fully support the S3 API for correct transactional processing.
- **Organizational Impact:** Local dockerized deployments of the storage solution can ease the development process as an already configured infrastructure setup can be provided. On top of that, infrastructure management is simplified for teams with limited Development and Operations (DevOps) resources.
- **Operational Impact:** An S3-compatible deployment at a cloud provider benefits flexible, cost-effective scaling of storage. Storage is able to scale or shrink on demand. However, the usage of S3 requires a reliable network connection between S3 and Spark. In distributed setups this connection has to be monitored. S3 also introduces more latency in comparison to HDFS, but this is partly mitigated by Delta Lake.

4.2.2 Query Processing

Context

To present analytical insights to end users data lakehouses typically must support two types of queries:

- **Fast Queries:** Queries that often operate on pre-aggregated insights such as metrics. Metrics can be pre-calculated and often do not need to include the latest data from every data source. On top of that, there is usually some hot, frequently accessed data that requires to be accessible instantly to be used by e.g., BI applications. Query runtime is critical to allow applications like dashboards with which the user is able to interact.
- **Complex Queries:** Long-running asynchronous queries that support advanced correlations including multiple data sources or that target to analyze historical data. These queries may require batch and distributed query processing.

While complex long-running queries are typically handled using technologies like Apache Spark, there are two common architectural approaches to implement fast queries as discussed in Serra (2021). This subsection discusses the usage of both architectural approaches and of an hybrid approach for query processing in the data lakehouse.

Alternatives

Three main alternatives were considered for handling fast and complex queries in the data lakehouse for software development data.

All Queries on the Data Lakehouse: Armbrust et al. (2021) and Databricks suggest also holding the hot frequently accessed data in the data lakehouse. The Gold layer of the medallion architecture is responsible for presenting the data to the outer world. This can even include mimicking data marts at the Gold layer that directly address departmental insight needs. However, these data mart like constructs exist in this architectural approach only logically, there is no physical separation of the data into a separate database. To meet performance requirements, also materialized views can be used at the Gold layer. Queries are executed entirely on the lakehouse storage layer. For complex queries Spark can be used. Fast queries can also be executed using Spark. However, for performance reasons it can be beneficial to directly access the Parquet files or using another engine than Spark at the Gold layer. This architectural approach to also hold Gold layer inside the data lakehouse is often promoted (especially by Databricks) and some papers argue that the term data lakehouse architecture primarily refers to this concept Armbrust et al. (2021). Table 4.3 summarizes advantages and disadvantages of this approach:

Advantages	Disadvantages
• Simple architecture without the complexity to maintain a dedicated storage for fast queries for BI-applications. • No synchronization between different storage technologies required (Delta Lake/Parquet vs. relational database). • Prevents data staleness issues. • Complex analytical queries can be efficiently executed leveraging Spark and distributed computing.	• Higher latency for real-time queries and less efficient for frequent queries on hot data. • The Gold layer of the Medallion architecture requires performance optimizations such as indexing (e.g., by date and tenantId).

Table 4.3: Advantages and Disadvantages of Executing All Queries on the Data Lakehouse

All queries on a Separate Relational Database: In contrast, others favor introducing an additional SQL layer to support high-performance frequent data access with reduced complexity for users. For example, Microsoft uses this approach in Azure Synapse Analytics (Microsoft Learn, 2024). Implementing a cloud-native SQL layer on top of the data lakehouse seems out of scope for the software architecture discussed in this thesis, as the expected load does not justify this complexity. Therefore, this approach is referred to by loading the required data to a dedicated database with high read performance in this thesis. This approach involves executing queries exclusively against this database, e.g., a relational database such as a PostgreSQL that works as a data mart or a relational database that mirrors the Gold layer of the Medallion architecture.

Introducing a relational database downstream of the data lakehouse for a subset of the data comes at the cost of additional complexity and requires data synchronization. Still, the advantage of this approach is that metadata and data are both accessible in a relational database. This eases data usage for BI applications. Serra (2021) points out that understanding the metadata of a relational database may seem to be easier than the metadata model of data lakehouses that is subject to less strict rules than a model of a relational database. This is especially a concern as lots of consumers are familiar with relational databases and SQL, but less with Data Lake Metadata frameworks. (Serra, 2021)

The advantages and disadvantages of the usage of a relational database for both fast and complex queries are given in table 4.4.

Advantages	Disadvantages
• Efficient and low-latency handling of fast queries. • Data scientists and developers can use SQL, a query language with which they are often familiar, to access data Serra (2021). • Easy integration of data into other external systems like BI tools such as PowerBI.	• Inefficient for complex analytical asynchronous queries. This approach is less scalable in comparison to Spark based query processing and lacks native distributed query processing capabilities. • Less cost-effective as data is duplicated. • Additional complexity to maintain and synchronize a dedicated storage.

Table 4.4: Advantages and Disadvantages of Executing All Queries on a Separate Relational Database

Hybrid Query Architecture: To leverage the advantages of both architectural approaches also a hybrid query architecture can be considered. This involves routing fast queries to a dedicated data mart (e.g., PostgreSQL), while complex queries are processed via the data lakehouse using Spark. Table 4.5 summarizes the advantages and disadvantages of a hybrid query architecture.

Advantages	Disadvantages
• Optimized for both fast frequent and complex analytical queries. • Efficient separation of queries against pre-aggregated frequently accessed data and infrequent complex queries. • Only hot data is duplicated.	• Additional complexity in routing queries to the appropriate engine. • Synchronization between the data mart and data lakehouse is required.

Table 4.5: Advantages and Disadvantages of a Hybrid Query Architecture

Decision

While a hybrid architecture using a separate relational database for fast queries was initially considered, a walkthrough of the software architecture with an experienced software architect led to the conclusion that this additional complexity should only be introduced if strictly required.

A dedicated relational database for fast queries introduces additional complexity into the architecture. It is dependent on the actual use case if this complexity is really necessary or if a well designed Gold layer is sufficient to met the desired performance requirements. For that reason, in the solution an approach is adapted, where all queries are executed against the data lakehouse. If strict performance requirements cannot be met with a Gold layer in the data lakehouse, the solution should be extended with separate fast and complex query processing. This dedicated layer for fast query processing is modeled optionally and involves the mirroring of pre-aggregated Gold layer data that is often accessed to data marts based on business requirements.

Consequences

- **Technical Impact:** The system avoids the additional complexity of introducing a separate database for fast query processing. It relies entirely on Spark-based query processing against the data lakehouse or if necessary direct Parquet-based data access. A dedicated building block, the Analytics Service, is required that handles secure data access and authorization to Gold layer data. The Analytics Service does not need to differentiate between fast and complex query processing, which eases the buildings blocks implementation. However, if query performance becomes a bottleneck the Analytics Service may need to be extended by separate fast and complex query processing. Beforehand, query optimization techniques should be applied to the Gold layer, such as pre-aggregations, materialized views, and other query optimization techniques.
- **Organizational Impact:** Users interact with the system through interfaces provided by the Analytics Service. The Analytics Service provides an interface that provides secure access to Gold layer data. This interface can be accessed directly by users or used as a basis for dashboards. The way users like developers retrieve data from the data lakehouse is different from the traditional SQL usage with which they are familiar. Documentation and knowledge transfer must ensure that users understand the differences in data access and the data model of the data lakehouse.
- **Operational Impact:** System complexity is reduced, because there is no need to provision or monitor a separate relational database. Keeping the data in the data lakehouse also has the advantage that no synchronization pipelines are required between the Gold layer and external systems which reduces maintenance costs. All data remains centralized within the lakehouse. This further reduces data duplication, consistency or data staleness issues and has a positive impact on costs. Observability can focus on the performance of Spark jobs.

4.2.3 Scheduling and Orchestration

Context

The data lakehouse requires multiple pipelines, including pipelines for ingesting various data sources and pipelines for integrating data between different layers of the data lakehouse. These pipelines usually consist of a data reader to retrieve the data, some transformation steps and a data writer that saves the results of a transformation step back into the targeted layer of the data lakehouse.

Some pipelines are dependent on other pipelines. Often pipelines need to be started after other pipelines have been finished. For example, consider a metric that is calculated based on a combination of data from two data sources that should be regularly updated in an interval of one day. Updating the metric requires two ingestion jobs to be triggered every day, one for each data source. These ingestion jobs can run in parallel, but subsequently an integration job to integrate the data of both data sources can only be started once both ingestion jobs have terminated. Pipelines often form the building blocks of workflows. Workflows are related activities or tasks that must be executed in relation to each other to achieve a common goal or retrieve a particular result (Belcastro et al., 2017). These chains of tasks are often modeled and also referred to as Directed Acyclic Graph (DAG)s. This complexity necessitates orchestration and scheduling functionality.

The data lakehouse requires scheduling and orchestration of data ingestion, integration, and transformation jobs. These steps differ between the single-tenant and multi-tenant solution. In single-tenant mode, a single tenant has control over the entire system and has access rights to all connected data sources. Data ingestion and transformation schedules can be completely controlled by the tenant. In both single-tenant and multi-tenant solution, data ingestion and integration jobs share the same infrastructure. A workflow orchestration platform must ensure that resources are allocated equally to all jobs running in the data lakehouse. However, this is especially important in the multi-tenant mode as multiple tenants run jobs in the data lakehouse concurrently. Workload needs to be fairly distributed as these tenants may have conflicting scheduling requirements. Furthermore, a tenant's data must be protected from the unauthorized access of other tenants. A tenant must only be able to access the data, which he owns, and only manage workflows of his concern.

It is unlikely that tenants require to alter the orchestration logic of jobs running in the data lakehouse. However, they may have different scheduling requirements of workflows. In multi-tenant mode, the scheduling and orchestration solution must therefore offer the possibility to adapt schedules, e.g., of data ingestion jobs.

Alternatives

Two different solution designs and a hybrid approach for scheduling and orchestration of jobs in the data lakehouse were considered, which will be discussed in the following:

Workflow automation platform for the single-tenant and multi-tenant solution: There exist tools on the market to schedule and orchestrate jobs running on Spark that can be used as central workflow orchestrators. Apache Airflow (The Apache Software Foundation, 2025c), that serves as an industry-proven open-source solution with a user-friendly interface, is one of the most commonly adopted solutions. A more modern, cloud-native workflow automation platform is Flyte (Union.ai, 2025). However, the bundling of the source code to the orchestration platform solution seems to be tighter in Flyte. In terms of a workflow automation platform tool Apache Airflow is favored in this thesis due to its simplicity. Business logic can be implemented separately from orchestration logic and has only to be called in the DAG workflow. With such a modularized implementation design, orchestration logic can be implemented independently of business logic. On top of that, Apache Airflow is able to run with both Docker and Kubernetes.

However, Apache Airflow and other tools lack real multi-tenancy support. As Amazon Web Services (2025) states in their docs, tenants could create DAGs that are able to change user privileges of Apache Airflow and perform operations on the metadatabase of Apache Airflow. For this reason, in a multi-tenant setting, tenants must not be granted access to Apache Airflow as it would be possible for them to extend their user privileges. But this contradicts the requirement that tenants must be able to change schedules of jobs. Therefore, Airflow is not a viable solution for the multi-tenancy solution.

Table 4.6 states the main advantages and disadvantages of the use of a central workflow orchestration platform for a single-tenant and multi-tenant data lakehouse:

Advantages	Disadvantages
• Existing industry-proven solution that offers advanced features for scheduling and orchestration. • Airflow offers a user-friendly interface. • Supports basic observability features. • Workflows are defined as DAGs and can be written in Python. • Scalable as load can be distributed across multiple workers.	• Solutions such as Apache Airflow lack support for strict tenant-level isolation. Their use is considered too insecure for the multi-tenant implementation in this thesis. • The problem of fair resource sharing is not completely solved. Sufficient workers must be provisioned. • Introduces an architectural and infrastructure dependency to a workflow automation platform. • Scheduling configuration is handled in Apache Airflow, which makes it harder to centralize all required configurations, including management of ingestion intervals.

Table 4.6: Advantages and Disadvantages of Using Apache Airflow for Orchestration

Event-driven scheduling and orchestration: Scheduling and orchestration can also be realized by a separate building block, the Configuration Service. This building block holds scheduling configurations and publishes job events to a Message Queue (MQ), e.g., Apache Kafka, if jobs need to run. Jobs consume job events from this MQ and inform the scheduling service about their completion by publishing a completion event to the MQ once business logic execution has been finished. The Configuration Service also provides security and access control to the scheduling configuration.

In the multi-tenant scenario, job events contain an identifier that identifies to which tenant the job event belongs. Spark workers, which execute jobs, call the Configuration Service with this tenant identifier to retrieve a tenants configuration and perform their job based on this configuration. An extension, where jobs consume completion events of other jobs, is possible.

The custom event-driven scheduling and orchestration approach for the data lakehouse offers the following advantages and disadvantages as discussed in table 4.7:

Advantages	Disadvantages
• Modern event-driven architecture that offers high scalability. • Secure processing in a multi-tenant scenario. • The Configuration Service, a dedicated building block, handles security and access control and provides access to configuration data.	• Unnecessary complexity for single-tenant setups. • Requires custom orchestration and scheduling logic, which is difficult to implement and increases system complexity. • Lacks advanced features such as a user-friendly interface and cancellation of jobs. • Observability features must be implemented. • A MQ has to be maintained. The Configuration Service and data lakehouse job workers have to implement job event production and consumption functionality.

Table 4.7: Advantages and Disadvantages of a custom Event-Driven Scheduling and Orchestration

Hybrid approach: Based on the previous two design approaches, a hybrid design is considered thirdly. It involves the use of a centralized workflow orchestrator for single-tenant OnPremises installations and an event-driven architecture for multi-tenant mode. The advantages and disadvantages of this hybrid approach are discussed in table 4.8:

Advantages	Disadvantages
• Leverages existing workflow orchestration solutions in single-tenant mode (usage of an industry-proven workflow engine with a user-friendly interface). • Ensures tenant isolation in multi-tenant mode by design via the Configuration Service. • Highly scalable event-driven architecture in multi-tenant mode addressing the scalability requirements that are introduced by multiple tenants that use the same infrastructure. • Airflow introduces less complexity in contrast to the direct implementation of a custom event-driven scheduling and orchestration solution. • If business logic is implemented separately from orchestration logic, multi-tenancy can be added later by replacing Airflow.	• Additional complexity if both orchestration approaches have to be maintained at the same time. • Jobs must be implemented differently depending on whether they are triggered by a MQ or by Apache Airflow.

Table 4.8: Advantages and Disadvantages of a Hybrid Scheduling Approach

Decision

Multi-tenancy introduces additional complexity and requires strict isolation between tenants, which traditional orchestration platforms like Airflow do not fully support. Initially, the architecture should focus on implementing a single-tenant solution based on the usage of a centralized workflow orchestrator such as Apache Airflow.

Multi-tenancy support via an event-driven mechanism can be added if strictly required and multiple deployments for each tenant are no longer an option. This solution involves a scheduling component, the Configuration Service, that publishes job events to a MQ (Lakehouse Scheduling MQ). Job events contain all metadata required to configure job execution, including information to identify

tenant specific configurations. Spark Workers independently consume job events and run jobs on the data lakehouse. Workers publish completion events after they have finished the execution of business logic. If a workflow automation platform is enhanced by real multi-tenancy support with strict data isolation for multiple tenants, this decision should be reconsidered. It would simplify the architecture if scheduling and orchestration are solved with the same solution strategy in both single-tenant and multi-tenant setup.

Consequences

The decision to adopt an hybrid scheduling and orchestration approach has the following impact to the architecture:

- **Technical Impact:** Executors must be implemented differently depending on whether a MQ or a centralized workflow orchestration platform is used. To ease reusability of core components, business logic should be separated from orchestration logic. This can be realized for the Apache Airflow solution by separating DAG scripts and scripts to start a job from pipelines that include the core business logic. For the MQ solution independent modules to consume and publish job events should be created to allow the reuse of common business logic. Additionally, in multi-tenant setups observability functionality must be implemented that is able to replace required monitoring features. In single-tenant setups basic observability features are provided by Apache Airflow.
- **Organizational Impact:** Developers and data engineers must become familiar with two different scheduling and orchestration approaches, Apache Airflow and MQ driven orchestration).
- **Operational Impact:** There is a significant difference between the single-tenant and multi-tenant deployments. Single-tenant setups involve the deployment of Apache Airflow together with the source code and an Apache Spark cluster to execute Spark jobs. Multi-tenant deployments require a Spark cluster and a MQ such as Apache Kafka. Observability becomes more complex as separate tools are required to monitor distributed job execution. Fair resource allocation must be enforced to avoid resource starvation between tenants.

4.2.4 Customizable Data Lakehouse Configuration

Context

Different tenants may require different configurations in the domains data ingestion, data integration and scheduling and orchestration. Tenants may use different systems or use only a subset of the systems that are supported (e.g.,

some tenants rely on the ticketing system of Gitlab, others favor using Jira). As different data sources are supported and these data sources can be configured differently, some integration rules need to be adjustable. Also tenants may require different schedules for ingestion and integration jobs. Job schedules must be able to be adapted to a tenant's specific setting. For some tenants daily ingestion can be sufficient, others may require hourly intervals. On top of that, API keys and credentials of tenants differ and require separate and secure processing.

To allow adaptable configurations, the system must be able to perform Create, Read, Update, Delete (CRUD) operations on a tenants configuration. This configuration must be available to multiple jobs running in the data lakehouse. To maintain consistency of configuration data, all jobs should see the same configuration state at the same time.

Alternatives

Two possible solution designs for configuration management were considered:

Static configuration in configuration files: A simple approach is to maintain configuration in static configuration files that are deployed with the application. Each job must be deployed with the actual desired configuration that is required for its execution. For a multi-tenant setup this approach would mean that for each tenant and each job a separate worker has to be deployed. This worker would be exclusively responsible for handling a specific job for a specific tenant. Table 4.9 summarizes the advantages and disadvantages of this approach.

Advantages	Disadvantages
• Simple setup as configuration files can be adapted together with code changes. • Low operational overhead - configuration files are held with the building blocks that they configure.	• This approach is considered as not scalable. For multi-tenant setups the advantage of shared infrastructure use can not be achieved. • Deploying workers with different configurations for the same business logic introduces unnecessary operational overhead in the multi-tenant mode. • Configuration can hardly be changed at runtime. If configuration changes during job runtime, a job may run into errors given that it started with an old configuration version and has to terminate with the new configuration state.

Table 4.9: Advantages and Disadvantages of Using Static Configuration Files

Separate configuration service: Configuration is separated from application logic and is stored and managed by a central Configuration Service. Jobs can access the configuration that is required for their execution by calling the Configuration Service, e.g., via a provided REST interface. The Configuration Service provides read and write access to the configuration at runtime and acts as the single point of truth for configuration files. Jobs can get a configuration version at startup from the Configuration Service and then use this configuration state until their termination. Given this startup configuration loading, existing jobs are not affected by configuration changes at runtime.

For a single-tenant setup it would be sufficient if the Configuration Service is capable of writing on configuration files that are stored statically on its hard drive. For multi-tenant setups a dedicated database responsible for holding configuration data is beneficial. This aspect is discussed in 4.2.4. In a multi-tenant setup the Configuration Service also has to implement security and access control mechanisms and must ensure tenant specific configurations are processed securely. This includes securely storing credentials of tenants.

Advantages	Disadvantages
• Configuration and usage data are separated. • Configuration management is independent of jobs in the data lakehouse. • If configuration is separated from other parts of the data lakehouse, multiple versions can exist enabling different configurations for multiple tenants. • Jobs can retrieve different configuration files and be used by multiple tenants.	• Jobs must be able to work with different configuration states, if a downtime needs to be avoided. • Long-running jobs may be running with outdated configuration files if configuration changes at runtime. • Complexity to decide when to use which configuration. • Communication problems between the Configuration Service and jobs lead to job starvation. • The Configuration Service acts as single point of failure in the system.

Table 4.10: Advantages and Disadvantages of Using a Centralized Configuration Service

Decision

The solution adopts the establishment of a microservice, named *Configuration Service*, that stores and manages tenant-specific configurations separately from the data lakehouse. This approach ensures that different configurations for multiple tenants can be handled. Configuration can be changed at runtime without the need for new deployments of every job that requires configuration. So configuration updates do not affect jobs running in the data lakehouse.

Consequences

- **Technical Impact:** An interface of the Configuration Service, that can be used to access configuration data for jobs running in the data lakehouse, is required. This can be implemented as versioned REST interface for low-latency access. Jobs must be able to process the configuration, that the Configuration Service provides, with this interface. They only need read access to the configuration and are not responsible for modifying it. This is the single responsibility of the Configuration Service. The Configuration Service is also responsible for secure handling and storage of credentials.

  The configuration handling allows for multi-tenancy and is scalable as each tenant can access and update its configuration independently and concurrently. Configuration changes do not affect jobs, because configuration management is independently realized. If a job accesses the configuration

and the configuration is altered directly after the job has accessed it, the job is still running with the old configuration.

- **Organizational Impact:** The Configuration Service acts as single point of truth for configurations of the data lakehouse. As a microservice it has the single responsibility for configuration management and provides access to the data via a structured API. This eases administration and management of configurations.
- **Operational Impact:** The chosen approach introduces a separate building block for configuration management that has to be operated and maintained. Consequently, the operational complexity increases. Furthermore, long-running jobs in the data lakehouse may still run with an old configuration. If jobs with old configuration states are not compatible with newer application versions, they terminate with errors. If the time, while outdated configuration data is still used, should be limited, long-running jobs must be informed about configuration changes via a MQ or poll their configuration status regularly.

4.2.5 Configuration Management in Separate Database

Context

The Configuration Service as introduced in 4.2.4 is responsible for efficiently storing and managing configurations separately from the data lakehouse. Low-latency read access with high availability to configuration data is a major concern for jobs running in the data lakehouse as jobs are blocked without configuration access. These jobs must also be able to process the retrieved configuration. For that reason, the Configuration Service must ensure that configuration updates follow the appropriate configuration schema and validate any updates. In multi-tenant mode it must also ensure transactional consistency to avoid conflicts in concurrent configuration updates. It must further isolate sensitive data of tenants such as credentials like API keys from other tenants and ensure no unauthorized access to sensitive data is possible.

To fulfill these requirements the Configuration Service requires itself efficient access to the configuration data and must validate configuration updates. This includes at least a validation of the schema of the updated content.

Alternatives

The following storage and schema validation strategies for configuration data were considered:

Store configurations inside the Data Lakehouse (Delta Lake/S3): One option is storing configurations in the data lakehouse. So configuration and real data are stored in the same storage. Schema can be validated with the storage framework that is used to realize the data lakehouse, e.g., Delta Lake.

Advantages	Disadvantages
• An already existing storage is reused. • Delta Lake allows to enforce schemas.	• Configuration and real data is not strictly separated. • Configuration data must be optimized for efficient frequent low-latency reads, what requires performance optimization strategies.

Table 4.11: Advantages and Disadvantages of Storing Configurations Inside the Data Lakehouse (Delta Lake/S3)

Usage of a Separate Database (e.g., relational, document-based): Store the configuration in a dedicated database that also allows to enforce a schema. Selecting an appropriate database depends on implementation of the configuration files. E.g., if the system can be configured using JavaScript Object Notation (JSON) files, a document-based database, that is built on top of JSON schemes, is a beneficial choice.

Advantages	Disadvantages
• Fast low-latency data access with transactional consistency. • Allows querying of parts of the configuration using a query language. • Depending on the chosen database, schema evolution is also achievable.	• Requires maintaining an additional database.

Table 4.12: Advantages and Disadvantages of Using a Separate Database for Configuration Management

Usage of a Dedicated File Storage: Store the configuration files as objects in an object store (e.g., S3) or as files in a distributed file system (e.g., HDFS). The Configuration Service handles schema validation completely. Advantages and Disadvantages of using cloud object stores and HDFS have already been extensively discussed in subsection 4.2.1. Table 4.13 only summarizes important aspects regarding their usage as a simple file storage for configuration data.

Advantages	Disadvantages
• If static configuration files that are shipped with the application have been used beforehand, the migration is simple. The only difference is that configuration data has to be loaded from the file system or object store over the network. So only the location of configuration data changes. • Simple, highly available solution.	• Efficient query capabilities are limited, if more than simply loading entire configuration files by keys, is required. • The solution is not ideal for managing structured configuration settings. It lacks schema validation support.

Table 4.13: Advantages and Disadvantages of Using a dedicated File Storage for Configuration Management

Store configuration files at the disk of the Configuration Service: Keep configuration files in the container or file system of the host of the Configuration Service. The Configuration Service is responsible for schema validation.

Advantages	Disadvantages
• Simple and fast approach as application data can be directly accessed from the same machine or container.	• This approach is not considered scalable. When computation is distributed, each container must be provisioned with the correct configuration. For this reason, runtime updates are challenging. • This solution lacks transactional support of a database system and schema validation support. • Configuration changes at runtime are possible, but can influence already running jobs.

Table 4.14: Advantages and Disadvantages of Storing Configuration Files on the Local Disk of the Configuration Service

Decision

The Configuration Service has as a microservice the single responsibility for managing configuration data. For a single-tenant solution runtime updates are not

a strict requirement as the solution is initially configured. It is expected that changes happen only rarely afterward. Therefore, it is acceptable to redeploy the Configuration Service for once if configuration changes are required. The Configuration Service must ensure in this setting that the updated configuration files are valid. But this can be checked with tests before the deployment to production.

If the solution is enhanced by multi-tenancy, configuration should be stored in a separate database with schema validation managed by the Configuration Service. The Configuration Service has the responsibility for managing this database and providing secure access control to it. This strategy allows concurrently updating configuration data of multiple tenants at runtime. Other tenants are not affected by a tenants configuration changes. As this involves changes of configuration data at runtime, the updates must be validated by the Configuration Service. Choosing an appropriate database that allows enforcing a schema, helps in maintaining data quality.

This approach ensures efficient low-latency access for frequently read data and a separation of configuration and real data as configuration data is stored independently of the data lakehouse. For single-tenancy, the additional complexity of introducing an additional database is neglected. This architecture decision was subject of a structured review as discussed in the evaluation in chapter 7. At first, it was suggested to always introduce a separate database. The deviation for single-tenancy was introduced to address concerns during the review that this additional system complexity is not always really required. This decision should lower system complexity and facilitate initial development. However, for scalable configuration processing of configuration data of multiple tenants the dedicated database is required.

Consequences

- **Technical Impact:** Fast, optimized configuration reads improve the performance of Spark jobs. For this reason, the Configuration Service must offer low-latency access to configuration data by a clearly-defined (REST) interface. In single-tenant mode, the Configuration Service has to be redeployed if configuration changes. In multi-tenant mode, the Configuration Service handles updates of the configuration data to ensure consistency and communicates with an additional database for configuration data. The Configuration Service should offer a Management API with which users can alter configuration data. This API should be implemented separately from a data access API for Spark jobs. Jobs running in the data lakehouse only need read access to configuration data and should not be able to modify it.

  The Configuration Service is also responsible to control authorized access to sensitive tenant-specific data such as credentials. Credentials need to be

stored securely. This complexity must be tackled using database mechanisms (dependent on the used database) or by the Configuration Service.

- **Organizational Impact:** Improved multi-tenant data handling allows different configurations per tenant without affecting other tenants. The microservice Configuration Service acts as single source of truth for all configurations.
- **Operational Impact:** In multi-tenant mode, the complexity of maintaining a separate database for configuration data is introduced. This database can be a potential single point of failure if the database is deployed non-replicated. It should therefore be deployed as a cluster to ensure high availability. Furthermore, the database should be monitored, and backup procedures should be applied.

4.3 Derived Solution Strategy

Based on the technical context and key design decisions that have been introduced in the previous sections, this section derives the overall solution strategy for the software development analytics platform. Initially, a single-tenant system design is presented. Subsequently, it is discussed how the solution can be extended to support multi-tenancy and the architecture discussed in further detail from multiple architectural viewpoints. Lastly is shown in detail how a data lakehouse for software development data can be modeled from a data engineering perspective.

4.3.1 Single-Tenant Solution

The single-tenant solution targets individual organizations that require software development analytics, but do not need the additional complexity of multi-tenancy. Based on the key design decisions of the previous section, the solution focuses on simplicity, modularity and extensibility.

Architecture Overview

The designed solution consists of multiple layers:

- **Data Lakehouse:** The data lakehouse is modeled according to the Medallion architecture. It uses Delta Lake as storage framework and stores Apache Parquet files and the Delta Lake log in a S3 compatible object store. Multiple Spark jobs are responsible for data ingestion and integration between different layers of the data lakehouse.

- **Orchestration:** Apache Airflow orchestrates the execution of Spark jobs. Workflows and ingestion schedules are defined using DAGs.
- **Services layer:** The services layer is sliced clearly according to the two different domains analytical insights and configuration.
  1. The Configuration Service manages and offers the system's configuration data.
  2. The Analytics Service is responsible for providing access to analytical insights gained from software development data.
- **Frontend:** A React application serves as User Interface (UI) to visualize analytical insights and configure the system. It communicates through REST interfaces with backend services.
- **External Systems:** The system is connected to multiple data sources that provide the software development data.

An overview of the single-tenant software architecture is given in figure 4.3.

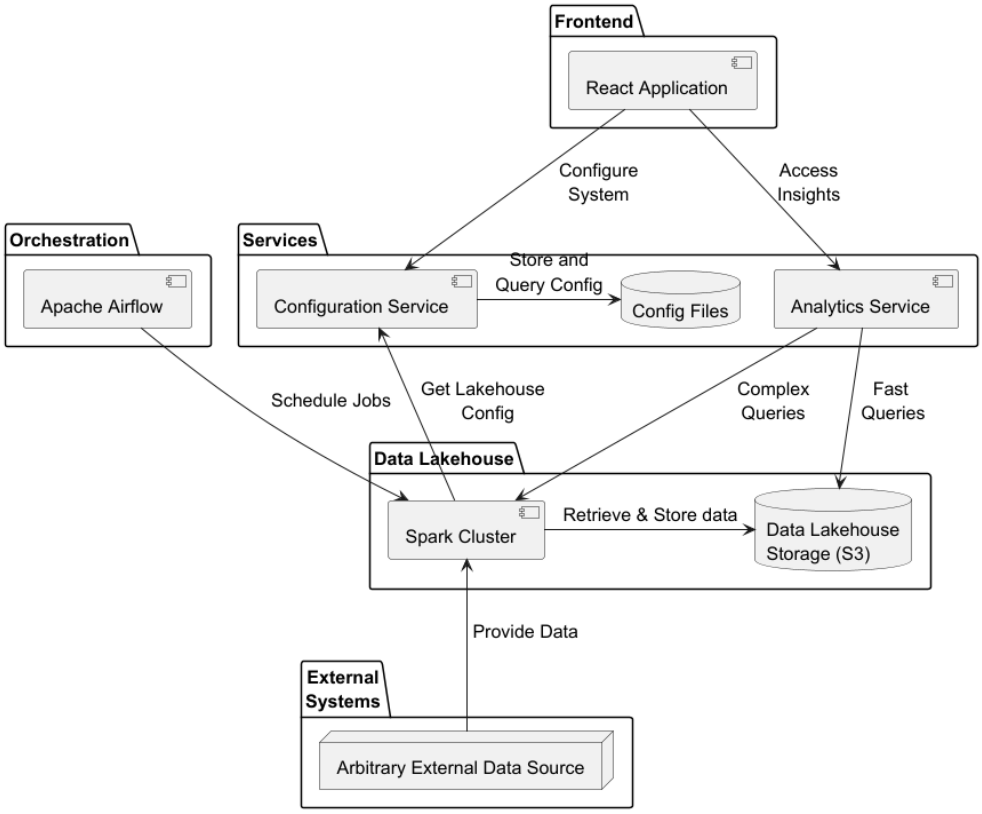


Figure 4.3: Overview of the Single-Tenant Software Architecture

Detailed Component Responsibilities

This section dives deeper into the responsibilities of single components in the architecture.

Data Lakehouse: The data lakehouse is modeled according to the Medallion architecture. Data ingestion jobs run on a regular basis to ingest data in its raw format directly into the Bronze layer. Afterwards, a data integration job curates the data from the Bronze layer and integrates it into the Silver layer. Analytical results are stored in the Gold Layer. The data lakehouse uses Delta Lake as storage framework and stores Apache Parquet files and the Delta Lake log in a S3-compatible object store. Multiple Spark jobs are responsible for correctly handling data ingestion and integration between different layers of the data lakehouse.

Orchestration: Data ingestion and integration jobs require orchestration. Orchestration of Spark jobs is realized with Apache Airflow. In Apache Airflow DAGs are defined which control the execution order of multiple Spark tasks. This also contains the definition of ingestion schedules.

Configuration Service: The Configuration Service acts as single point of truth for configuration data in the architecture. It manages the configuration files that configure the data lakehouse, provides access to this configuration and is responsible for providing a management API with which the frontend can modify configurations. Configuration files are stored on the local disk of the Configuration service or within its container for simplicity.

Analytics Service: The Analytics Service is responsible for providing access to analytical insights gained from software development data. It offers a REST-API with which external systems such as the Frontend can access analytical insights via a structured and versioned interface.

The Analytics Service is responsible to execute both complex and fast queries. Long-running analytical queries are submitted as a Spark job to the Spark cluster. This allows leveraging Spark for complex queries as their complexity may require distributed computation. The results of fast queries are derived directly from the storage of the data lakehouse. Therefore, the Analytics Service must access the Gold layer of the Medallion architecture directly by using Delta Lake and retrieving Delta tables in the form of Apache Parquet files from the S3-compatible object storage.

Frontend: The frontend is a React and TypeScript-based single page application. It visualizes analytical insights and offers configuration options to the user.

Analytical insights are retrieved through a REST interface from the Analytics Service. The Frontend uses another REST-API provided by the Configuration Service to allow users to manage data sources and transformation rules. The responsive UI should clearly separate the concerns presenting analytical insights and configuring the system as these functionalities also addresses different stakeholders.

External Systems: Data ingestion jobs communicate with various external data sources to ingest data. Multiple data ingestion jobs can run at the same time to retrieve data from multiple data sources in parallel. However, rate limits of external systems have to be acknowledged. This is especially important if multiple data ingestion jobs communicate with the same external system at the same time.

4.3.2 Extension to Multi-Tenancy

Architecture Overview

The multi-tenant solution targets building a large central platform for software development data of multiple teams or multiple organizations that is able to run on a shared infrastructure. This requires strict isolation of data between tenants to protect their sensitive data.

In contrast to the single-tenant architecture, two new building blocks are introduced:

- **MQ (e.g., Kafka):** A MQ is responsible for distributing job events to multiple Spark workers that perform a job passed on configuration provided with the event. It replaces Apache Airflow,
- **Config DB (e.g., CouchDB):** Differing configurations of multiple tenants are stored in a dedicated database.

An overview of the multi-tenant architecture is given in figure 4.4.

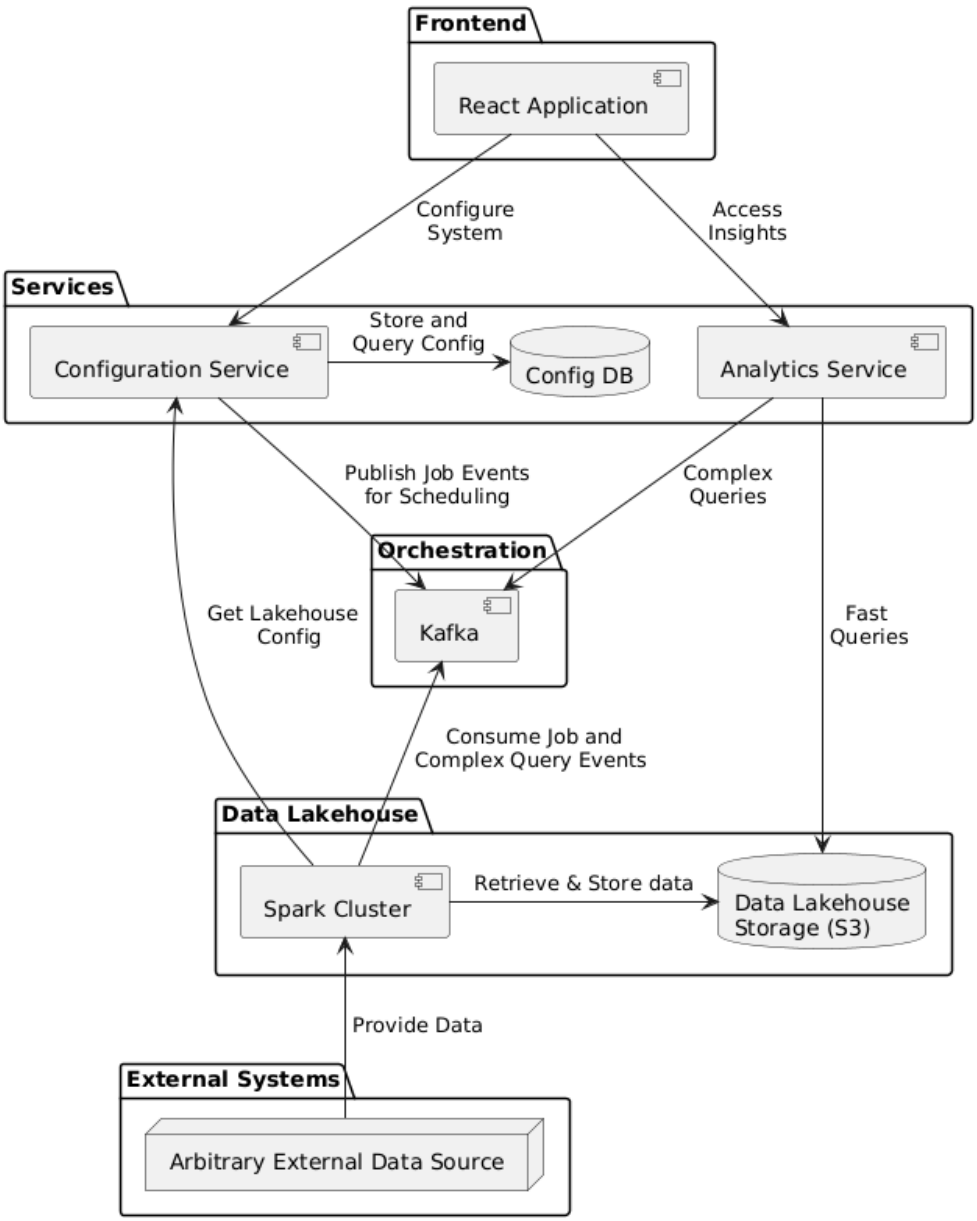


Figure 4.4: Overview of the Multi-Tenant Software Architecture

Additional Component Responsibilities

This section dives deeper into the responsibilities of single components in the multi-tenant software architecture. It is only stated what is different in comparison to the single-tenant architecture. Responsibilities, that are equal in both single-tenant and multi-tenant architecture are neglected.

Data Lakehouse: The data lakehouse is equally organized as in the single-tenant architecture. However, each table must contain a column `tenant_id` with a unique identifier that identifies to which tenant the data belongs.

Furthermore, Delta Lake allows to partition a table by a column. Internally, Delta Lake then stores each partition in a separate Apache Parquet file (Powers & Zhu, 2023). It is recommended to choose a column of a low cardinality for this, which are often f.e. dates. (The Linux Foundation, 2025). In the data lakehouse for software development data, each tenant owns large amounts of data. As long as the cardinality of the column `tenantId` does not become very high, it is further recommended to partition the data lakehouse by `tenantId`. This has significant performance benefits as Delta Lake can skip not relevant Parquet files of other tenants during query runtime. On top of that, data of each tenant is stored in distinct files, what can be seen as additional tenant data isolation.

Configuration Service: As in the single-tenant architecture, the Configuration Service is the single point of truth for configuration data in the solution. However, it now has to manage the complexity of providing access to configuration data of multiple tenants. The management API to modify configurations must be protected by security and access control mechanisms. A tenant must only retrieve and modify configuration data that he is responsible for.

Configuration data is stored and retrieved from a dedicated database called `Config DB` in figure 4.4. The Configuration Service and the Config DB must ensure that configuration data updates comply to the desired configuration schema and validate configuration updates.

Additionally, the Configuration Service is in the multi-tenant solution also responsible for scheduling data lakehouse ingestion and integration jobs. As the Configuration Service manages the configuration data, it is also responsible for ensuring that configurations are applied. If an ingestion or integration job is due, the Configuration Service sends a job event to the `Data Lakehouse Scheduling MQ`. It consumes completion events from the `Data Lakehouse Scheduling MQ` to be informed about completed jobs.

Config DB In contrast to the single-tenant solution, configuration data cannot be stored as files on the host machine or in the container, as this does not scale. Processing configuration data of multiple tenants requires a separate data store, in which the `tenant_id` is the primary key. To ease ensuring the data quality of configuration data, it is helpful to choose a database that supports enforcing a schema. For example, if configuration data is modeled and stored in the JSON-format, it is beneficial to choose a document-based database that supports enforcing JSON-schemes.

Analytics Service: Also in the multi-tenant architecture fast queries are directly executed with Delta Lake on the S3 storage of the data lakehouse. However, complex long-running queries must be distributed across multiple workers to ensure load balancing. For complex queries, also an event-driven approach is conducted, where the Analytics Service publishes query event to a MQ, called in the architecture `Data Lakehouse Complex Query MQ`.

Once a query result has been calculated, the Analytics Service is notified by another event. It is then responsible for providing the user with the result. Dependent on the amount of data that the query result comprises, it could be necessary to introduce a result cache.

Spark Job Orchestration (Kafka): For the multi-tenant solution an event-driven approach for scheduling and orchestration is conducted. The Configuration Services publishes job events and consumes completion events after a job has been finished.

A highly-available MQ is responsible for distributing work. Spark workers that run jobs in the data lakehouse consume job events, execute the desired job and send a completion event after business logic has been executed. To execute the job, workers require configuration. For this reason, the job event contains the `tenant_id` and all other necessary information to retrieve configuration data from the Configuration Service. A job is started by consuming a job event. It then loads initially the configuration data from the Configuration Service via a REST-interface to acquire the configuration that is necessary to run the job. Afterwards, the actual business logic of the job is performed and a completion event is sent upon finishing.

The multi-tenant solution requires multiple workers that consume different events. Workers responsible for data ingestion, Bronze to Silver integration and Silver to Gold integration consume job events that the Configuration Service publishes to the `Data Lakehouse Scheduling MQ`. Additionally, `Complex Query Executor` workers are required that consume events from the `Data Lakehouse Complex Query MQ`. They are responsible for asynchronously executing long running complex queries. The completion procedure is equal to the solution strategy with the `Data Lakehouse Scheduling MQ`. Upon finishing `Complex Query Executors` notify the `Data Lakehouse Complex Query MQ` about their completion. The Analytic Service is then responsible for providing the result to external systems such as the Frontend.

Optional Data Mart

As discussed in 4.2.2, a dedicated relational database that is responsible for handling fast queries, should only be introduced if strict performance requirements

cannot be met without it. Therefore, it is not part of the designed solution. Introducing it would require synchronization jobs that run on a regular basis. These synchronization jobs have to retrieve all data that was updated after the last synchronization time. Then, they have to transform the data, that is formatted according to the data model of the Gold layer, to the data model of the Data Mart and store the update to the Data Mart. To allow for such a synchronization algorithm, tables in the data lakehouse require a column `load_time` that can be used to identify data that has not already been moved. However, such a synchronization has the disadvantage that the actuality of data in the Data Mart is always behind the corresponding Gold Layer, which may inherit data staleness issues.

4.3.3 Software Architecture Views

After having discussed key design decisions and having introduced both single-tenant and multi-tenant solution strategies, this subsection describes the architecture in further detail from multiple software architectural viewpoints.

As part of the created software architecture documentation, PlantUML diagrams of the building block view, runtime view and deployment view were created. For the purpose of clarity and to not overwhelm the reader, not every single building block is discussed from all viewpoints in the thesis. For these precise details, the architecture documentation in appendix A can be referenced. Instead, only selected building blocks, runtime views and the deployment views are presented to explain the proposed solution.

Logical View

Initially, the solution strategy is presented from a logical viewpoint illustrating core logical components of the software architecture.

Participating Building Blocks with their Exposed Interfaces Figure 4.5 illustrates all major building blocks of the solution including their provided and used interfaces. Apache Airflow is used in the single-tenant mode to trigger `Data Ingestion`, `Bronze To Silver Integration` and `Silver To Gold Integration`. In the multi-tenant solution, an event-driven approach is conducted that involves these three workers to consume events from the `Data Lakehouse Scheduling MQ`. Job events are published by the Configuration Service that is responsible for scheduling tasks.

Additionally, the `Data Lakehouse Complex Query MQ` is responsible for distributing query events that are consumed by `Complex Query Executors` and published by the Analytics Service. The MQs have been separated because query events and job events are different concerns and do not share the same model.

Query events must contain the query itself, while job events configure jobs by only referencing the configuration. All Spark workers have a dependency to the `Config API` exposed by the Configuration Service, which provides the required configuration data for their execution.

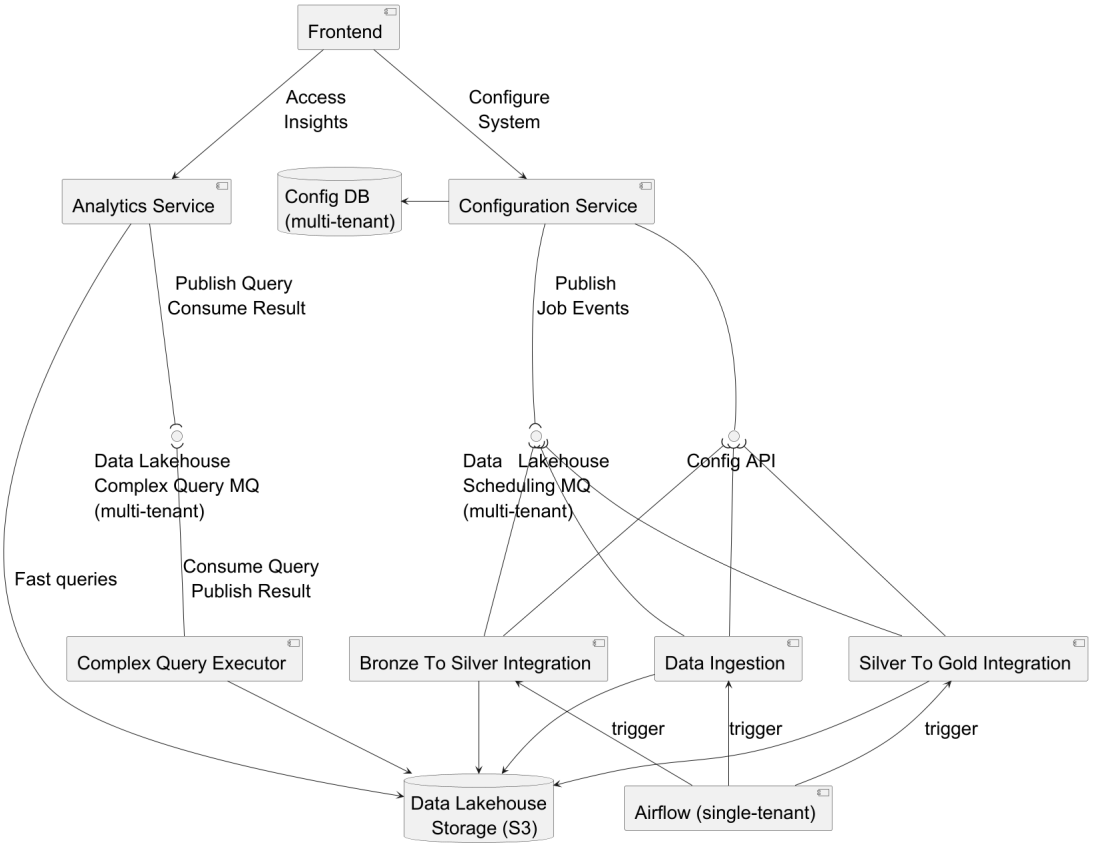


Figure 4.5: Whitebox View of the Overall System

Overview of Core Building Blocks In the following selected building blocks are detailed. Figure 4.6 shows a whitebox view of the Analytics Service. It includes functionality for both the single-tenant and multi-tenant solution. A `Metrics Provider` is responsible for executing fast queries directly against the Gold layer of the data lakehouse. It can either be implemented via Spark or by direct Parquet-file-based access from S3.

In the multi-tenant solution, a module `Complex Query Handler` of the Analytics Service is responsible for handling complex queries via the `Data Lakehouse Complex Query MQ`. In the single-tenant solution the Analytics Service itself submits complex queries to the Spark cluster.

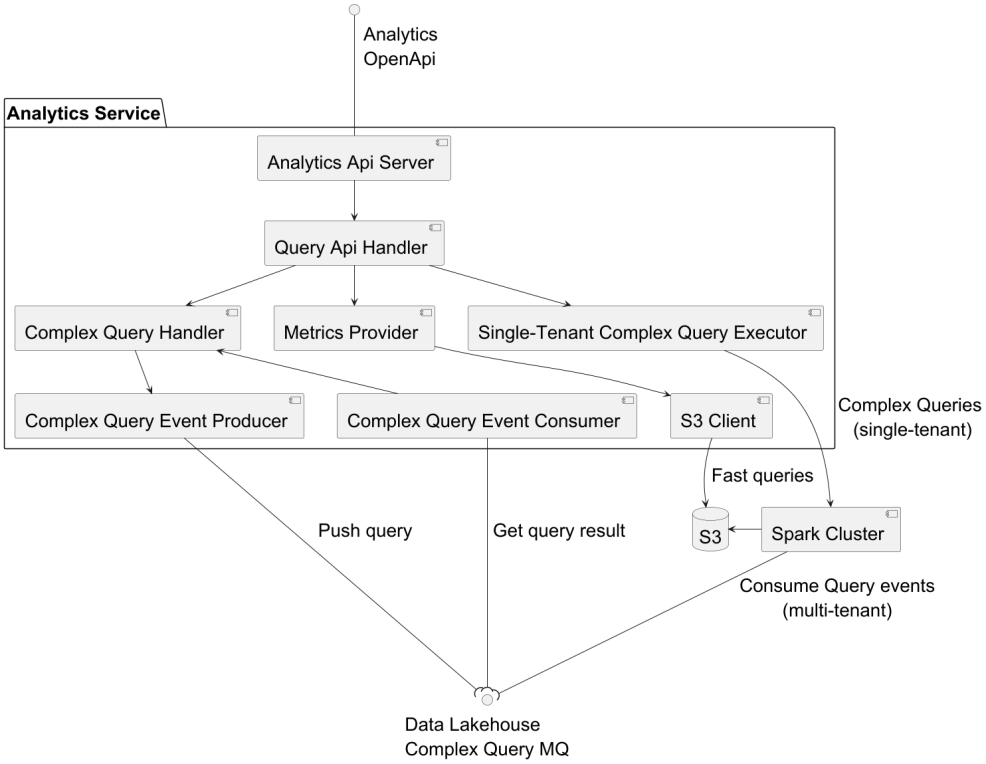


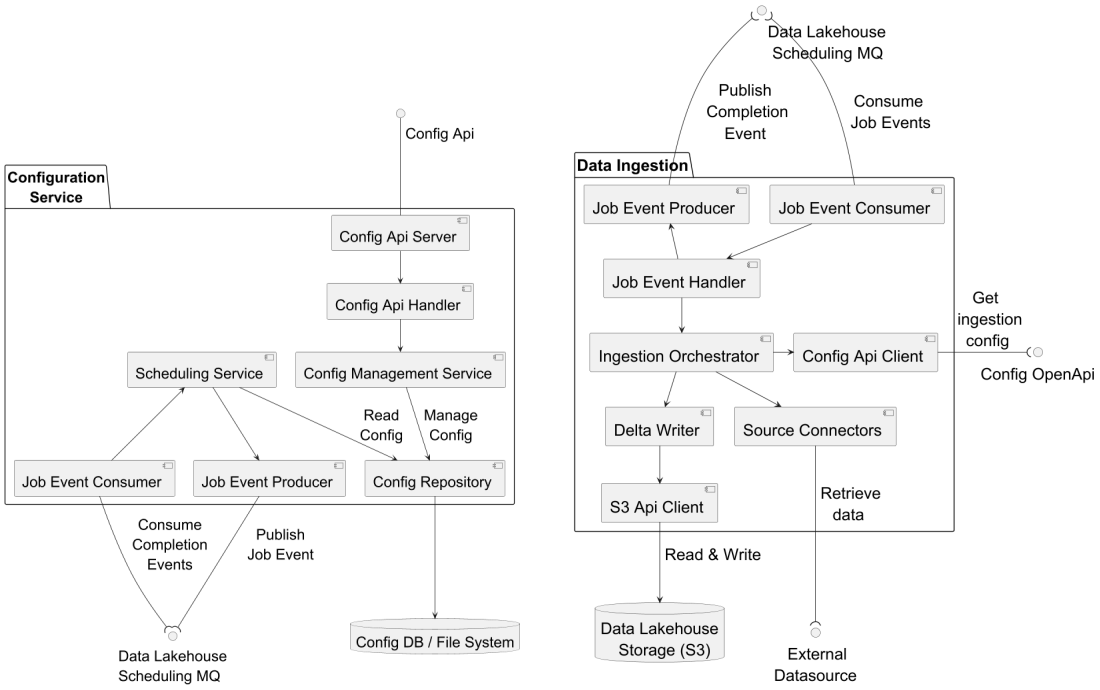
Figure 4.6: Whitebox View of the Analytics Service

Figure 4.7 presents the whitebox view of other core components. The Configuration Service provides an API to access and manage configuration data. In the multi-tenant setup, it is also responsible for scheduling via the `Data Lakehouse Scheduling MQ`.

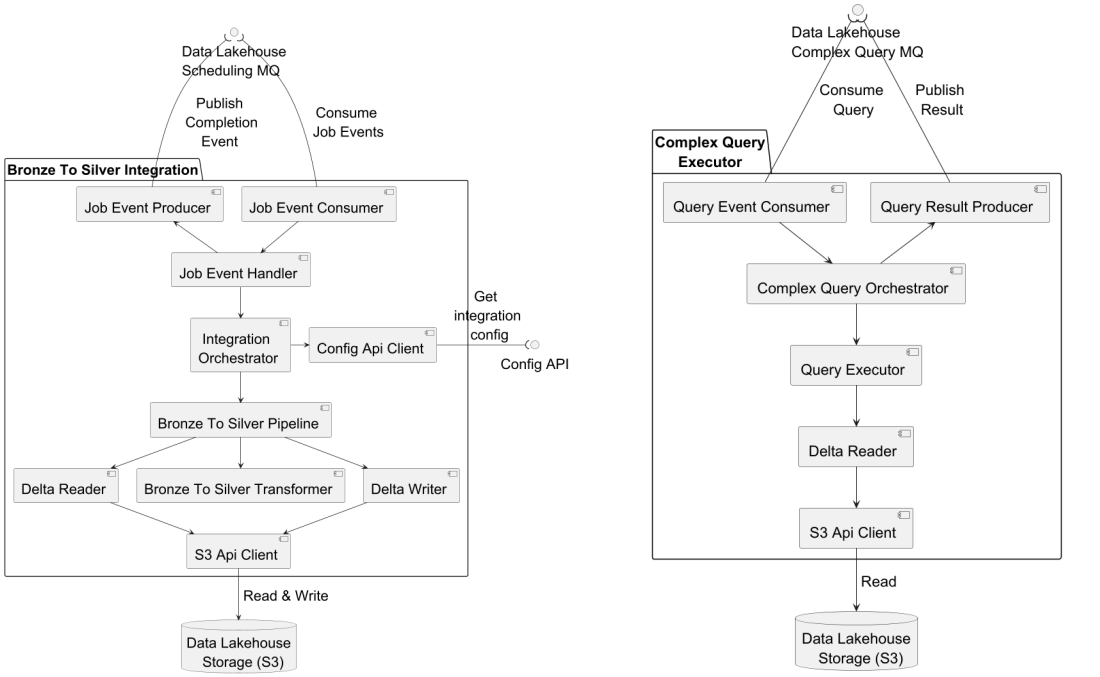
The data ingestion of sub-figure 4.7b writes ingested data to S3 using Delta Lake. This building block has dependencies to external data sources and the Configuration Service. In single-tenant mode, it is triggered via Apache Airflow. In multi-tenant mode, it consumes job events from the `Data Lakehouse Scheduling MQ`.

The Bronze to Silver Integration illustrated in sub-figure 4.7c and the Silver to Gold Integration (not depicted) share the same structure. They both have a dependency to the Configuration Service. Execution steps include configuration loading, reading data using Delta Lake, transforming the data, and writing it back to the data lakehouse.

The Complex Query Executor only consumes query events, is responsible for their execution using Spark and data from the data lakehouse, and publishes the query result back to the MQ.



(a) Configuration Service **(b)** Data Ingestion



(c) Bronze to Silver Integration **(d)** Complex Query Executor

Figure 4.7: Whitebox View of Core Components of the Logical Architecture

Process View

After outlining core logical components, the system is assessed from a runtime point of view, which illustrates how the components interact with each other. First, the runtime view of a data ingestion and a data integration job are presented, because these jobs shape the overall system design. Subsequently, the complex query execution of the multi-tenant solution is presented, due to its inherent complexity.

Data Ingestion to the Bronze Layer Figure 4.8 illustrates the data ingestion of data from an external system to the Bronze layer of the Medallion architecture. The ingestion is triggered by a scheduler, which is represented by Apache Airflow in the single-tenant solution design. Subsequently, configuration data required for the job's execution, is fetched from the Configuration Service. Based on this configuration the external system like GitHub or Jira is called to fetch the data. To job finishes by storing the ingested data into the Bronze layer of the data lakehouse.

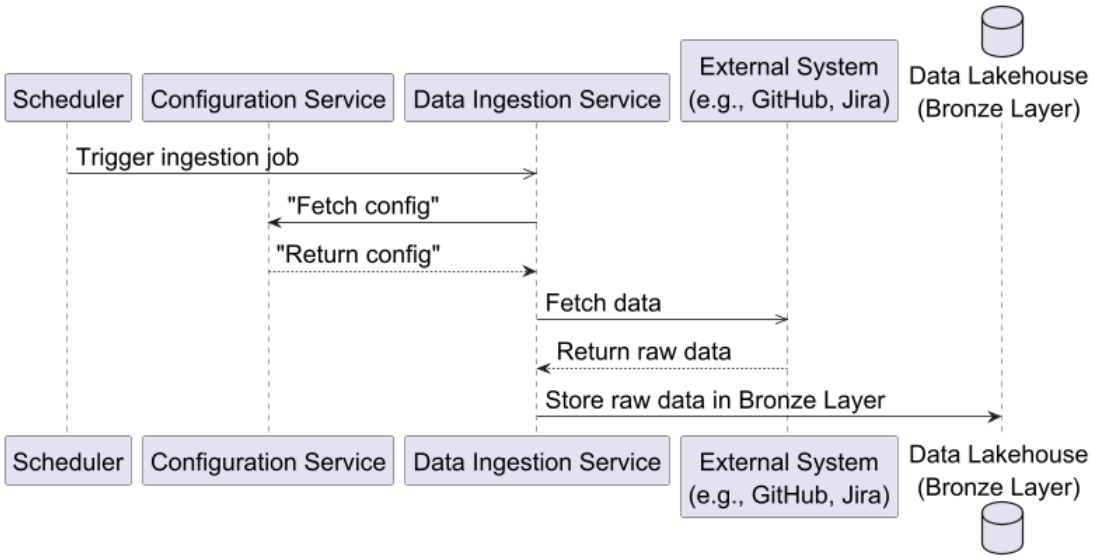


Figure 4.8: Process View - Data Ingestion to the Bronze Layer

Data Integration to the Silver Layer The integration of data from the Bronze layer to the Silver layer is presented in figure 4.9. Again, the job is triggered by a scheduler and the job fetches configuration data from the Configuration Service. Regarding the Bronze to Silver integration job, this configuration data consists of transformation rules. These rules are applied to loaded data of the Bronze layer and the result is written back to the Silver layer of the data lakehouse.

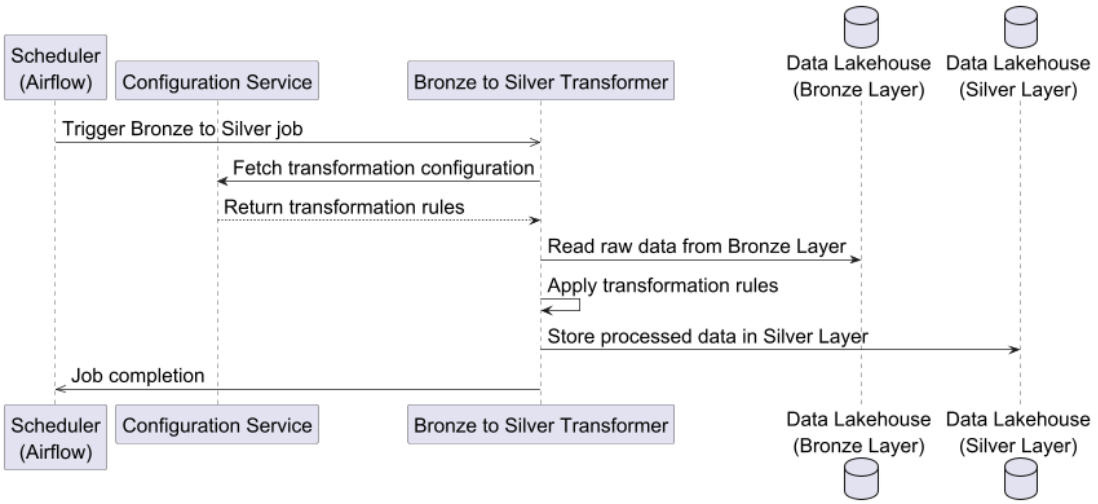


Figure 4.9: Process View - Data Integration from Bronze to the Silver Layer

Complex Query Execution Lastly, the runtime view of a complex query execution of the multi-tenant solution is presented. A tenant would like to retrieve advanced insights and therefore submits a request via the Frontend to the Analytics Service. This service publishes a query request event to the `Data Lakehouse Query MQ`, which is consumed by a `Complex Query Executor` Spark worker. This worker reads required data from the data lakehouse, possibly in batches, and executes the query using Spark. The query result is stored in a temporary results storage (e.g., in a Parquet file on S3). Upon finishing the `Complex Query Executor` sends a completion event to the `Data Lakehouse Query MQ`, which is consumed by the Analytics Service. This service retrieves the query result from the temporary results storage, if required in chunks, and serves the result to the Frontend and user.

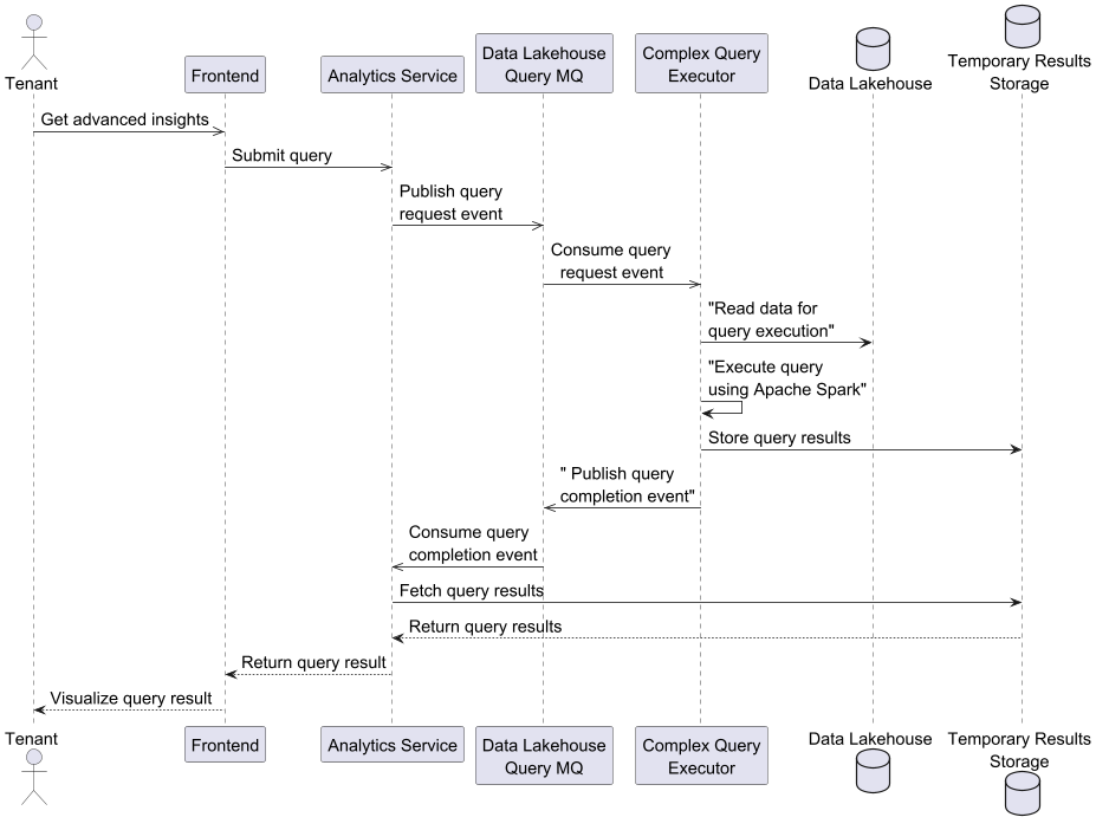


Figure 4.10: Process View - Complex Query Execution

Deployment View

Finally, the difference in the deployments of the single-tenant and multi-tenant solution is stated.

Single-Tenant Deployment Figure 4.11 depicts the deployment view of the single-tenant solution based on a dockerized deployment. Apache Airflow runs in a Docker container dedicated Docker container. Inside Airflow's Docker container multiple Airflow Workers submit tasks to Spark. This can be realized as submitting Spark tasks to a dedicated Spark cluster or by running Spark inside Airflow, which is discussed further in section 5.3.

Airflow applications communicate with another Docker container, that hosts a bucket, named `lakehouse`, in an S3-compatible object store. This bucket holds the entire data of the data lakehouse. Additionally, containers for the Analytics Service and the Configuration Service exists, that serve these two applications. The backend services act as gateway between the data lakehouse and external systems such as the Frontend that access the application from outside of the deployment infrastructure.

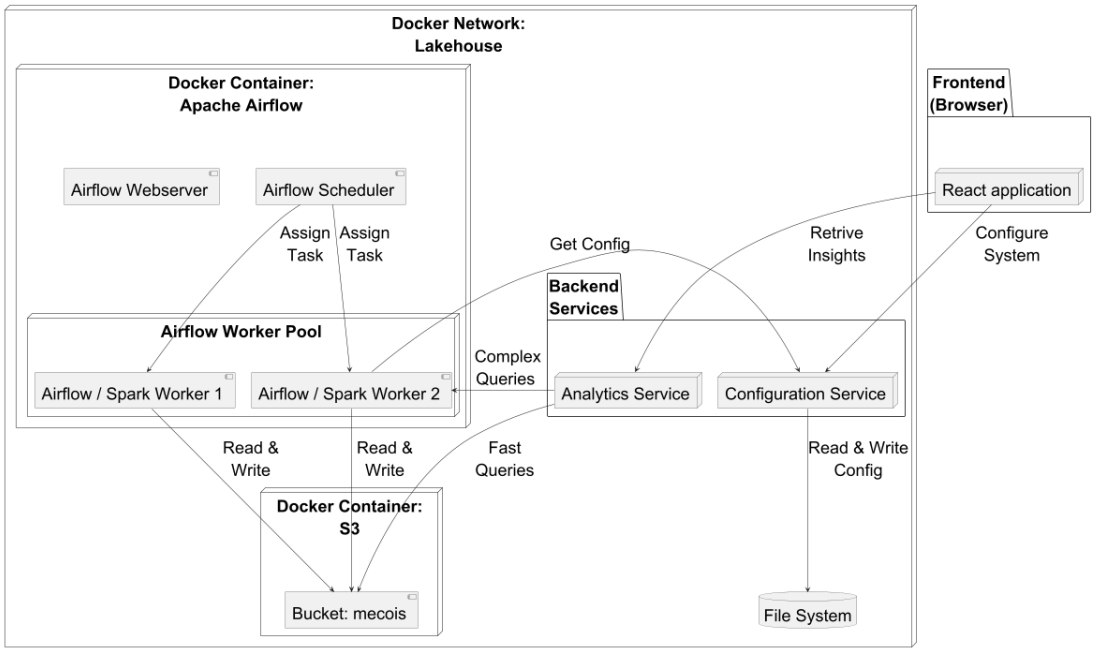


Figure 4.11: Deployment View of the Single-Tenant Solution

Multi-Tenant Deployment Figure 4.12 presents the deployment view of the multi-tenant solution strategy. In contrast to the single-tenant solution, scheduling and orchestration require a MQ. This necessitates an highly-available installation of a Message Broker such as Apache Kafka. Configuration files also require a dedicated database. As a document-based database is seen as beneficial for handling configuration files, a possible permissively licensed solution is CouchDB.

Furthermore, the solution strategy requires multiple workers to listen to incoming events. Therefore, instances of `Data Ingestion Services`, `Data Integration Services` and `Complex Query Executors` have to be started. The logical components Bronze to Silver and Silver to Gold integration are based on the same dependencies and follow a similar implementation. Therefore, they can be deployed as one `Data Integration Service`, which is configured by configuration of the Configuration Service, including the information to which layer the data has to be integrated.

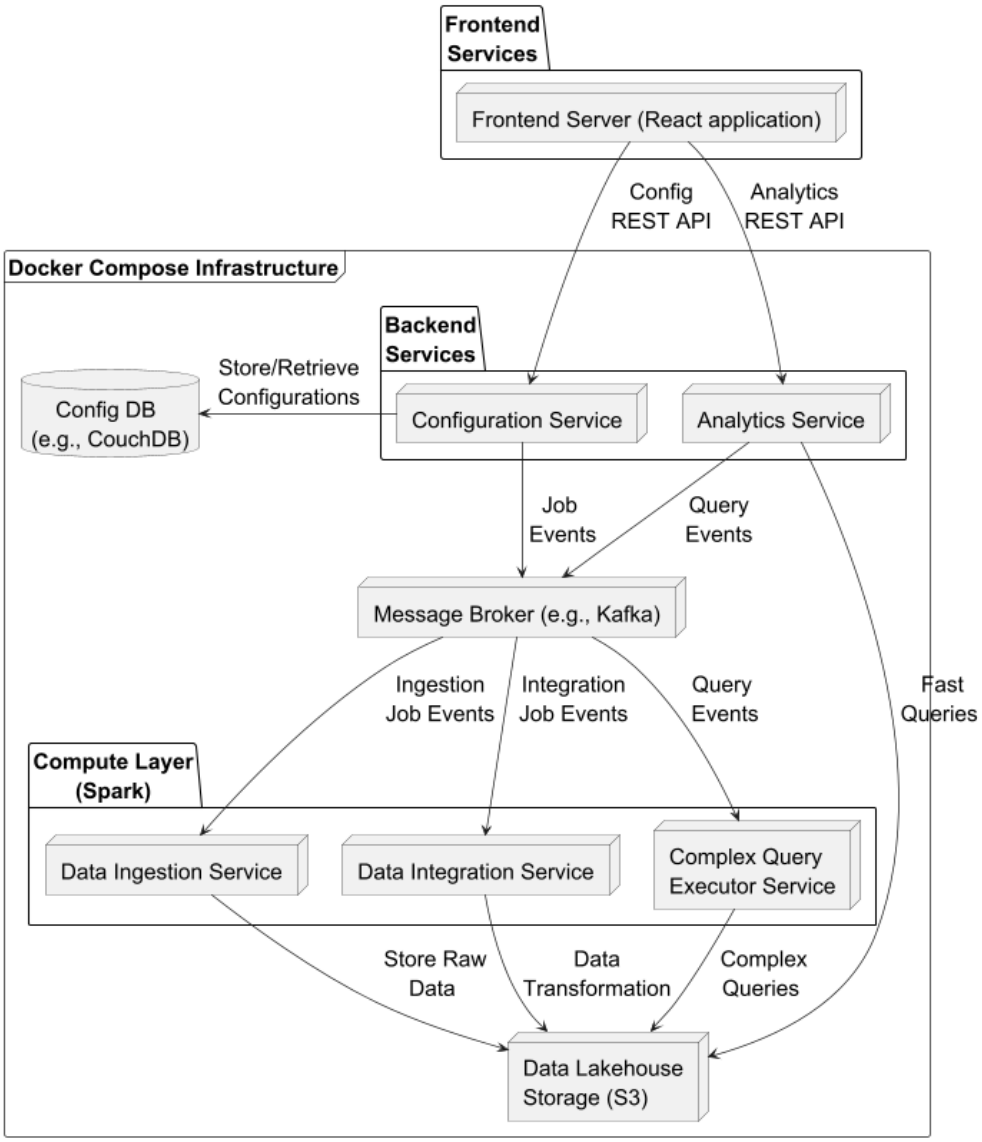


Figure 4.12: Deployment View of the Multi-Tenant Solution

4.3.4 Data Engineering of the Data Lakehouse for Software Development Data

The solution should be able to process data of various sources, which requires different source connectors for data ingestion and algorithms for data integration per data source. As previously stated in section 4.1, the solution adopts a Medallion architecture as data architecture of the data lakehouse. This subsection dives deeper into the concrete realization of a data lakehouse for software development data by outlining how the different layers are modeled from a data engineering perspective.

Data Vault as Model of the Silver Layer

To transform data from the Bronze to the Silver layer of the Medallion architecture it has to be cleaned and normalized. Correlating data from multiple data sources with distinct data models requires normalizing it to a unified data model. This unified data model at the Silver layer of the Medallion architecture is realized as a Data Vault in this thesis.

Data Vault Modeling was developed by Linstedt (2015). Originally, the Data Vault model was designed for data warehouses. Nogueira et al. (2018) first applied Data Vaults to the modeling of data lakes. A Data Vault model can be seen as a business-oriented model for data warehousing gathered from the original natural data model. Data Vaults represent business processes that can be identified by business keys. Business keys are keys, such as Universally Unique Identifier (UUID)s or e.g., an invoice id for invoices, that can be used to identify business objects. They can be represented by a single value or compound. (Linstedt, 2015)

Data Vaults consist of three entity types - Hubs, Links and Satellites (Linstedt, 2015):

- **Hub** entities do only contain business keys with some additional information (metadata). Their aim is to ease identifying business objects as they keep all business keys separately from contextual information. As metadata, often load date and record source are included. The load date is the timestamp when the data was loaded into the data warehouse. The record source identifies the original source of the ingested data. Including this metadata allows for better tracing problems.
- **Links** represent the relationship between these business keys by connecting Hubs. Describing the relationship (Links) between business keys (Hubs) is essential in the Data Vault model, but still both relationships and business keys need to be enriched with contextual information.
- **Satellites** hold the information describing business objects or the relationship between business objects and are associated to Hubs or Links.

Creating a Data Vault for Software Development Data involves multiple steps. These steps are summarized in Table 4.15, each with an example.

Step	**Example**
Multiple data sources contain common entities that share the same semantic model, but differ in their data model. They can be unified by identifying the business object. Business objects must have a unique key (business key).	Business objects such as a `Repository`, a `Commit` or an `Issue`
Create a Hub table for each business object. Each Hub stores a unique business key, load date and record source.	Create the Hub tables `Hub_Repository`, `Hub_Commit` and `Hub_Issue`.
Often data objects of software development data contain foreign keys that reference other objects. This represents relationships between business objects that have to be identified.	An `Issue` belongs to a `Repository`, a `Commit` is linked to an `Issue`
Create a Link table for each relationship. If two Hubs should be linked, the corresponding Link must include the business keys of both Hubs.	Create the table `Link_Commit_Issue`. The table `Link_Commit_Issue` contains the business keys of `Hub_Commit` and `Hub_Issue`
Identify attributes that describe the business objects in Satellites	The attributes `Created_at`, `Closed_at` and `Labels` of an `Issue`
Create at least one Satellite for each Hub with attributes that describe the business object that is represented by the Hub. The business key of each object that is described must be included as foreign key. A Hub may be described by multiple Satellites.	Create the table `Sat_Issue` with the attributes `Created_at`, `Closed_at` and `Labels`. `Sat_Issue` includes the business key of `Hub_Issue`

Table 4.15: Steps to Create a Data Vault for Software Development Data

The steps as shown in table 4.15 have been applied to software development data from GitHub, GitLab and Jira. The key underlying idea of modeling the Silver data with a Data Vault is to find business objects that are only represented by business keys. If only keys need to be merged to combine data from multiple

data sources in Hubs, this eases integration. Descriptive attributes are exclusively stored in associated Satellites.

Figure 4.13 presents an excerpt of a Data Vault model for software development data from Git. The `Hub_Issue` can contain issues from multiple data sources such as GitHub, GitLab and Jira.

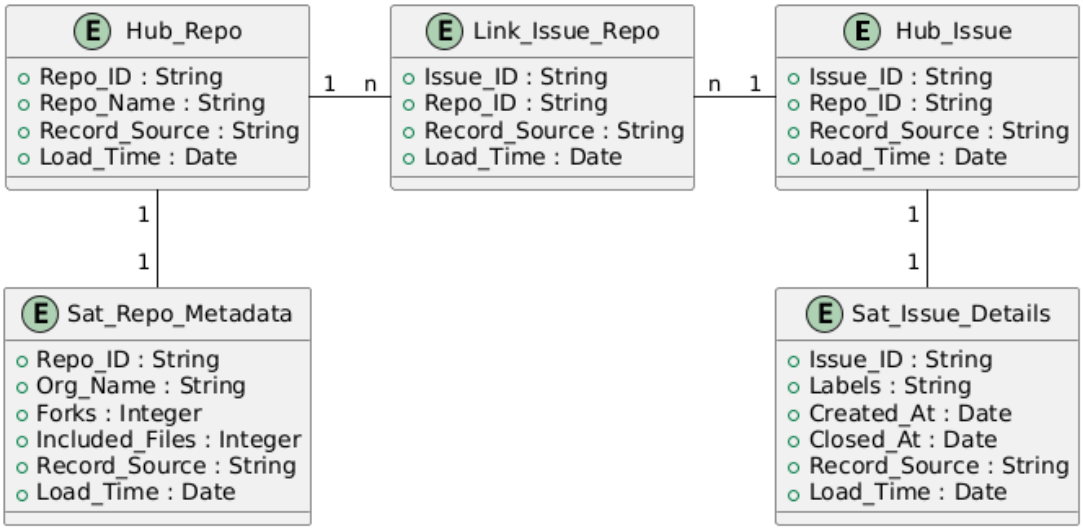


Figure 4.13: Excerpt of a Data Vault for Git Development Data

Star Schema as Model of the Gold Layer

The Star Schema was developed for intuitive, high-performance querying of data warehouses for BI-application. Unlike to Third Normal Form (3NF)-normalized relational database systems, the Star Schema is denormalized which offers the advantage to perform queries with fewer joins. Star Schemas, also referred to as dimensional models, are star-like structured. They are composed of a central `Fact` table that is connected to multiple `Dimensional` tables. Fact tables store performance measurements such as metrics that are acquired from an organization´s business processes. Each Fact corresponds to a business measure. Dimensional tables hold attributes describing the context of the measurement events from the Fact tables. Each Dimension table has a unique primary key that can be referenced in Fact tables to ease joining of Fact and Dimension tables. (Kimball & Ross, 2013)

Modeling the Gold layer of the data lakehouse with a Star Schema offers the advantage that the solution enables the integration of BI applications such as PowerBI that benefit from a Star Schema data model. This would allow customers to explore business-oriented aggregates that are situated at the Gold layer on their own.

The transformation from the Data Vault model (Silver layer) to the Star Schema model (Gold layer) is non-trivial. Dimensions contain most of the descriptive attributes and can primarily be derived from Satellites, neglecting columns that are not of direct interest to users. Facts represent business measures which can be derived from Hubs and Links. Puonti and Raitalaakso (2019) describe a systematic mapping approach for transforming a Data Vault model into a dimensional model, which can be partially automated.

Figure 4.14 illustrates an excerpt of a Star Schema for software development data from Git. The `Fact_Issues` Fact table references `Dim_Repo` and `Dim_Issue` that contain the attributes describing the Fact. Based on a Fact, aggregations can be created, whose result can be stored in other Facts. For example, the `Fact_Issue_Stats` summarizes issue metrics and is calculated based on the Fact `Fact_Issues`.

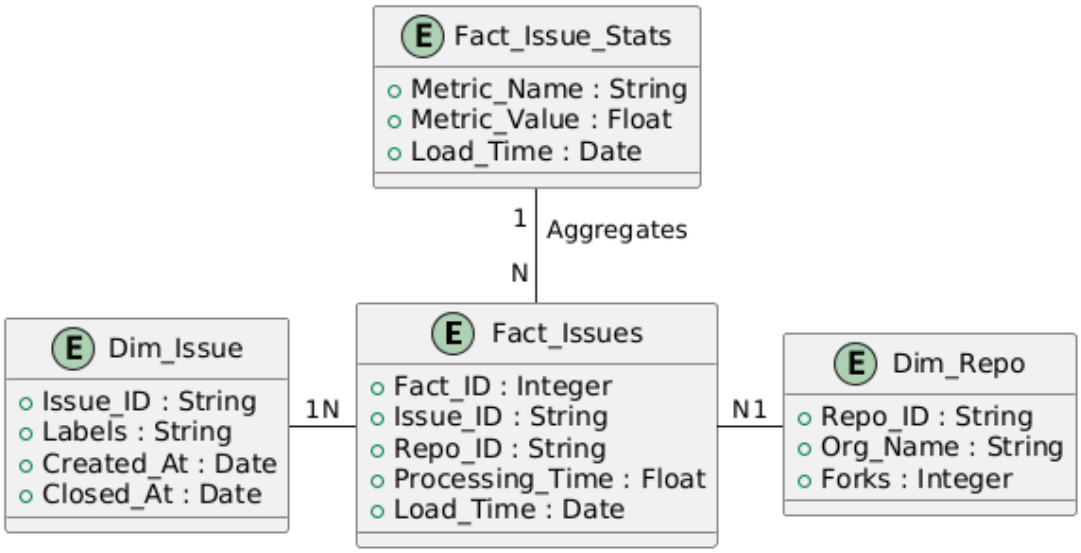


Figure 4.14: Excerpt of a Star Schema for Git Development Data

5 Implementation

Chapter 5 targets the implementation of a prototype of the designed solution presented in chapter 4. Initially, used technologies are introduced. Subsequently, it is presented how the prototype is implemented and how the application can be deployed.

Although section 4.3 introduced both, a single-tenant and multi-tenant solution strategy, the implementation of the prototype focuses on a single-tenant solution. The aim is to demonstrate that the overall data lakehouse approach is feasible, while neglecting the additional complexity, which multi-tenancy introduces. Furthermore, the prototype concentrates on building the data lakehouse itself. It does not yet focus on providing functionality for configuration management or providing user-friendly insights to users. Consequently, the Configuration Service, Analytics Service, Frontend and the Gold layer have not yet been implemented.

5.1 Technology Stack

To realize the prototype, the existing solution of the development team, on which this thesis builds, is enhanced. The existing system already features the following main technologies that are reused:

- **Apache Spark:** The solution uses Apache Spark with its Python interface PySpark (The Apache Software Foundation, 2025b) to process large amounts of data efficiently.
- **Delta Lake:** The Delta Lake storage framework (Armbrust et al., 2020) is used as foundation of the data lakehouse. Delta Lake is a storage layer that supports ACID transactions on Apache Parquet files.
- **Docker Compose:** For containerized deployments Docker Compose (Docker Inc., 2025) is used to run all the application's components across multiple containers.

As discussed in 4.3.1, implementing the desired data lakehouse necessitates additional technologies for data storage, scheduling and orchestration. The implementation therefore adopts the usage of an S3-compatible object store for the data storage layer and Apache Airflow for scheduling and orchestration of the system.

- **S3-compatible Object Store:** There exists various object stores that are compatible to the AWS S3-API (Amazon Web Services, 2024). In this thesis, the S3-compatible object stores Zenko (GitHub Repository scality/cloudserver, 2025) and SeaweedFS (GitHub Repository seaweedfs/seaweedfs, 2025) were evaluated. Both storage solutions are Apache 2.0-licensed.
- **Apache Airflow:** Apache Airflow (The Apache Software Foundation, 2025c) is one of the most commonly adopted solutions for scheduling and orchestration of workflows.

5.2 Development of a Prototype

5.2.1 S3-compatible Object Store as Data Lakehouse Storage

A data lakehouse requires a scalable storage solution, because large amounts of data needs to be processed. The solution favors using an object store that is compatible to the AWS S3-API.

Apache Spark and Delta Lake support reading and writing to S3-compatible object stores that may be hosted on different machines. Specifically, Spark uses the `S3a://` interface. Setting up Spark using an S3-like object store, requires providing Spark with credentials and the correct endpoint, at which S3 is available, and ensuring a connection is possible.

First, the use of Zenko, which states to provide an S3-compatible interface (Zenko, 2025), was evaluated. It can be deployed locally with a provided Docker Image. However, Delta Lake's reliability depends on comprehensive support of all S3 semantics due to the eventually consistent nature of distributed object stores (Armbrust et al., 2020). Executing Delta Lake with Zenko leaded to a `Hypertext Transfer Protocol (HTTP) 501 - Not implemented` error at the Zenko webserver in a test setup, which indicates that Delta Lake requires S3 semantics that Zenko does not support.

Since the evaluation of Zenko showed that Zenko does not meet the requirements to be used as the data lakehouse object store, SeaweedFS was considered as another alternative. The Configuration of SeaweedFS requires to set up credentials

and to create a bucket, which corresponds to a folder in S3. For the local deployment, a provided Docker image was used as well. Integration tests verified that Delta Lake is able to work with SeaweedFS.

5.2.2 Transformation of Development Data to a Data Vault

In 4.3.4 the Data Vault was introduced as data model of the Silver layer in the data lakehouse. Table 4.15 in this subsection has already outlined important steps of transforming software development data to a Data Vault, including creating Hub tables for business objects, creating Link tables for relationships between business objects and creating Satellites for contextual information.

Mapping of Data Sources to Hub, Satellites and Links

Transforming software development data to a Data Vault requires pursuing a clear schematic approach. The algorithm, that was created as part of the prototype, offers the possibility to define the mapping of data sources to Hubs, Links and Satellites in a single Yet Another Markup Language (YAML)-file. Figure 5.1 illustrates an excerpt of this structured YAML-file to configure the mapping.

```yaml
Sources:
  - name: commit
    entities:
      - hub_name: hub_commit
        business_keys:
          - sha
        satellite_name: sat_commit
        attributes:
          - commit_message
          - commit_author_date
        attribute_renames:
          commit_author_date: commit_authored_date
      - hub_name: hub_user
        business_keys:
          - author_login
        satellite_name: sat_user_author
        attributes:
          - author_id
          - author_name
    links:
      - link_name: link_commit_author
        hubs:
          - name: hub_commit
            business_keys:
              - sha
          - name: hub_user
            business_keys:
              - author_login
```

Figure 5.1: Excerpt of the YAML configuration to map a commit to Hubs, Links and Satellites

As presented in figure 5.1, each data source can contain multiple entities. Consider a `commit` of the data sources GitHub or GitLab. Trivially, parts of the `commit` can be resolved to an entity `entity_commit`. This entity can include information of the `commit`, such as `commit_message` and `sha`. `sha` is an unique identifier of a `commit`. For this reason, it can be used as business key to identify the entity `entity_commit`. Based on the business object `entity_commit`, a Hub table `hub_commit` can be created that includes the business key `sha` to identify single `entity_commit`s. Furthermore, the contextual information `commit_message` must be stored. Therefore, a Satellite table `sat_commit` can be created, which includes the `commit_message` and a foreign key to the Hub table `hub_commit`.

However, the original `commit` of the data source is composed of multiple entities. In addition to the entity `commit_entity`, the `commit` contains another business object `user_entity`. It contains information, such as an `author_name`, which represents the name of the author that was responsible for the commit. This `user_entity` must be divided into another Hub table `hub_user` with an associated Satellite table `sat_user_author`.

Still, the relationship between the entities `hub_commit` and `hub_user` must be stored. With the business keys of both entities, an additional Link table to represent the relationship between the business objects `link_commit_author` must be created. This Link table must contain both business keys of `hub_commit` and `hub_user` as foreign keys to the associated Hub tables.

Structure of the Composed Data Vault Tables

Previously was shown, how an example data source composed of multiple entities can be mapped to Hubs, Links and Satellites. However, creating a Data Vault table involves additional steps. To support traceability textttrecord_source and `load_date` should be added to each table in the Data Vault model. On top of that, each table requires a primary key, which can be generated by hashing business keys.

Different data sources may share common entities. GitHub and GitLab commits share the same concept of the entity `commit_entity`. GitHub and GitLab commits can be combined by configuring the same Hub `hub_commit`. If columns are missing while merging a common entity of two data sources, these columns can be appended. The result is a data model that combines multiple data sources around the shared concept of business objects.

In same cases, different data sources contain columns, which share a common semantic meaning with columns of other data sources. However, their column name is likely to differ as different data sources do not share the same data model. To gain a more compact and meaningful Data Vault `attribute_renames` were introduced as another configuration option. If both columns share the same

column name, they can be merged. In the example of figure 5.1, the attribute `commit_author_date` is renamed to `commit_authored_date`. The latter represents the column name of a column with the same semantic meaning of another data source.

5.2.3 Scheduling and Orchestration with Apache Airflow

Apache Airflow is able to orchestrate multiple jobs, which can be composed into a workflow, referred to as DAG (Directed Acyclic Graph). Each workflow must be a directed acyclic graph in order to ensure that execution plans are achievable and do not contain cycles. Jobs within a workflow can be of different types. Apache Airflow supports the orchestration of various job types, including Apache Spark, Kubernetes and the invocation of simple Bash scripts.

The data lakehouse is built upon Spark. Therefore, for each job of a workflow, a `SparkSubmitOperator` has to be configured. The `SparkSubmitOperator` must especially include the following key information:

- `conn_id`: Airflow requires a connection to the Spark master, that is responsible for the execution of Spark jobs. This connection can be configured in Airflow using the Airflow Admin UI and involves providing the endpoint of the Spark Master.
- `application` The location of the source code of the application, such as a Python script, which is triggered by the `SparkSubmitOperator`.
- `py_files` The location of all Python dependencies that are required for the execution of the application. Therefore, these dependencies can be packaged into an archive and mounted into the Apache Airflow container. Then, a reference to the archive inside of the container is provided.
- `conf` Additional Spark configurations, such as specifying Java Archive (JAR) files that are required for the execution of Spark.

After setting up all `SparkSubmitOperator`s, their execution order has to be defined. If correctly configured and the workflow is executed, Airflow submits the provided Spark applications to the Spark master for execution in the defined order.

5.3 System Deployment

After pointing out how the prototype was developed, this section discusses further the deployment of the created system.

The system deployment is realized using Docker and Docker Compose. Although it would be possible to deploy the system as a whole within one multi-container

application, the deployments of SeaweedFS and Apache Airflow were separated to allow deploying them independently on possibly different hosts. However, as SeaweedFS provides the object storage solution that holds the data lakehouses data, it must be accessible from the Apache Airflow installation. This can be achieved using a Docker network.

5.3.1 SeaweedFS

SeaweedFS features a ready-to-use Docker Image that does not require major adjustments. After a first startup, a user role for `Spark` with appropriate credentials has to be created. These credentials must be made available to Spark workers, that require to use the SeaweedFS object store. Additionally, one or multiple buckets have to be created, in which Delta Lake can store tables. In the prototypical installation, this was realized using the AWS Command Line Interface (CLI) (Amazon Web Services, Inc., 2025), a client tool, that can also be adopted to communicate with other S3-compatible object stores than AWS S3. The prototype stores all tables in a single bucket.

5.3.2 Apache Airflow

Beside other components, the Apache Airflow Docker Image contains an Airflow webserver, that is responsible for hosting the Airflow UI, a Scheduler for scheduling tasks and Airflow workers that are responsible for the execution of these tasks. A dedicated database ensures that configurations, that are applied to the Airflow container, survive restarts of the webserver and workers.

Although Apache Airflow can also be connected to Apache Spark clusters, the prototype focused on a local installation for simplicity. For this reason, the Docker Image of Apache Airflow was extended with a Spark installation and additional dependencies, that are required to run Spark and the prototype with Airflow. This includes the installation of a specific Spark customization of the Apache Airflow Image. Furthermore, after the first startup of Apache Airflow, a Spark connection between Apache Airflow and Spark needs to be configured. The Apache Airflow installation has to be secured by defining appropriate credentials.

Additionally, as already stated in 5.2.3, the created DAG, the application and all required dependencies of the application, have to be mounted into the Airflow container. Unfortunately, this means that whenever source code of the application is changed, these changes have to be made available to Airflow by packaging the source code into an archive and mounting the generated archive into the Airflow container. It would be beneficial to automate this using Continuous Integration (CI)/Continuous Delivery (CD) pipelines.

6 Demonstration

Based on the developed prototype, chapter 6 demonstrates that the solution works as expected by presenting the prototype's execution using software development data.

6.1 Data Sources for Demonstration

The used test dataset consists of development data of three different data sources. It contains data from one GitHub repository, two GitLab repositories and two Jira projects. Data is retrieved directly by requesting publicly available APIs of the data sources. Multiple APIs are called to retrieve different types of data for each data source, such as `commits` or `issues` from a GitHub repository. Table 6.1 summarizes, which data types from which data source are included in the test dataset:

GitHub	**GitLab**	**Jira**
• `github_commits`	• `gitlab_commits`	• `jira_issues`
• `github_events`	• `gitlab_events`	• `jira_history`
• `github_issues`	• `gitlab_issues`	
• `github_releases`	• `gitlab_releases`	
• `github_jobs`	• `gitlab_jobs`	
• `github_pipelines`	• `gitlab_pipelines`	
• `github_sbom`		

Table 6.1: Overview of Software Development Data Included in the Test Dataset

6.2 Test Deployment of the System

To verify that the system works as expected, it first has to be deployed. As already discussed in section 5.3, the prototype's deployment was realized using

Docker Compose.

For the test setup, a separate Ubuntu Virtual Machine (VM) with a Docker, Java and Python installation was created. On this virtual machine, separate Docker deployments of SeaweedFS and Apache Airflow were achieved. To allow communication between both Docker installations, a common Docker network was created. All other applied steps have already been discussed in section 5.3.

Figure 6.1 illustrates the resulting deployment view of the prototype. Inside an Ubuntu VM a Docker container for SeaweedFS and another for Apache Airflow are deployed. Both Docker containers are part of a Docker network to allow communication between Spark and SeaweedFS. Inside Airflow, the Airflow Scheduler is responsible for assigning tasks to workers according to workflows, that are defined as DAGs. Data ingestion and integration jobs are executed by Airflow workers that run Spark and communicate with SeaweedFS as storage layer of the data lakehouse.

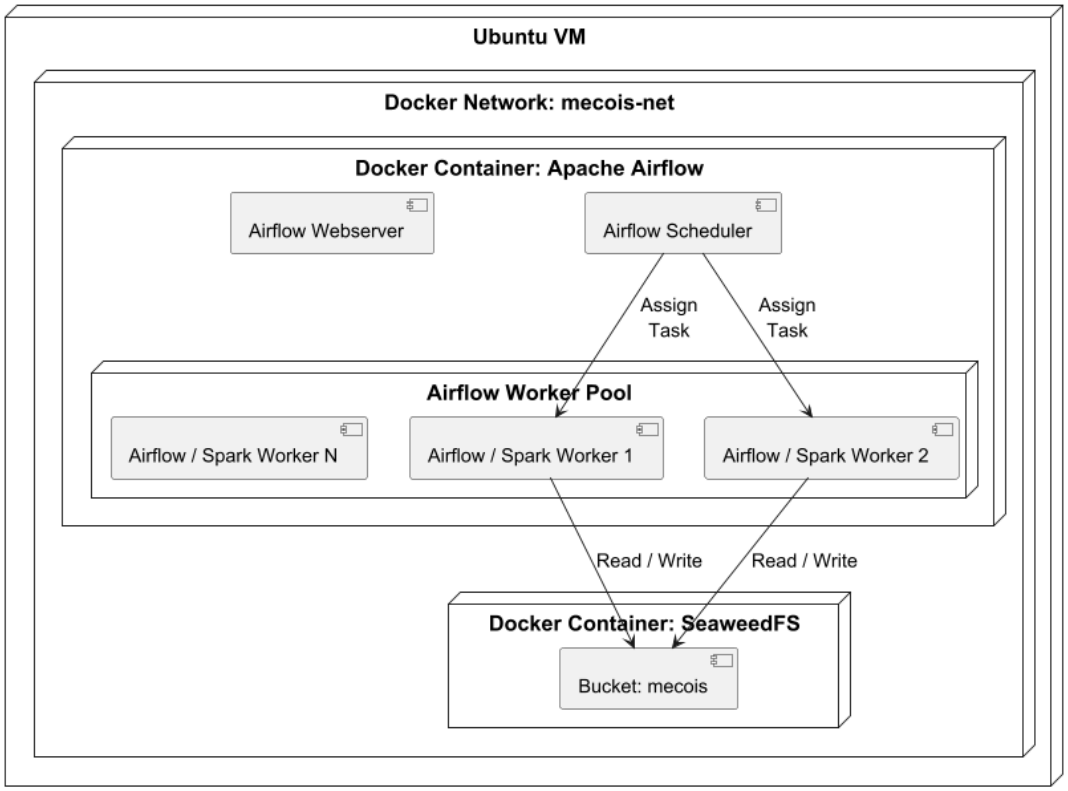


Figure 6.1: Deployment View of the Test Setup for Demonstrating the Prototype

6.3 Execution of the Application and Example Scenarios

To conclude the demonstration, it is shown that the developed prototype works as expected by executing the application with the test dataset. Therefore, data of the test dataset was first ingested to the Bronze layer. Subsequently, data integration jobs executed cleansing steps such as unifying date formats and removing not needed columns. Afterwards, the software development data was transformed to a Data Vault-based Silver layer according to the specified schema.

6.3.1 SeaweedFS as Data Lakehouse Storage

SeaweedFS builds as cloud object store the foundation of the data lakehouse. This distributed object store is responsible for reliably storing Delta tables and making the data accessible to multiple workers for distributed computing.

To verify that SeaweedFS works as expected and Spark jobs correctly store data into Delta tables in this S3 compatible storage, the AWS CLI was used. The prototype stores all Delta tables into the same bucket named `mecois`. With the AWS CLI it was verified that this bucket contains:

- Apache Parquet files of Bronze tables of the Medallion architecture. This demonstrates that data can be correctly ingested.
- Apache Parquet files of Silver tables of the Medallion architecture. This demonstrates that data can be correctly integrated to the Silver layer.
- Delta Lake log files, which Delta Lake requires for transactional processing, to ensure Delta Lake can work as expected in SeaweedFS.

The test was conducted after all data of the test dataset as described in 6.1 was ingested and integrated until the Silver layer of the Medallion architecture. Figure 6.2 lists the commands that have been executed with the AWS CLI and the responses of SeaweedFS.

```
# Listing - Sample Bronze layer Apache Parquet files of a Table for Jira
    development data

powershell -Command "aws --endpoint-url http://localhost:8333 s3 ls s3://
    mecois/MECOIS/jira_issues/bronze --recursive --profile seaweedfs |
    findstr '.parquet' | Select-Object -First 2"

2025-04-16 18:19:39      46562 MECOIS/jira_issues/bronze/part-00000-b575d274
    -3965-44e0-b86b-d1e90b351555-c000.snappy.parquet
2025-04-28 19:58:11     403273 MECOIS/jira_issues/bronze/part-00000-cafa22c7
    -c54a-4eaa-99db-715f3fad7738-c000.snappy.parquet

------------------------------------------------------------

# Listing - Sample Silver layer Apache Parquet files of a Table for GitHub
    development data

powershell -Command "aws --endpoint-url http://localhost:8333 s3 ls s3://
    mecois/MECOIS/github_commits/silver --recursive --profile seaweedfs |
    findstr '.parquet' | Select-Object -First 2"

2025-04-16 17:25:57     466397 MECOIS/github_commits/silver/part-00000-
    0797015e-96a1-4a76-9936-22a9a1f887ae-c000.snappy.parquet
2025-04-28 19:08:25     466397 MECOIS/github_commits/silver/part-00000-2
    c39c6fb-dddc-4a5f-a35f-0e6c024890c6-c000.snappy.parquet

------------------------------------------------------------

# Listing - Sample Delta Lake transaction log files

powershell -Command "aws --endpoint-url http://localhost:8333 s3 ls s3://
    mecois/ --recursive --profile seaweedfs | findstr '_delta_log' | findstr
     '.json' | Select-Object -First 2"

2025-04-16 16:46:21       1856 MECOIS/data_vault/hub_commit/silver/_delta_
    log/00000000000000000000.json
2025-04-16 16:50:07       2074 MECOIS/data_vault/hub_commit/silver/_delta_
    log/00000000000000000001.json
```

Figure 6.2: Demonstration of using SeaweedFS as Storage for Delta Lake files using the AWS CLI

6.3.2 Transformation of Development Data to a Data Vault

The previous demonstration of SeaweedFS has already shown that the solution can process data between tables of different layers correctly. Additionally, it was assessed, if the transformation of development data to the Data Vault based Silver layer works as expected. Therefore, Apache Parquet files were retrieved from SeaweedFS, which represent parts of the following tables:

- Bronze layer GitHub Commit table
- Silver layer Hub Commit table
- Silver layer Satellite Commit table that is associated to the Hub Commit table

- Silver layer Link Commit Author table that defines a relationship of the tables Hub Commit and Hub Author

To verify that data was transformed correctly, the table content of the mentioned tables have to be checked. For this reason, the named Apache Parquet files were imported to Microsoft's PowerBI and explored.

Figure 6.3 depicts a visualization of the data created with PowerBI that shows selected table contents of the `Bronze GitHub Commit` table and the generated Silver layer Hub, Link and Satellite tables. All transformation steps did work as expected. `Link Commit User` and `Satellite Commit` contain the primary key `hk_hub_commit` of the `Hub Commit` table as foreign keys. The `Hub Commit` table contains the `sha` of the commit with the message "Create sprint-11 folder" as the `sha` represents the business key of the commit. This commit message can also be found in the `Satellite Commit` as this Satellite contains the contextual data associated with business object commit.

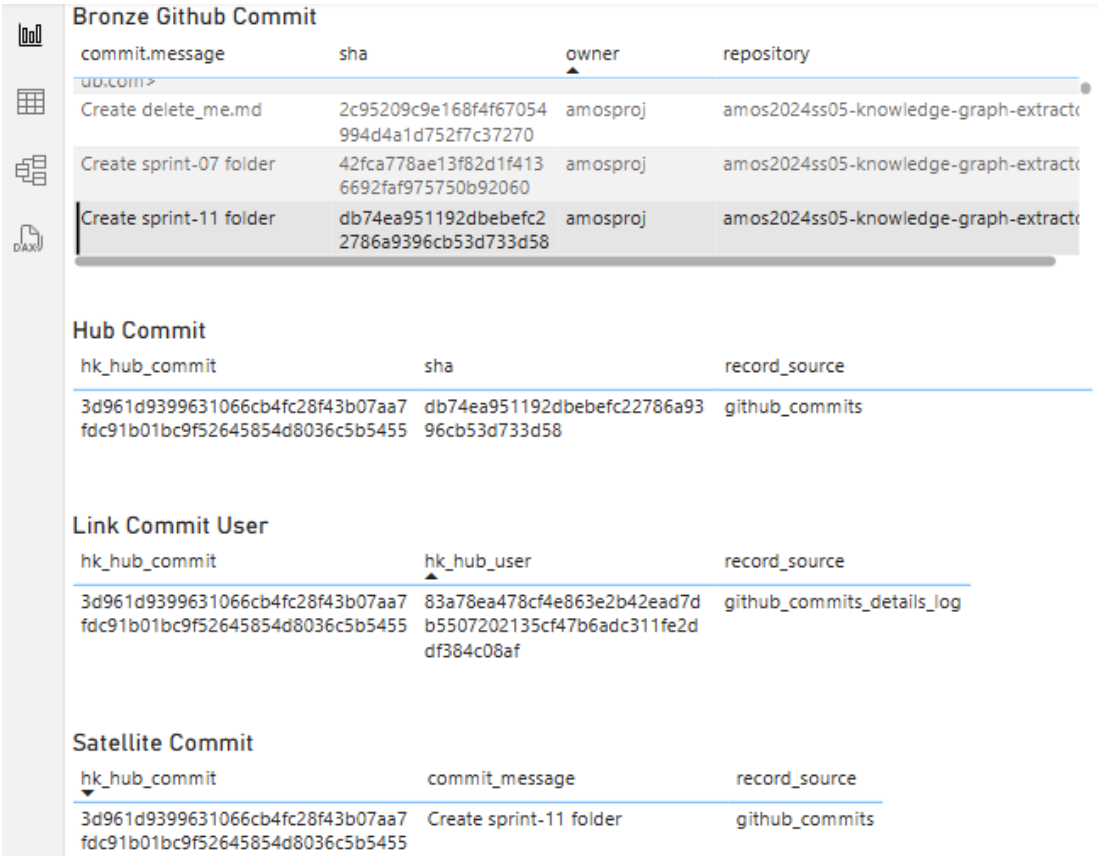


Figure 6.3: PowerBI Visualization of Bronze and Silver Tables to Demonstrate the Correct Transformation of GitHub Commit Data into the Data Vault Model

6.3.3 Scheduling and Orchestration with Apache Airflow

To demonstrate that also scheduling and orchestration with Apache Airflow works as expected, a DAG consisting of three data ingestion jobs and one data integration job was created. Ingestion jobs are divided by data source: one for GitHub, one for GitLab and one for Jira. The workflow executes all three data ingestion jobs in parallel. After all data ingestion jobs have been finished, Airflow executes the data integration job, which is responsible for integrating the ingested data to the Silver layer of the data lakehouse.

Figure 6.4 presents this DAG to ingest software development data from GitHub, GitLab and Jira to the Silver layer of the Medallion architecture in the Apache Airflow UI. In the depicted state, `ingest_jira_data` has finished and waits for the completion of the jobs `ingest_gitlab_data` and `ingest_github_data`. Once all three data ingestion jobs have been completed, Airflow will start the `integrate_to_silver` job to integrate the data into the Silver layer.

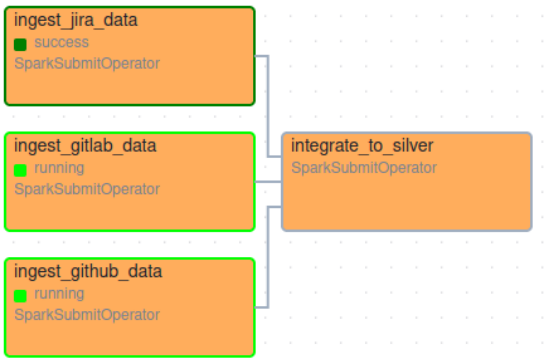


Figure 6.4: DAG that Orchestrates Multiple Ingestion Jobs and an Integration Job in Apache Airflow

Apache Airflow offers various management capabilities for this DAG. A larger excerpt of the Airflow UI for this `development-data-to-silver-dag` is shown in figure 6.5.

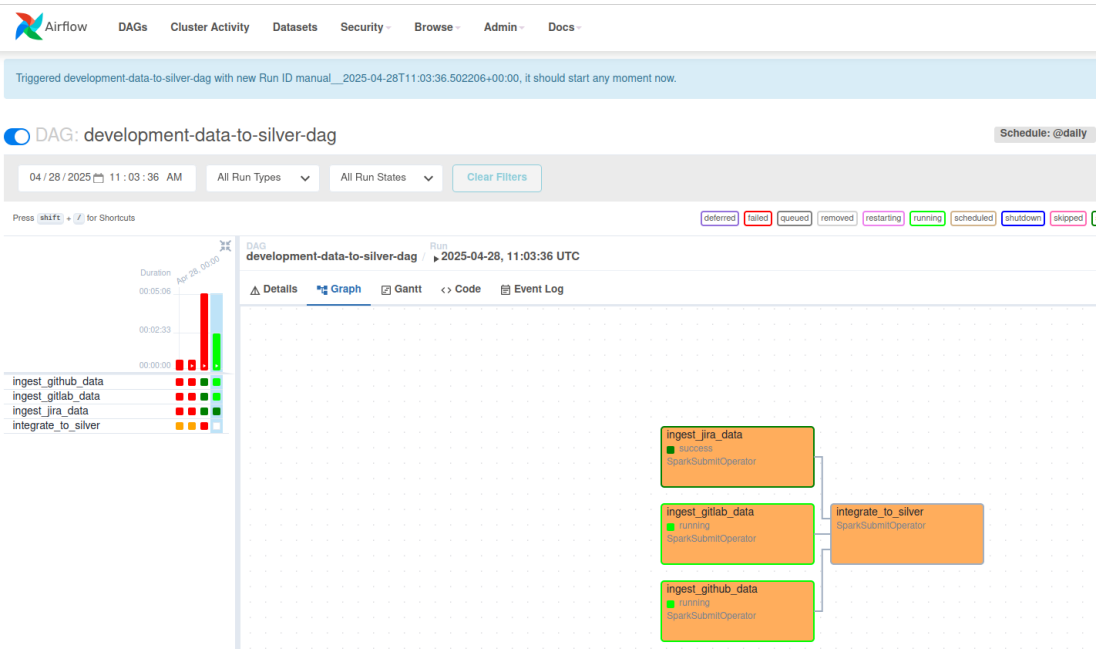


Figure 6.5: Demonstration of a DAG in the Apache Airflow UI

7 Evaluation

To evaluate the designed software architecture, an informal walkthrough with an experienced software architect was conducted after a draft of the solution design had been completed. The walkthrough began with a presentation of the system's objectives, key requirements and relevant architectural constraints. Afterwards, the business and technical context of the application were described. Subsequently, both the single-tenant and multi-tenant solution were first presented from a high-level point of view and later concretized from different architectural viewpoints, including the level-one building block view, runtime view and deployment view.

The result of the evaluation is that the software architecture offers the possibility to process large amounts of software development data. The building blocks were comprehensively sliced according to the separation of concerns principle and the usage of technologies such as Apache Spark and Kafka allows for scalable processing of software development data. Also, the usage of Apache Airflow seems to be a suitable solution to orchestrate data lakehouse jobs in the single-tenant architecture. However, several limitations were identified. Especially the need for multi-tenancy was questioned. It was argued that different organizations have heterogeneous data and that it may be practically challenging to integrate development data from multiple organizations that may have different schemes. Furthermore, it was questioned whether the additional complexity introduced by the multi-tenant mode in terms of orchestration and security is really necessary. As for single-tenancy with Apache Airflow, an industry-proven solution exists, it was recommended to initially focus on implementing a single-tenant solution. Additionally, the architecture included a data mart (PostgreSQL) for fast access to pre-aggregated analytical insights. While this enables low-latency fast queries, it introduces duplicate data storage. Therefore, it was recommended that this additional complexity should only be introduced, if strict performance requirements cannot be met through directly querying the data lakehouse.

Also it was noted that for most software development analytical tasks, a two-tier data lake architecture, consisting of landing and curated zone can be sufficient. However, the review took place before data organization was applied to multiple

layers including a Data Vault-scheme for the Silver layer and a star scheme for the Gold layer.

With the feedback of the walkthrough, the software architecture was revised and findings of the walkthrough were incorporated into the design. Especially the trade-offs associated with multi-tenancy and the inclusion of a data mart for fast queries on basis of a relational database were critically assessed. Consequently, the software architecture draft was adapted and it was further discussed that multi-tenancy should only be implemented if strictly required. The data mart component was designed as optional. It was noted that it should only be included if strict performance requirements cannot be met without it. The introduction of a star-scheme at the Gold layer of the data lakehouse also decreases the need for a separate data mart for low-latency queries, as the star-scheme supports fast low-latency queries by reducing join operations. The trade-offs mentioned have already been discussed in more detail in section 4.2.

8 Conclusion

Aggregating and analyzing software development data is challenging as development data is available from lots of data sources such as repositories, ticketing systems and communication channels with heterogeneous data. Handling the partially semi-structured and unstructured data with traditional data warehousing approaches is difficult while data lakes often suffer data quality issues. The objective of this thesis was to bring the concept of modern data lakehouse architecture, which has received increasing attention in the scientific community in recent years, into the field of software development analytics.

This thesis builds upon the work of an existing development team that aims to obtain insights into the software development process in the domains productivity measurement, transfer pricing and impact assessment. The existing architecture was evolved to a complete data lakehouse architecture, addressing the need for a scalable analytics platform with high data quality that is capable of providing valuable analytical insights into the software development process by correlation of data from various data sources. To ease the solutions adoption, only permissively licensed open-source components have been used.

The solution offers the possibility to integrate new data sources and is targeted to be used within different organizations. While the prototypical implementation focused on developing a working prototype that is able to be used by one single tenant, the extension of the software architecture to multi-tenancy was an important goal in the design phase. Key design decisions are document using industry-oriented Architecture Decision Records (ADRs). The ADRs state in which context the decision was made, what alternatives were considered, the decision itself and which consequences follow. Multi-tenancy inherits additional complexity as a tenants data must be protected from the access of other tenants and processed securely. Therefore, a multi-tenant enhancement of the system is only suggested, if this is strictly required and multiple deployments for different tenants are no longer an option.

After the initial design phase, an evaluation of the proposed software architecture with an experienced software architect was conducted in the form of a walk-

through. Feedback from this evaluation was incorporated into the final design of the solution.

As a method of choice a design science approach for information systems was conducted that led to the development of a working prototype of the designed solution. The prototype extends the existing system of the development team with orchestration via Apache Airflow, a scalable S3-compatible object store as data lakehouse storage and a Data Vault-based Silver layer. The prototypes applicability was demonstrated using development data from GitHub, GitLab and Jira.

8.1 Addressing the Research Questions

In the introduction in section 1.2 four research questions have been defined. This section summarizes how the research questions are addressed in the thesis.

1. What are the core logical components required for a system that aggregates and analyzes software development data?

As core logical components Data Ingestion, Data Integration, Configuration Handling, Processing of Analytical Queries, and Scheduling and Orchestration were identified. These logical components shape the designed software architecture and form the foundation of a scalable and flexible solution to analyze software development data from heterogeneous data sources.

2. Which challenges do the logical components of such a system need to address?

Each logical component is responsible for handling at least one major complexity in the data lakehouse:

- **Data Ingestion:** Data is accessible from various data sources in different serialization formats and may be structured, semi-structured or unstructured.
- **Data Integration:** Data with a common semantic meaning has different data models across data sources and must be unified into a common data model. This data model must support efficient query handling to support analytical queries.
- **Configuration Handling:** The data lakehouse needs to be customizable to stakeholder needs. Different organizations require different ingestion intervals or need to adjust data sources.
- **Handling of Analytical Queries:** Both fast queries to access pre-computed aggregations such as metrics, and complex long-running asynchronous quer-

ies needs to be supported.

- **Scheduling and Orchestration:** Jobs running in the data lakehouse must be scheduled and orchestrated.

3. How would a suitable single-tenant data lakehouse architecture for aggregating and analyzing development data look like?

A modular, scalable and extensible data lakehouse architecture for software development analytics, based on Apache Spark, Delta Lake, and S3, which is orchestrated via Apache Airflow, was developed. The solution design follows the Medallion Architecture, including a structured Data Vault-based Silver layer and a Star Schema-based Gold layer. A prototype of the solution was implemented and successfully verified using software development data from GitHub, GitLab and Jira.

4. How can the single-tenant architecture be enhanced to support the aggregation and analysis of development data from multiple tenants?

The thesis discussed in detail the extension of the created single-tenant architecture to multi-tenancy. Multi-tenancy introduces additional complexity, such as ensuring that a tenant's data is isolated from the access of other tenants. The architecture addresses this additional complexity by adjusting the orchestration and configuration handling. However, the implemented prototype remains single-tenant, and multi-tenancy should only be pursued if operating separate deployments becomes infeasible.

8.2 Main Contributions

This thesis contributes the software architecture of a modular and flexible data lakehouse for software development analytics. The solution can be extended to multi-tenancy and supports further analytical research in this domain.

The main contributions of this thesis are:

- The design of a modular, scalable and extensible software architecture for a data lakehouse that supports software development analytics.
- The discussion of how the data lakehouse can be enhanced from a single-tenant, on-premises solution to support multi-tenancy.
- A working prototype that builds upon an existing solution, enhanced with scheduling and orchestration via Apache Airflow, an S3-compatible object storage and a Data Vault-based Silver layer.
- The documentation of the designed software architecture, which can be found in appendix A, using the industry-proven arc42 template.

8.3 Limitations and Future Work

Several limitations of this work have been identified and present opportunities for future work. Although it was extensively discussed how the existing system can be evolved to multi-tenancy, the implemented prototype remains single-tenant. The solution's applicability in a multi-tenant context must still be validated by implementing the proposed multi-tenant scheduling and orchestration concept for jobs running in the data lakehouse. However, as previously stated, this introduces further complexity and should only be pursued if strictly required.

Furthermore, the prototype was deployed using Docker containers on a single virtual machine and tested with a small dataset. While horizontal scalability was an import concern in the architectural design, this characteristic has not yet been demonstrated by deploying and executing the system on multiple nodes in a distributed setup. Despite Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Container as a Service (CaaS) have not yet been discussed in detail, used technologies are compatible with Kubernetes and support cloud-native deployments in principle. Designing and evaluating a Kubernetes based deployment architecture for the solution would be another promising topic for future research. Additionally, evaluations of the performance and stress testing would be beneficial to validate the scalability and efficiency of the system. Monitoring and observability of the data lakehouse also remain open topics that could benefit from basic observability features that are offered by Apache Airflow.

The solution was designed to present valuable insights to users. The working prototype is capable of correlating data from multiple data sources. Yet, it still lacks a user interface and the associated API of the Analytics Service that allows non-technicians to explore the gained insights into the software development process. Furthermore, functionality for configuration management and the Gold layer have only been designed, but not yet implemented.

Another promising topic for further research is that the solution design would allow to provide data from the Gold layer to other external business intelligence solutions, such as PowerBI. Creating semantic models for these BI tools, that explain how the data in the data lakehouse should be interpreted, would allow organizations to explore calculated metrics with existing business intelligence tools and may promote the solutions adoption.

Appendices

A Software Architecture Documentation

To document the designed solution in detail the following software architecture documentation was created as part of this thesis. The software architecture documentation follows the arc42 standard and was partially enhanced with the support of AI tools. The documentation was originally written in Markdown and later converted to PDF to integrate it into the thesis. I apologize for any minor appearance related issues that may occur due to the automated conversion from Markdown to PDF. Some diagrams and sections that are part of the software architecture documentation also appear in the main part of the thesis.

Software Architecture Documentation

1. Introduction and Goals

Mecois (https://www.mecois.com) is gathering software development data from various sources, combines this data and gains valuable insights into the software development process from correlation.

The aim of Mecois is to gain knowledge from development data especially in the following domains:

- **Productivity Measurement** Measure and Monitor the productivity of software development teams
- **Transfer Pricing:** If software is developed in different organizational units across tax boundaries, source code must be legally priced and taxed. Mecois aims to determine the transfer prices (compensation payments)
- **Impact Assessment:** Assess the monetary impact of work of development teams by correlating lines of code with the value that these lines create

For details see https://www.mecois.com

1.1 Requirements Overview

Main Functional Requirements

The main purpose of the application is to gain knowledge from development data of various sources by correlation. This requires the following features in particular:

Label	Requirement	Description
FR1	Data ingestion of software development data	The system must support data ingestion from multiple sources such as GitHub, GitLab, Jira, Confluence, Microsoft Teams, etc.
FR2	Extract, transform, and normalize data	The system must be able to process raw data of various sources and normalize and transform it into a common structured format to support further analysis steps.
FR3	Correlate data from different sources	The system must be able to identify related entities across different data sources and link these entities together to provide advanced insights through data correlation.

Label	Requirement	Description
FR4	Gain business-oriented insights	By advanced data correlation the system should provide insights into the development process to stakeholders. Different stakeholders are interested in different insights of different domains, such as work-time calculations, productivity metrics, and financial impact assessments.
FR5	Present insights to stakeholders	The system must offer a visulisation of gained insights via dashboards, reports, and APIs to end users.

Multi-Tenancy

The final system should be able to analyze software development data from multiple organizations (or from distinct projects). Therefore, the system must work for multiple tenants, which introduces further complexities:

Label	Requirement	Description
FRMT1	Data isolation per tenant	Data of different tenants is isolated from each other. Accessing data of other tenants is not possible. Data from different tenants must not be combined or correlated.
FRMT2	Security and Access Control	The system ensures that a tenant can only access data which belongs to the tenant. A tenant must not be able to access data of other tenants.

The system is designed to evolve from a single-tenant to a multi-tenant system. For that reason, also the single-tenant architecture is discussed. Two modes of the system are considered:

- **Single-tenant mode:** Separate OnPremises installation of the system and isolated system use. Reduced complexity as Security requirements such as data isolation per tenant are of no concern.
- **Multi-tenant mode:** The system runs on a shared infrastructure (e.g. Cloud environment) and is used by multiple organizations or distinct projects. Access control and data isolation per tenant are key requirements.

1.2 Quality Goals

Label	ISO 25010:2023 Quality	Motivation
QG1	**Functional Suitability**	The system must provide realistic insights into the software development processes. Therefore, the system must ensure correct data ingestion, integration and analytics processes to gather meaningful insights. There should be a possibility to revert incorrect inserts (data ingestion).
QG2	**Performance Efficiency**	The end user should be able to see business insights fast even if this requires pre-calculation. The system should be horizontally scalable and able to handle large volumes of data. Data ingestion and integration should work in a reasonable amount of time and can be scheduled.

Label	ISO 25010:2023 Quality	Motivation
QG3	**Reliability**	Reports should provide consistency. Business insights to the end user on dashboards can be eventually consistent. It is sufficient to calculate resource-intensive evaluations (e.g., once a day) and display the pre-calculation to users.
QG4	**Compatibility**	New data sources can be added easily and the schema of existing data sources can be evolved.
QG5	**Flexibility**	Tenants can configure which data sources and to some extent which parts of a data source are to be used. The system should be able to run in cloud computing environments, but also OnPremises installations on servers should be possible.
QG6	**Security**	Credentials and sensitive data should be processed securely. Data of a tenant should be protected from the access of other tenants.
QG7	**Maintainability**	The system should be easily maintainable with standardized API generation and containerized deployments.
QG8	**Interaction Capability**	The user interface should be easily usable and be self-explanatory.
QG9	**Safety**	The likelihood of the solution posing a risk to humans or the environment is minimal. Therefore, the quality characteristic safety is not a primary concern.

1.3 Stakeholders

Role/Name	Expectations
(Project) Managers	Managers need project insights for good decision-making processes. They are interested in business-insights such as trends of key performance indicators, alerts, forecasts on a high abstract level as an overview. They will interact with the system via the frontend (dashboard view)
Data Scientists	Configure the analytics software and connect relevant data sources
Developers	Like to easily develop the software, prefer low complexity for individual components and may be interested in improving their development process
Software Architects	Are interested in how the solution is build, which design decision where made and in a good documentation

2. Architecture Constraints

2.1 Technical Constraints

Software Constraints

Label	Constraint	Motivation/Background
TSC1	All used third party software must be available under a permissive license and be accessible via a package manager	Developers should be able to easily install dependencies and execute the application. Only software with permissive licenses should be used to avoid costs and legal problems
TSC2	Implementation with Python, Apache Spark and Delta Lake	As the application has already been partly developed in Python using Spark and Delta Lake, it should be further used. Developers working on the project are also often students, which are often familiar with the programming language Python

Hardware Constraints

Label	Constraint	Motivation/Background
THC1	OS independent development	The system should be executable on all commonly used operating systems (Linux, Windows, macOS)
THC2	Deployable to Linux containers	The system should especially be able to be easily deployable to containers on Linux/Ubuntu basis
THC3	Locally executable	For easy development it should be able to run the system on one machine

2.2 Organizational Constraints

Label	Constraint	Motivation/Background
OC1	Implementable by a team of a small size	The application should be implemented and maintained by a team of small size
OC2	The source code should remain in a private repository	So far, there are no plans to make the source code freely available

2.3 Conventions

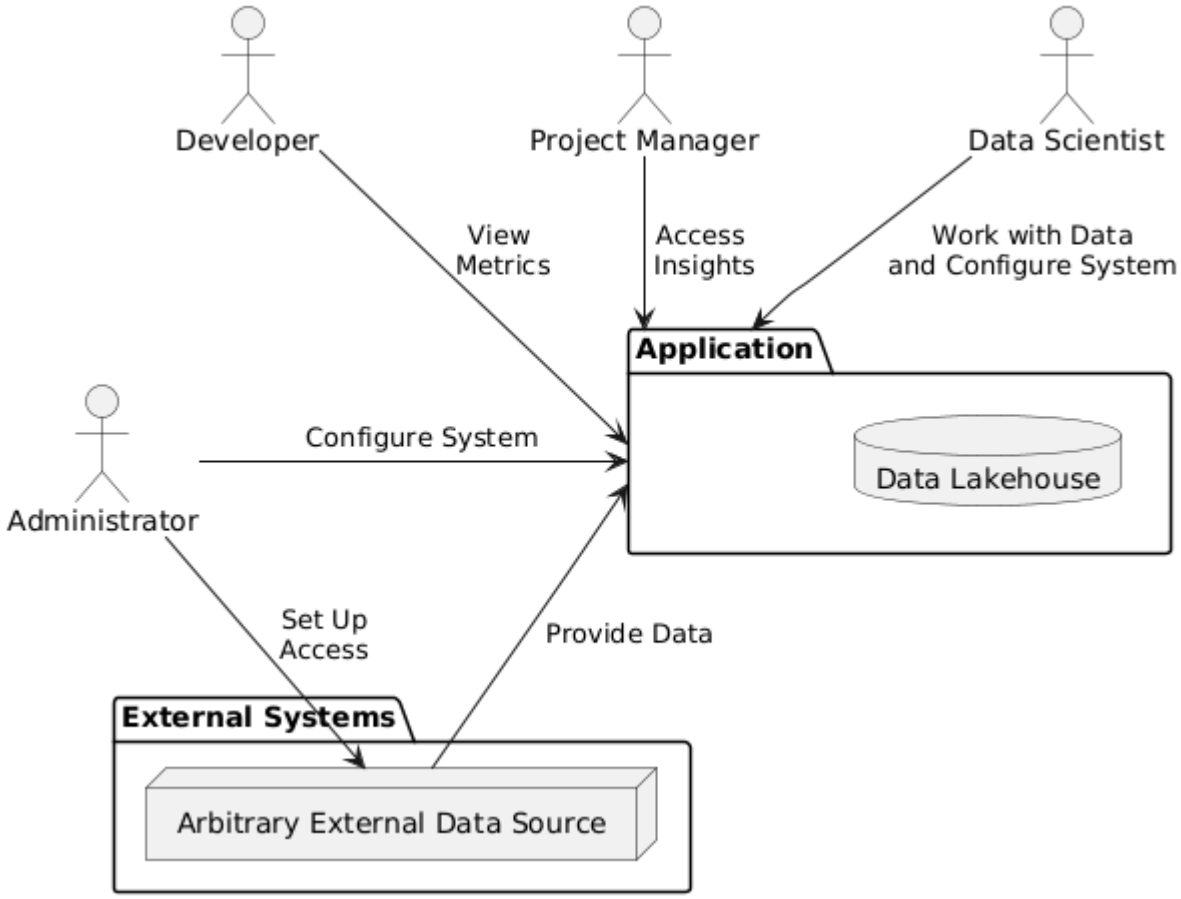
Label	Convention	Motivation/Background
C1	Coding conventions	Developers should align with the clean code principles. Before integration, code is reviewed via Pull Requests. The conventional commit framework for tagging commit messages is followed (see https://www.conventionalcommits.org/en/v1.0.0/).
C2	Language	As potentially international students work on the project and also the scientific community requires results in written english, english should be used in the entire project

Label	Convention	Motivation/Background
C3	Architecture Documentation	The architecture documentation is based on the english arc42 template (see [#Credits - Software architecture template (arc42)]

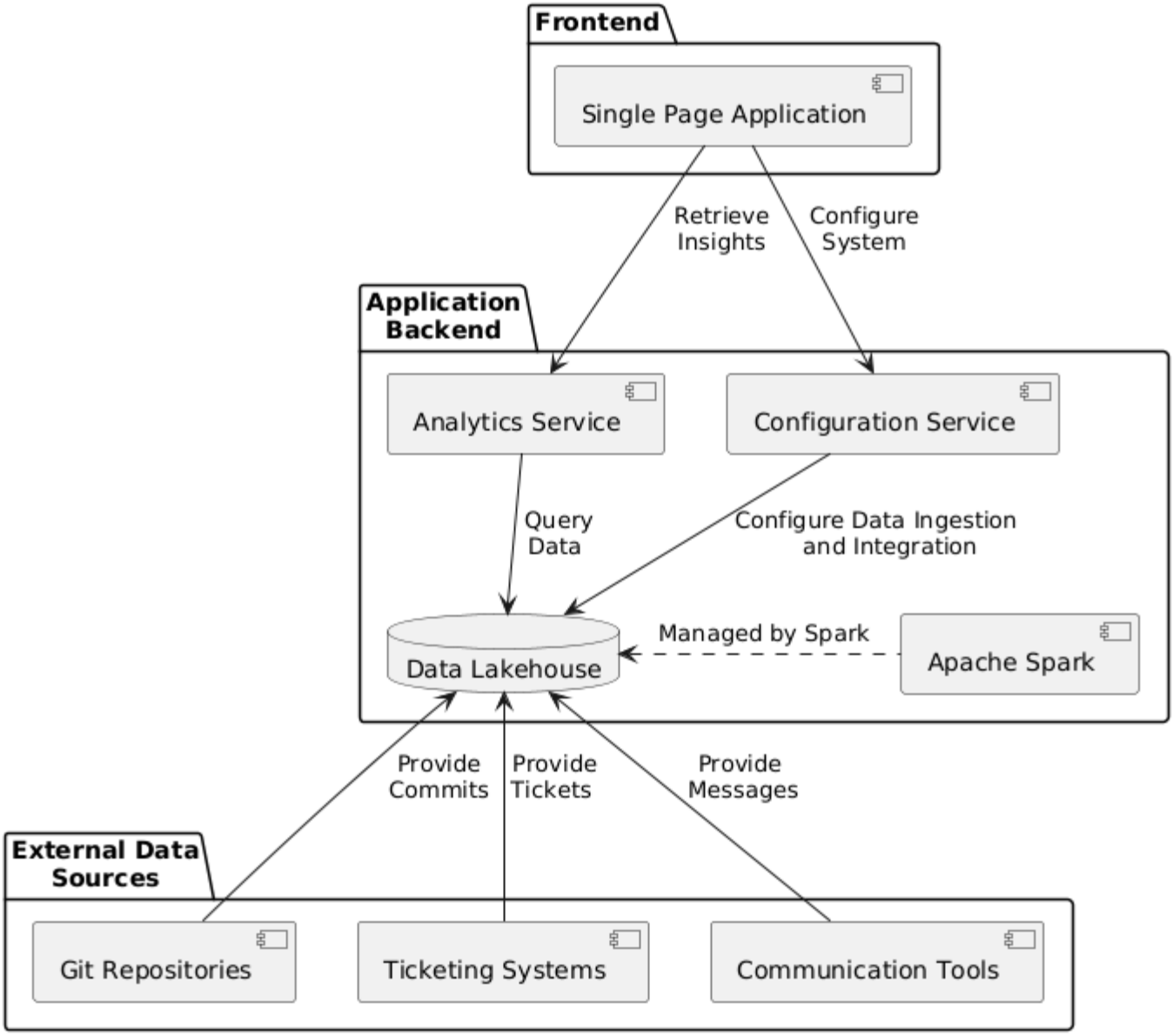
3. Context and Scope

3.1 Business Context

The application addresses the need for aggregating and analyzing software development data for various stakeholders, including developers, project managers and data scientists. The system enables the correlation of data from diverse sources such as Git repositories, ticketing systems and communication channels. Insights from software development data can support decision-making and improve software development practices by insights into the software development process. Also accounting issues in software development can be addressed like transfer pricing.



3.2 Technical Context



The application consists of four main components:

1. Data Lakehouse

The foundation of the application is a data lakehouse for software development data. It is managed by Apache Spark and supports the ingestion, normalization, and analysis of raw data.

2. Analytics Service

Based on the data of the data lakehouse the Analytics Service offers programmatic access to insights derived from the data lakehouse to the outer world.

3. Configuration Service

This service enables tenants to:

- Configure credentials such as API keys
- Manage data sources such as ingestion schedules and intervals.
- Define some integration rules for external data sources

4. Frontend

The frontend serves as the user interface, presenting analytical insights and metrics via the Analytics Service API. It also facilitates configuration management through the Configuration Service API.

Mapping Input/Output to Channels

Input	Channel	Output
Development Data Sources	API Endpoints, Message Queues, Ingestion Tasks	Ingested and Integrated Data in the Lakehouse
Analytics Queries	Frontend, Analytics Service API	Analytical Insights and Dashboards
Configuration Parameters	Frontend, Configuration API	Updated System Configuration

4. Solution Strategy

4.1 Overview

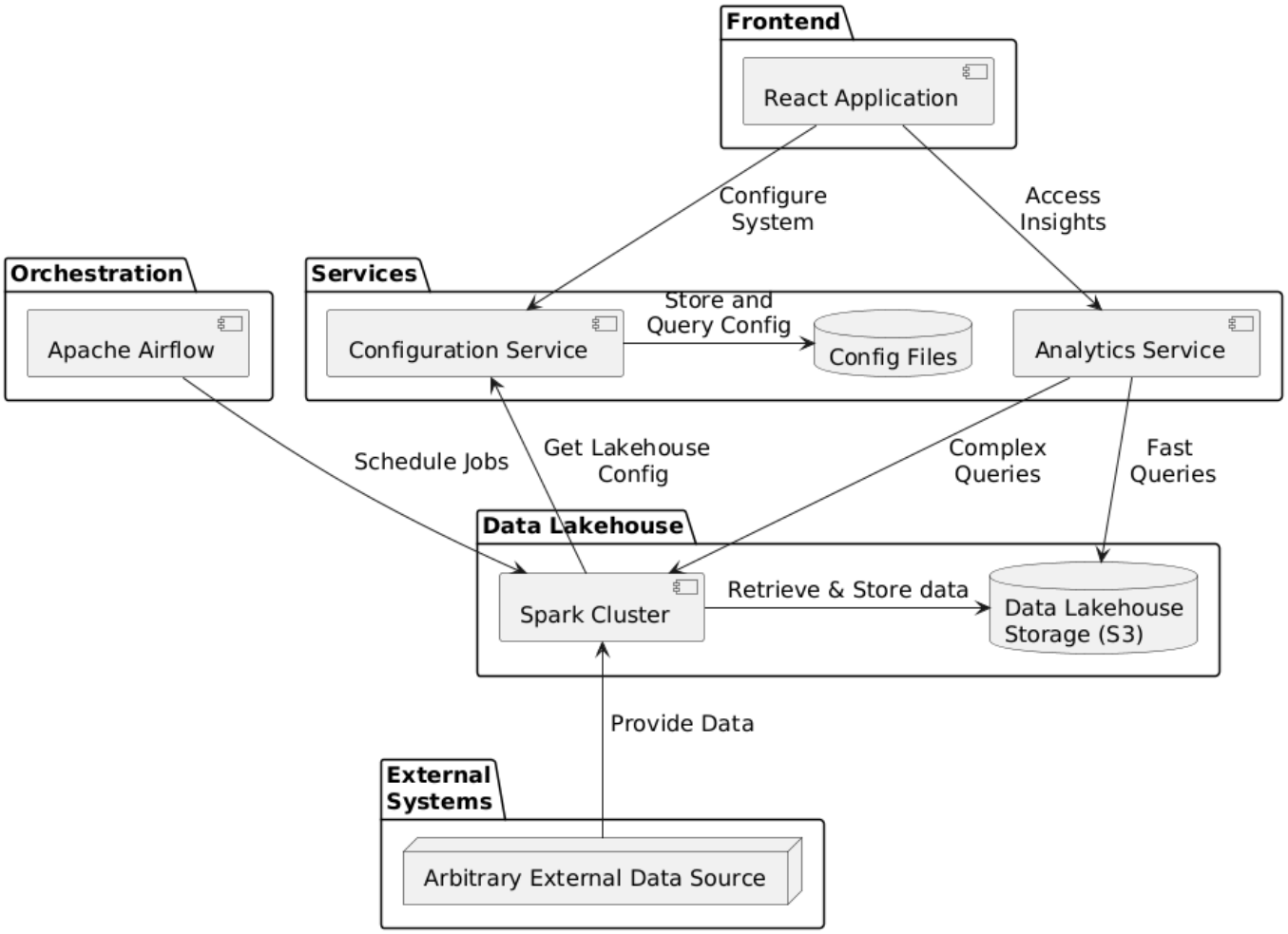
This section describes a high-level overview of the software architecture. The solution is divided into multiple layers:

- **Frontend layer:** The frontend consists of a React application visualizing analytical insights and configuration options
- **Service layer:** The service layer contains of two microservices that clearly separate the concerns calculating analytical insights and configuration management
- **Data management layer:** The data management layer / data lakehouse controls the ingestion and integration of data from arbitrary software development data sources
- **Orchestration:** Scheduling and orchestration of jobs in the data management layer is handled via Apache Kafka and Apache Airflow

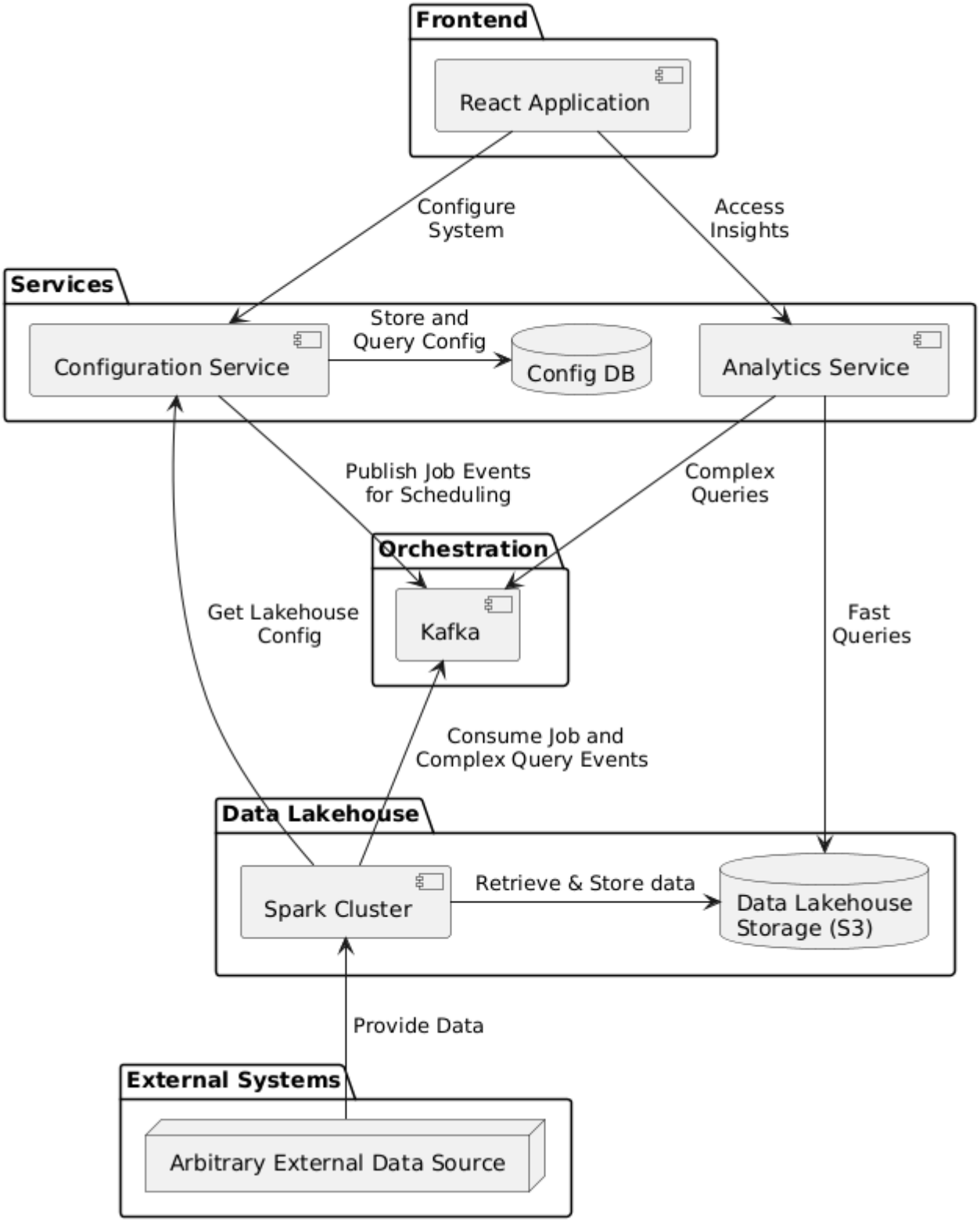
The architecture is designed for modularity, horizontal scalability, multi-tenancy and secure processing of sensitive data. The deployment of components is containerized for scalability and portability reasons.

The solution strategy differs for the single-tenant and multi-tenant solution. However, the multi-tenant solution introduces additional complexity and should only be persued if strictly required and OnPremises installations for multiple tenants are no longer an option.

4.1.1 Single-tenant Solution



4.1.2. Multi-tenant Solution



4.2 Key Design Decisions

The following design decisions are crucial for the solution.

Data Lakehouse

- Core of the solution is a data lakehouse that offers the ingestion and integration of big data volumes leveraging Apache Spark, Delta Lake and Amazon S3 based cheap file object stores.
- Data ingestion and integration are controlled via Spark.

Medallion Architecture

The data lakehouse is implemented as Medallion Architecture separating the data into different layers of integration.

1. Raw data is ingested into the Bronze layer
2. From the Bronze layer raw data is cleaned and transformed to the Silver layer
3. On the data of the Silver layer business-oriented aggregates are computed and stored to the Gold layer

Separation of Concerns

A microservices approach is used to clearly separate concerns:

- **Analytics Service:** Provides analytical insights by querying the Gold layer of the Medallion architecture directly through S3 for fast queries and the Gold layer using Spark for complex queries
- **Configuration Service:** Manages data source configurations, credentials and ingestion intervals. Therefore, the Configuration Service controls a separate database Config DB in the multi-tenant soluton only for datasource and tenant-specific configurations. In the single-tenant solution, it stores the configuration data as files in his container/at the host machine.
- Both microservices provide REST interfaces

Scheduling and Orchestration

Scheduling and Orchestration differs whether the solution is used for a single tenant only or for multiple tenants

- **Single-tenant:** Apache Airflow orchestrates ingestion and integration jobs
- **Multi-tenant:** Spark is also deployed as a consumer of Kafka job events to trigger execution. Job events are published by the configuration service as this service also controls ingestion intervals

The multi-tenant solution can also be used in a single-tenant mode. Nevertheless, the usage of Airflow seems to be simpler. Unfortunately Airflow does not support real privacy preserving multi-tenancy so far.

Security

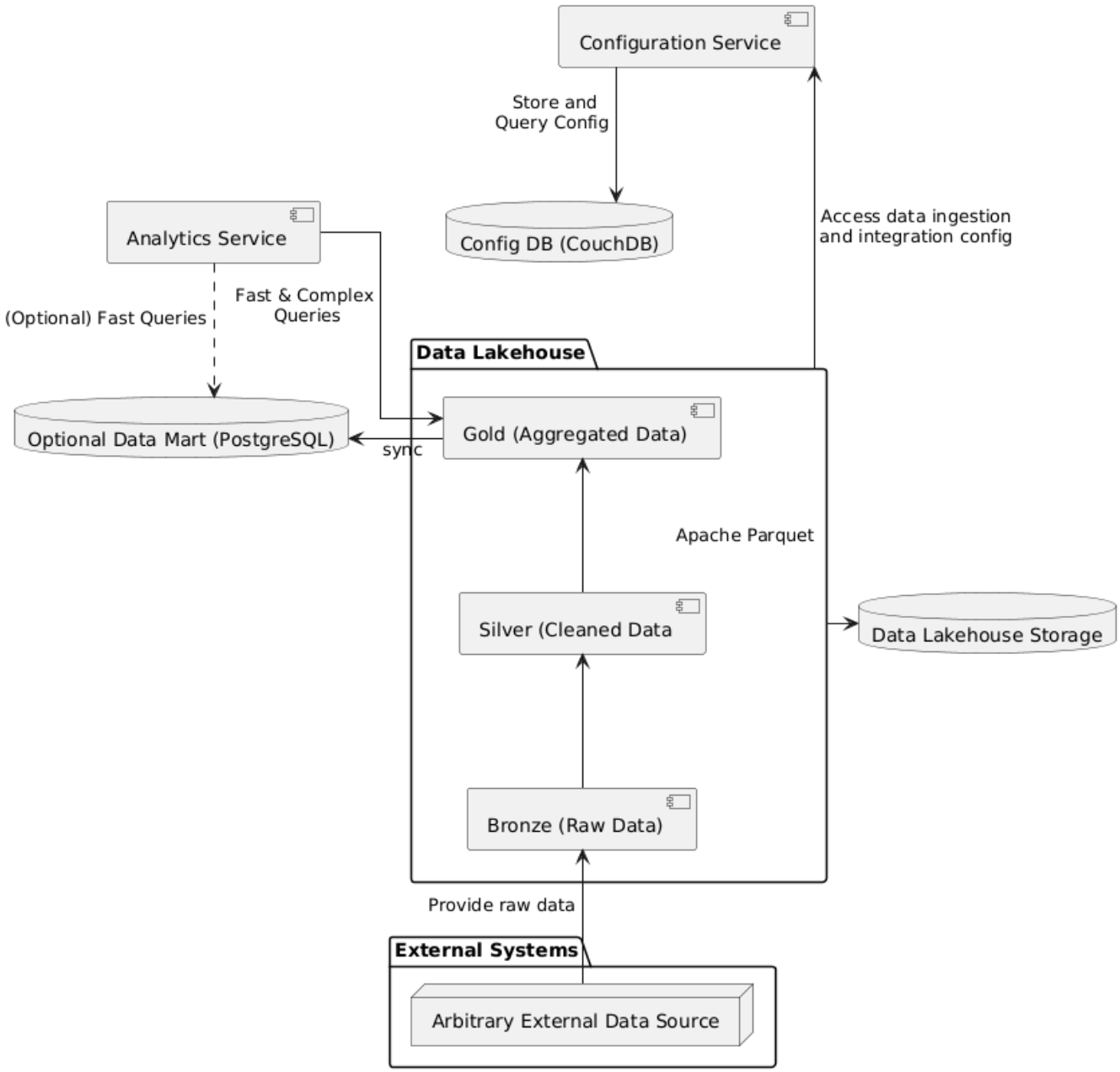
- API keys and sensitive data are hashed and stored securely in the Config DB using pepper and salt.
- Tenant data is isolated to ensure there is no unauthorized access
- Configuration and Analytics Service require authentication via JWT for access
- Access to all other systems is prohibited (except Airflow in single tenant mode)

Implementation according to standards

To follow the principle of least surprise, the solution follows industry standards:

- OpenApi
    - An API first approach is conducted
    - REST APIs are documented with OpenApi
    - API clients and Server stubs are generated using OpenAPI-Generator
- The software architecture documentation is based on an arc42 Template

4.3 Data Management



Medallion Architecture

The data lakehouse follows the Medallion Architecture:

- **Bronze Layer:** Stores raw data from external sources
- **Silver Layer:** Processes, cleans and normalizes data
- **Gold Layer:** Aggregates data for business insights

Key Components

- **Spark Cluster:** Manages data ingestion, transformation and integration.
- **S3 Object Storage:** Stores all lakehouse layers as Apache Parquet files. In multi-tenant mode files are partitioned by tenantId
- **PostgreSQL Gold Layer (Data Mart:** Holds pre-aggregated Gold Layer data for efficient and fast queries by the Analytics Service. Is designed to be optional and should only be introduced, if strict performance requirements can not be met without it.
- **Config DB:** Holds credentials and data ingestion and integration configuration

4.4 Mapping of Quality Goals to the Solution

Quality Goal	Design Decisions Supporting It
QG1 Functional Suitability	Data is ingested, integrated, and transformed in separate Spark jobs following the Medallion architecture. A Data Vault model provides traceable and meaningful structure. Incorrect ingestions can be avoided by job isolation and monitored ingestion logic.
QG2 Performance Efficiency	Metrics and other insights are pre-aggregated and persisted in the Gold layer. If necessary, they can be further synchronized to PostgreSQL to ensure fast user access. Spark ensures scalability and jobs can be scheduled for efficient batch processing.
QG3 Reliability	Reports are based on pre-aggregated and pre-validated data in the Gold layer. Consistency is ensured via ACID-compliant Delta Lake operations. Dashboards are allowed to be eventually consistent by design, e.g. via daily updates.
QG4 Compatibility	The Configuration Service allows tenants to configure new sources, including credentials and transformation rules. Schema evolution is supported via dynamic handling in Spark and by allowing changes to the configuration without redeploying jobs.
QG5 Flexibility	Tenants can define which sources and parts of sources they want to include. The system runs containerized and can be deployed in the cloud or on local infrastructure.
QG6 Security	Sensitive data like API keys is securely stored and hashed in the Configuration DB. Tenants are isolated logically within the data lakehouse by using tenant IDs and Delta Lake partitioning.
QG7 Maintainability	The system uses containerized deployments with standardized APIs defined via OpenAPI. Spark jobs are modular and can be re-used across different configurations.
QG8 Interaction Capability	A user interface allows end-users to explore insights. Interaction happens via a RESTful API. A star schema design in the Gold layer supports integration with BI tools such as PowerBI.
QG9 Safety	Not explicitly addressed as the application does not pose risks to people or the environment. Therefore, safety is not a primary concern.

4.5 Scheduling and Orchestration

- Data ingestion intervals can be configured by the user
- Following up on a data ingestion, data is integrated until the silver layer automatically
- Data is integrated to the Gold layer on a regularly (daily) basis
- Scheduling and Orchestration differs between single-tenant and multi-tenant mode.

Single-tenant

- Apache Airflow: Airflow is used to schedule and orchestrate jobs running on Spark. This includes scheduling data ingestion and orchestrating data integration jobs between Bronze, Silver and Gold layers.

Multi-tenant

- Queuing:
  - Spark consumes job events from a message queue. Jobs running on Spark are triggered via new job events.
  - For job orchestration purposes job events can be correlated and ordered via headers included in the event.
- The configuration service configures data ingestion intervals. Therefore, job events are published by the Configuration Service.

4.6 Security

Security is a critical consideration in the architecture as sensitive data is being processed.

Processed data

The following table describes which sensitive data is being processed in the application, how it is used, and the strategy to preserve its privacy.

Data Description	Usage	Privacy Preserving Strategy
Data Source Credentials	Authenticating to external systems for data ingestion (e.g. Git, Jira, etc.)	Stored in the Configuration DB. Hashed with pepper and salt. Encrypted when stored.
User Authentication Tokens	Used to authenticate clients (users) using JWT	Tokens are signed and verified and expire after a short time.
Tenant-Specific Configurations	Configuring data sources and ingestion schedules.	Stored securely in the Configuration DB. A tenant can only access its own configuration.
Ingested Data	Raw, processed, and aggregated data in the data lakehouse.	Restricted access and tenant isolation enforced at all layers. Data can only be retrieved from the outside via the Analytics service.

Key Privacy Measures

1. **Encryption**: Sensitive data such as API keys and credentials are encrypted.
2. **Role-Based Access Control**: Access to sensitive data is restricted based on user roles and permissions.
3. **Tenant Isolation**: Tenant-specific configurations and data are isolated to prevent unauthorized cross-tenant access.
4. **Token Expiry**: JWTs have short lifespans to minimize long-term misuse if a key gets leaked.

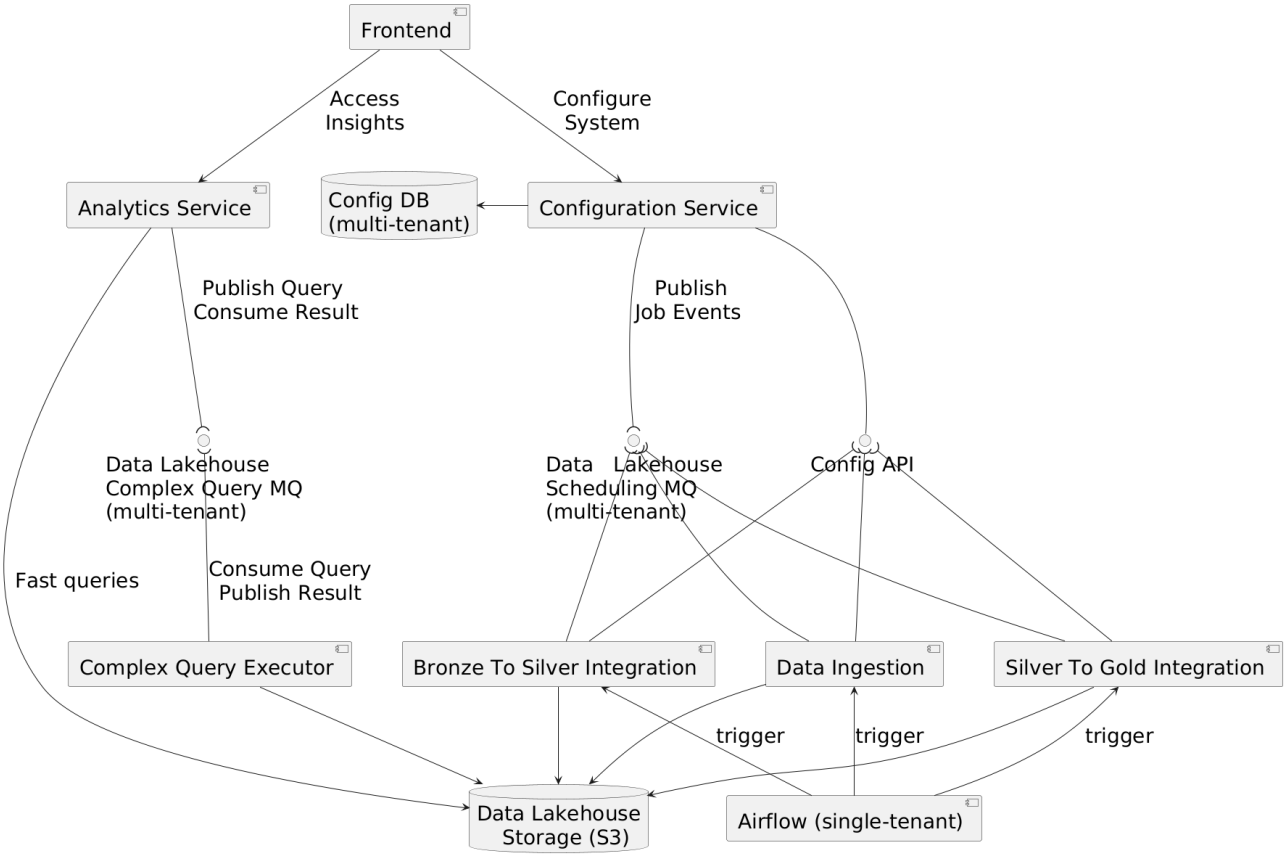
4.7 Technology Choices

The project is realized based on the following technologies:

Component	Technology
Frontend	React, TypeScript, OpenAPI generator
Analytics Service	TypeScript, Node.js, Express, OpenAPI generator
Configuration Service	TypeScript, Node.js, Express, OpenAPI generator, Couch DB (multi-tenant)
Data Lakehouse	Delta Lake, PySpark, Amazon S3 API, SeaweedFS, Apache Parquet, (PostgreSQL)
Scheduling & Orchestration	Apache Airflow, Apache Kafka
Deployment	Docker, Docker Compose

5. Building Block View

5.1 Whitebox Multi-Tenant Solution Overall System



Motivation

The system is designed with a layered, modularized architecture to achieve:

1. **Separation of concerns:** Each building block has its own responsibility with high cohesion
2. **Scalability:** The system has to be able to handle large data volumes (big data). For this reason horizontal scalability is a key goal.

3. **Maintainability:** Components have clearly defined interfaces and are independently deployable of each other
4. **Portability and simple Deployments:** Components are containerized
5. **Security:** Data isolation, access restrictions and secure processing of credentials

Contained Building Blocks

Building Block	Purpose/Responsibility
Frontend	The user interface for visualizing analytical insights and configuration management.
Analytics Service	Provides access to analytical insights, executes fast queries on pre-aggregated data and complex long-running queries.
Configuration Service	Manages data source configurations, credentials, and ingestion schedules.
Data Ingestion	Ingests raw data into the system from various external sources and stores it into the Bronze Layer.
Bronze To Silver Integration	Transforms raw data from the Bronze layer into cleaned, normalized data in the Silver Layer.
Silver To Gold Integration	Processes Silver data into the Gold Layer by performing business-oriented aggregations.
Gold Data Sync (optional)	Synchronizes pre-aggregated Gold Layer statistics to PostgreSQL for fast, frequent queries.
Complex Query Executor	Processes long-running analytical queries on the data lakehouse.

API Overview

The backend services expose OpenAPI compliant interfaces. Detailed interface descriptions are provided as OpenApi and can either be retrieved as raw openapi.yaml file or via a Swagger UI endpoint for each service.

These interfaces have not yet been implemented and can be seen as a proposal.

Analytics Service

- **GET** /tenants/{tenantId}/metrics
  - Get all metrics for a specific tenant.
- **GET** /tenants/{tenantId}/metrics/{metricId}
  - Retrieve a specific metric for a tenant.
- **POST** /tenants/{tenantId}/queries
  - Execute a SPARQL query on the data lakehouse.
- **GET** /tenants/{tenantId}/queries/{queryId}
  - Retrieve the status or result of a previously executed query.

Configuration Service

- **Configuration Management**
  - **GET** /tenants/{tenantId}/configurations
    - Retrieve all configurations for a tenant.
  - **POST** /tenants/{tenantId}/configurations
    - Create a new configuration for a tenant.
  - **DELETE** /tenants/{tenantId}/configurations/{configurationId}
    - Delete a specific configuration for a tenant.
  - **PUT** /tenants/{tenantId}/configurations/{configurationId}
    - Update an existing configuration for a tenant.
- **Job Management**
  - **POST** /tenants/{tenantId}/jobs
    - Create a new job for a tenant.
  - **GET** /tenants/{tenantId}/jobs/{jobId}
    - Retrieve the status or details of a specific job for a tenant.
- **Tenant Management**
  - **GET** /tenants/{tenantId}
    - Retrieve information about a tenant's user account.
  - **POST** /tenants
    - Create a new tenant's user account.
  - **DELETE** /tenants/{tenantId}
    - Delete a tenant's user account.
  - **PUT** /tenants/{tenantId}
    - Update information for a tenant's user account.

Black Box: Frontend

Purpose/Responsibility:
The Frontend is a React and TypeScript-based single page application that:

- Visualizes analytical insights of software development data
- Allows the user to manage data source configurations and ingestion schedules

Interface(s):

- UI available from a browser, does not provide additional interfaces

Quality/Performance Characteristics:

- Efficient visualization of business insights (QG1).
- Support of user interaction for data source configurations (QG4).
- Integrates interfaces with OpenAPI-generated client code for maintainability reasons (QG8).
- Responsive UI that clearly separates concerns (analytical insights vs. configuration) (QG9).

Fulfilled Requirements:

Black Box: Analytics Service

Purpose/Responsibility:

The Analytics Service provides analytical insights by:

- Performing fast queries against PostgreSQL on pre-aggregated Gold Layer data.
- Executing complex long-running queries on data in the data lakehouse using Spark.
- Providing the results via a OpenAPI generated REST-interface

Interface(s):

- **GET** /tenants/{tenantId}/metrics
- **GET** /tenants/{tenantId}/metrics/{metricId}
- **POST** /tenants/{tenantId}/queries

Quality/Performance Characteristics:

- Low-latency querying for pre-aggregated data such as metrics (QG1).
- Eventually consistent metrics and support for consistent resource-intense queries (QG2).
- Handles complex resource-intensive queries on the data lakehouse (QG5).
- Only offers access to data of a tenant. Access to data of other tenants is prohibited (QG7).
- Standardized API generation (QG8).

Fulfilled Requirements:

Black Box: Configuration Service

Purpose/Responsibility:

The Configuration Service manages:

- Tenant-specific data source configurations and credentials.
- Scheduling and management of ingestion jobs (multi-tenant mode without Airflow)

Interface(s):

- **Configuration Management**:
  - **GET** /tenants/{tenantId}/configurations
  - **POST** /tenants/{tenantId}/configurations
  - **DELETE** /tenants/{tenantId}/configurations/{configurationId}
  - **PUT** /tenants/{tenantId}/configurations/{configurationId}
- **Job Management**:
  - **POST** /tenants/{tenantId}/jobs
  - **GET** /tenants/{tenantId}/jobs/{jobId}

Quality/Performance Characteristics:

- New data sources can be added easily (QG3).
- Configuration which data source should be used and how (QG4).
- Securely store credentials using pepper and salt (QG7).
- Standardized API generation (QG8).

Fulfilled Requirements:

Black Box: Data Ingestion

Purpose/Responsibility:

The Data Ingestion is responsible for ingesting raw data into the system from various external sources.

- Data is retrieved via API calls, database access etc.
- Ingested data is stored in the Bronze layer via data storage.

Interface(s):

- **Ingestion Configuration**: Retrieves ingestion configuration from the Configuration Service.
- **Ingestion Execution**: Ingestion jobs are triggered based on job events from the Scheduling & Orchestration component.

Quality/Performance Characteristics:

- Handles various external data sources (QG3).
- Ingestion works for various data sources based on a tenant´s configuration (QG4)
- Ingestion is stateless and can be scaled horizontally (QG5).

Black Box: Data Integration

Purpose/Responsibility:
Data Integration handles the transformation of raw data, to cleaned data and business aggregates according to the Medallion architecture.

Interface(s):

- **Integration Configuration**: Retrieves transformation rules and configurations from the Configuration Service.
- **Data Processing**:
    - Process to silver: Data cleansing and normalization
    - Process to gold: Business-oriented aggregation of data

Quality/Performance Characteristics:

- Scalable data processing using PySpark (QG5).
- Ensures data isolation per tenant (QG7).

Black Box: Data Storage

Purpose/Responsibility:
Data Storage manages the storage of data in the data lakehouse.

- Stores data in the data lakehouse in Apache Parquet format in an S3 compatible object store
- Syncs pre-aggregated data of the Gold layer in the data lakehouse to a Gold PostgreSQL for efficient, low-latency access

Interface(s):

- **Data Storage Management**: Provides APIs for storing and retrieving data at each layer.
- **Analytics Query Support**: Serves processed data to the Analytics Service for querying.

Quality/Performance Characteristics:

- Low-latency querying from the Gold Layer using PostgreSQL (QG1).
- Stores tenant-isolated data to ensure privacy and security (QG7).
- Implements partitioning and compression to handle large-scale data efficiently (QG5).

Black Box: Scheduling & Orchestration

Purpose/Responsibility:
Coordinates job scheduling and orchestration of jobs running in the data lakehouse using:

- **Apache Airflow**: Handles periodic ingestion jobs (single-tenant mode)
- **Kafka**: Kafka is used as a middleware in multi-tenant mode. Job events are published by the configuration service. Job events are consumed from Kafka by Pyspark to trigger the execution of jobs.

Interface(s):

- **Airflow**: Schedules single-tenant ingestion jobs.
- **Kafka**:
  - Publish job events: **POST** /tenants/{tenantId}/jobs
  - Consume job events: Internal consumers triggered by Kafka topics.

Quality/Performance Characteristics:

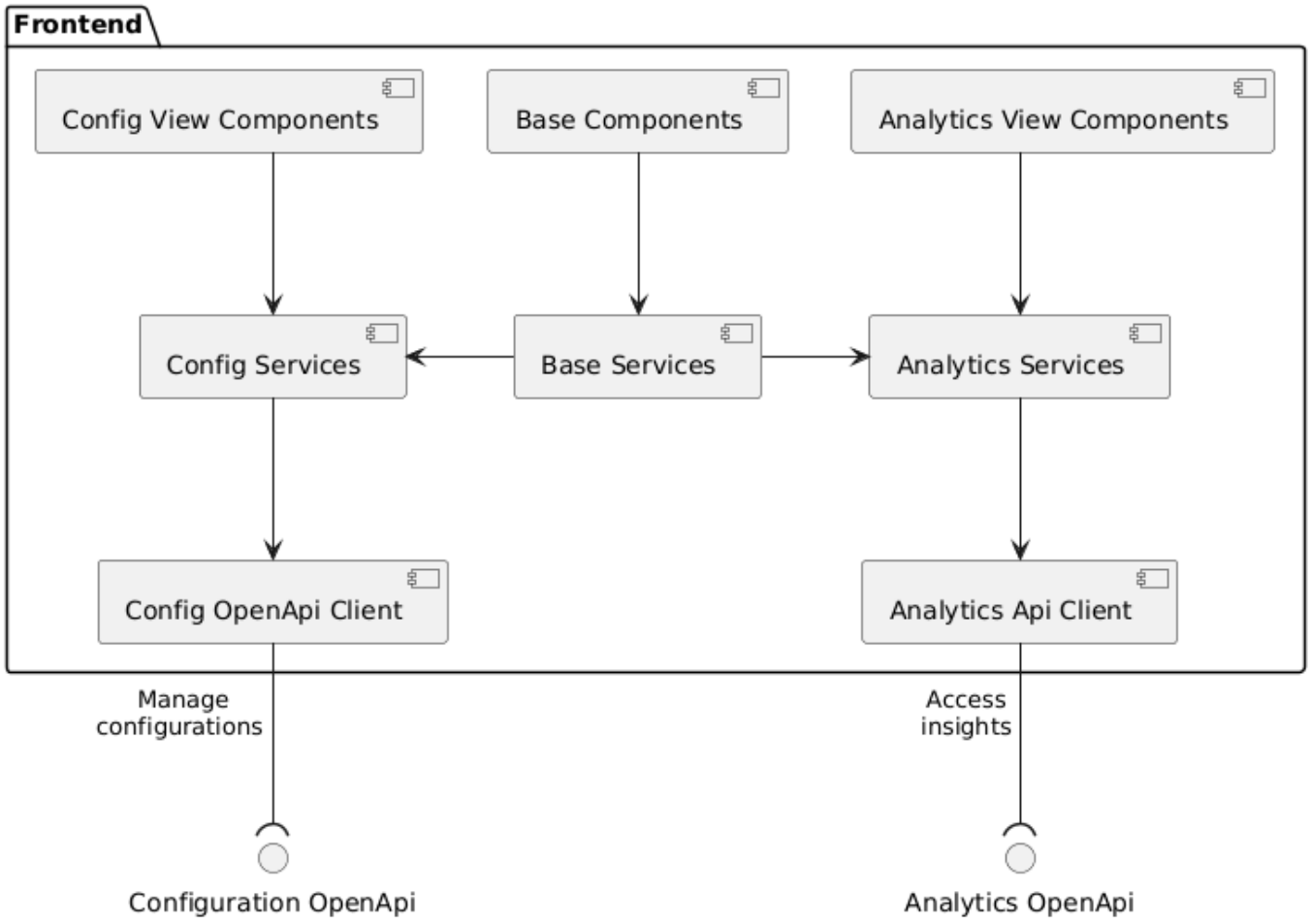
- Scalable, event-driven job orchestration (QG5).

Fulfilled Requirements:

5.2 Whitebox View of Building Blocks

In the following, each building block is described in detail with a focus on major internal components, interfaces used or provided and data flow.

Whitebox: Frontend



Purpose/Responsibility

- The **Frontend** is implemented as a single-page application in React and Typescript.
- Users can access analytical insights and manage data ingestion and integration configuration.

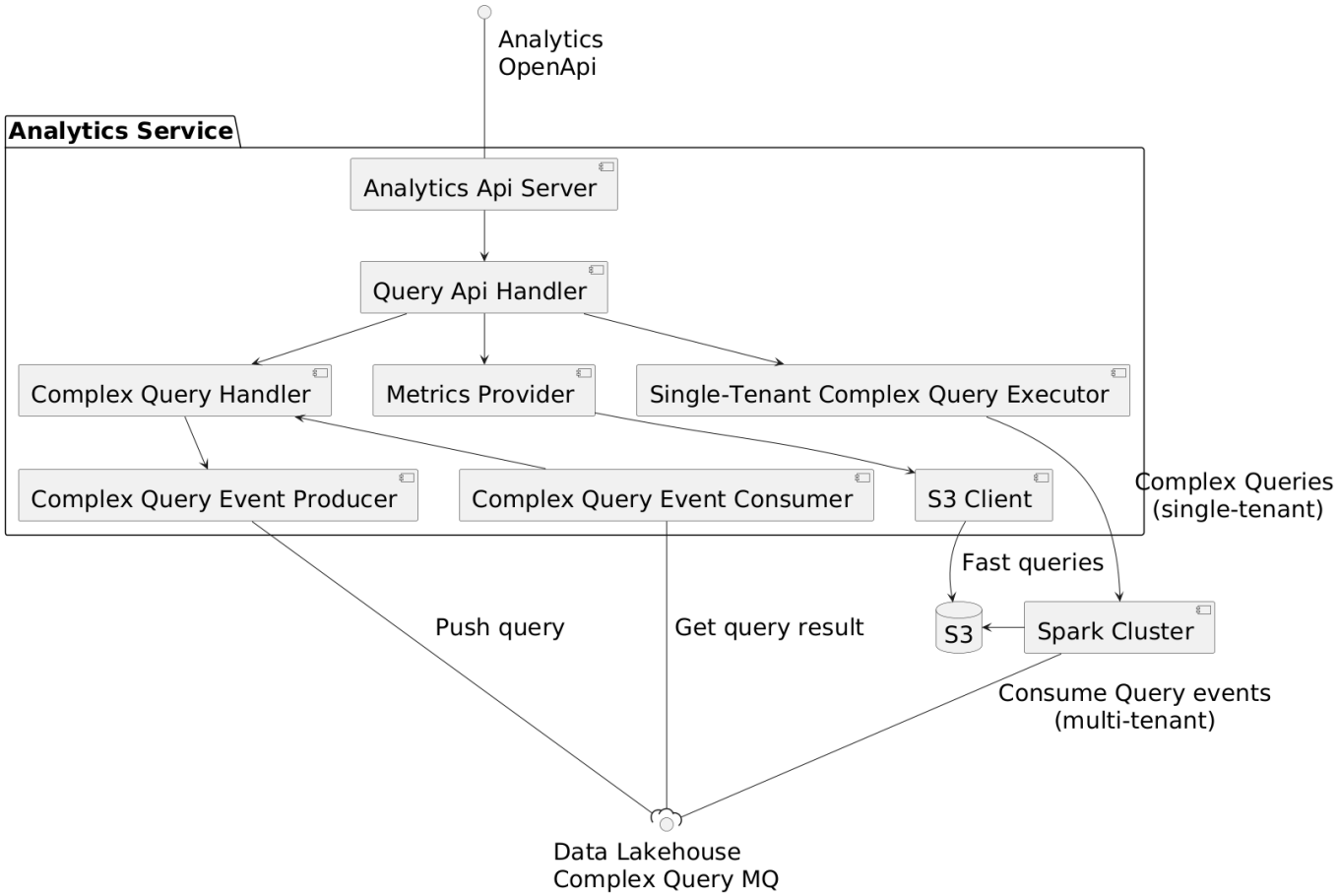
Internal Structure

Component	Responsibility
Base Components	Basic UI elements (footer, navbar, search, etc.).
Analytics View Components	UI elements for visualizing analytical insights.
Config View Components	UI elements for managing ingestion and integration configurations.
Base Services	Encapsulates services commonly used in different components and shared frontend logic.
Analytics API Client	Communicates with the **Analytics API**. Client code is partly generated using OpenApi Generator.
Config API Client	Communicates with the **Configuration API**. Client code is partly generated using OpenApi Generator.

Interfaces

Interface	Direction	Purpose
Analytics OpenApi	**Used**	Retrieve insights and metrics.
Configuration OpenApi	**Used**	Manage system configurations.

Whitebox: Analytics Service



Purpose/Responsibility

- Provides analytical insights by querying pre-aggregated data from **PostgreSQL (Gold Layer)** or executing **complex queries** on the **Data Lakehouse** by using a message queue.

Internal Structure

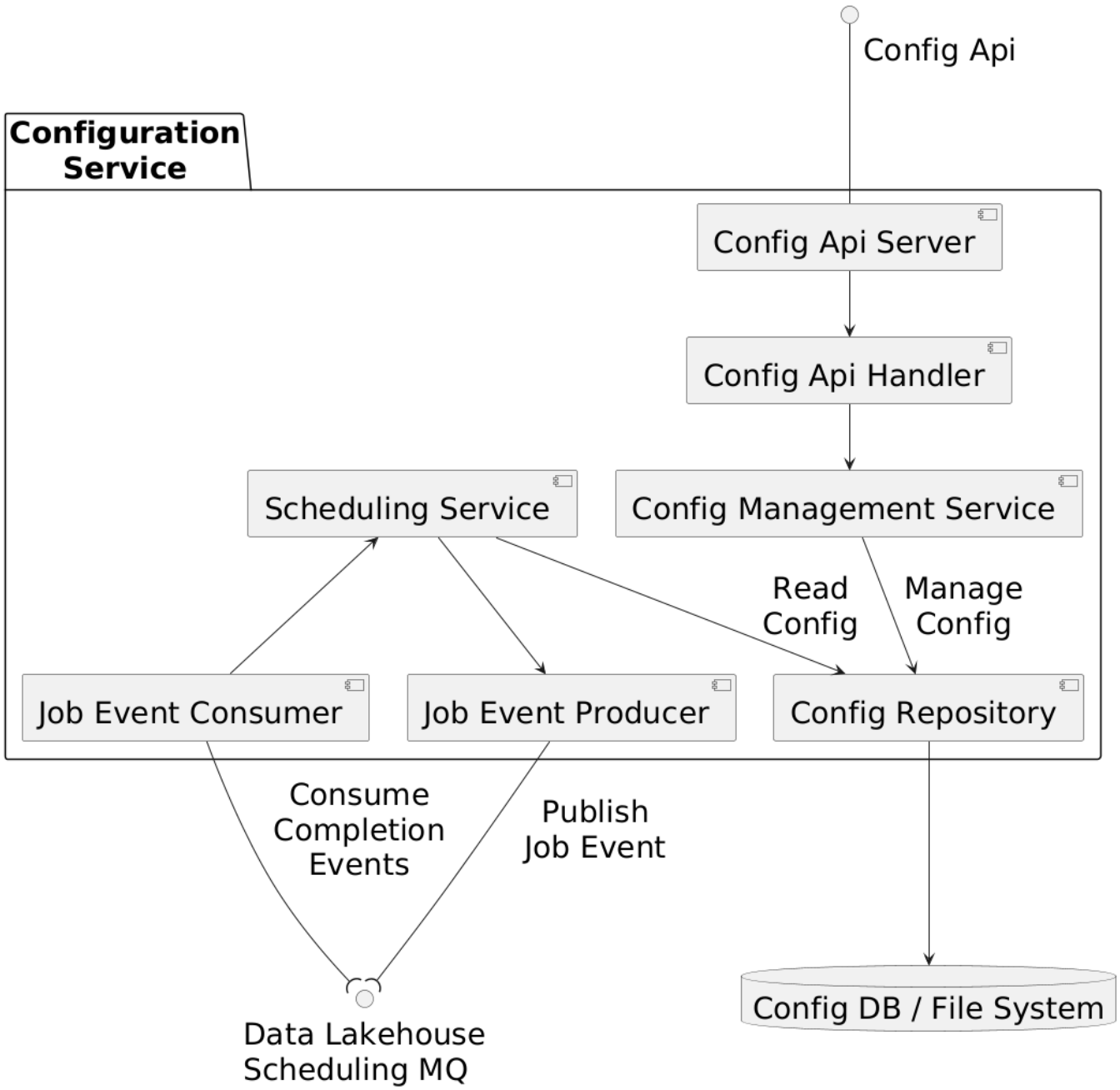
Component	Responsibility
Analytics Api Server	The boundary of the Analytics Service providing a REST interface offering analytical insights to the outer world. Server code is partly generated using OpenApi Generator.
Query API Handler	Handles query requests from the frontend and decides whether to perform a fast query or a complex-long running query asynchronously.
Metrics Provider	Executes business logic necessary to provide pre-aggregated metrics from the PostgreSQL Gold Layer.

Component	Responsibility
PostgreSQL Client	Queries PostgreSQL for fast, efficient data retrieval.
Complex Query Executor	Executes business logic necessary to handle complex queries.
Complex Query Event Producer	Publishes query requests to the MQ.
Complex Query Event Consumer	Retrieves query results from the MQ.

Interfaces

Interface	Direction	Purpose
Data Lakehouse Complex Query MQ	**Used**	MQ for executing complex queries on the data lakehouse asynchronously. Queries can be pushed to the MQ. After the query has been executed, the result will be published to the MQ and can be retrieved by consuming the result event. This building block publishes query events and consumes their result.
Analytics OpenApi	**Provided**	Provides analytical insights to software development data in the data lakehouse as REST interface.
PostgreSQL (Gold Layer)	**Used**	Holds pre-aggregated analytical metrics for fast, frequent reads.

Whitebox: Configuration Service



Purpose/Responsibility

- Manages tenant-specific configurations, credentials and scheduling of jobs in the data lakehouse.

Internal Structure

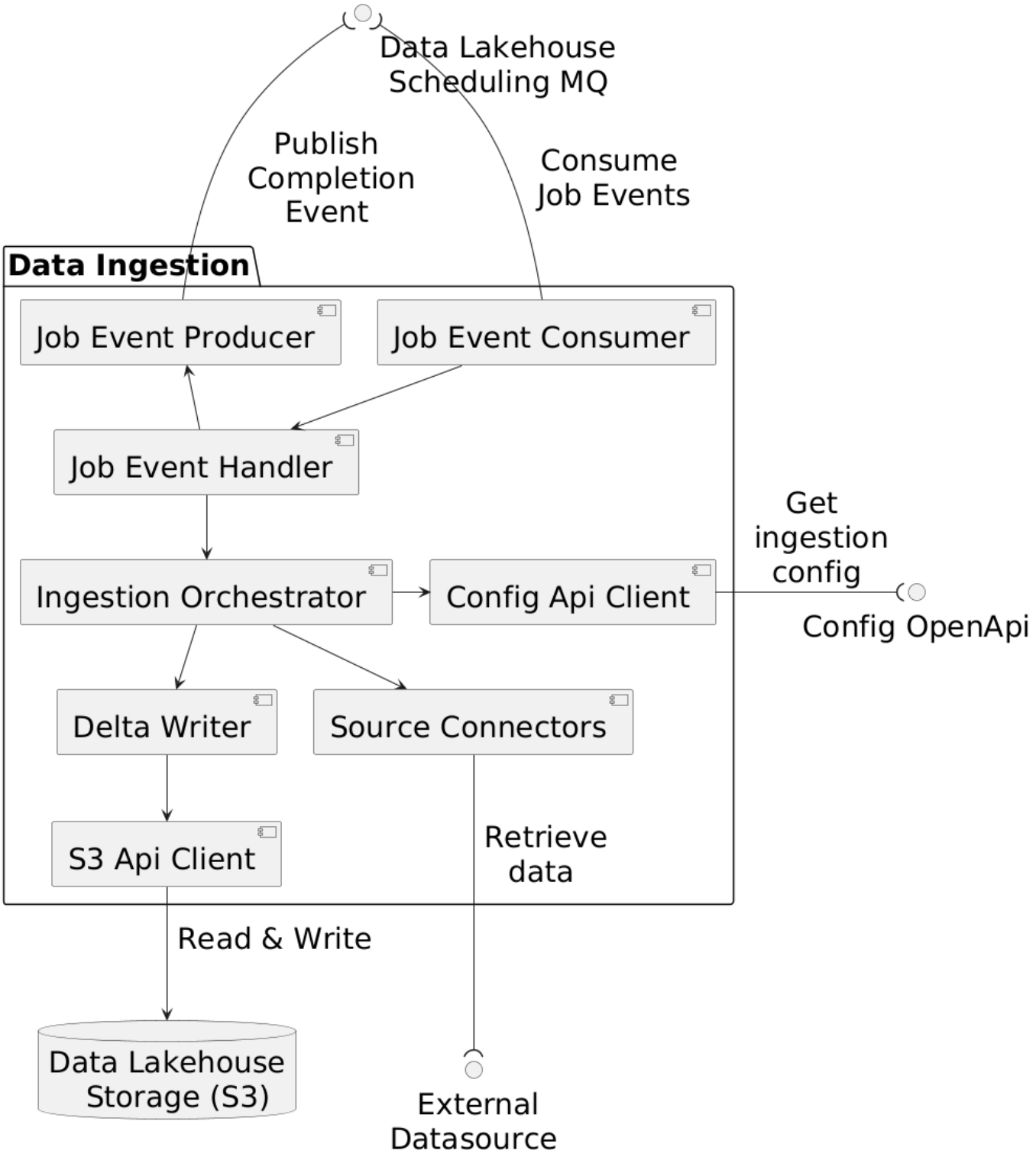
Component	Responsibility
**Config Api Server	The boundary of the Configuration Service providing a REST interface offering configuration management to the outer world. Server code is partly generated using OpenApi Generator.
Config Api Handler	Manages API requests for configurations and therefore calls various config management interfaces from the Config Management Service.

Component	Responsibility
Config Management Service	Acts as service layer that involves business logic for managing configuration.
Config Repository	Layer responsible for communicating with the database and providing access to the data in the Config DB.
Scheduling Service	Reads the config and continuously publishes job events to the MQ based on the configuration.

Interfaces

Interface	Direction	Purpose
Data Lakehouse Scheduling MQ	**Used**	MQ used for scheduling jobs in the data lakehouse. Workers subscribe to job events and perform jobs for which they are responsible in the data lakehouse. This building block publishes job events such as ingestion and transformation jobs.
Configuration OpenApi	**Provided**	Provides configuration management to manage data ingestion and integration in the data lakehouse via a REST interface.
Configuration DB	**Used**	Stores tenant-specific ingestion and integration configurations, credentials, and scheduling information.

Whitebox: Data Ingestion



Purpose/Responsibility

- Extracts raw data from various external sources and writes it to the Bronze Layer in the data lakehouse.

Internal Structure

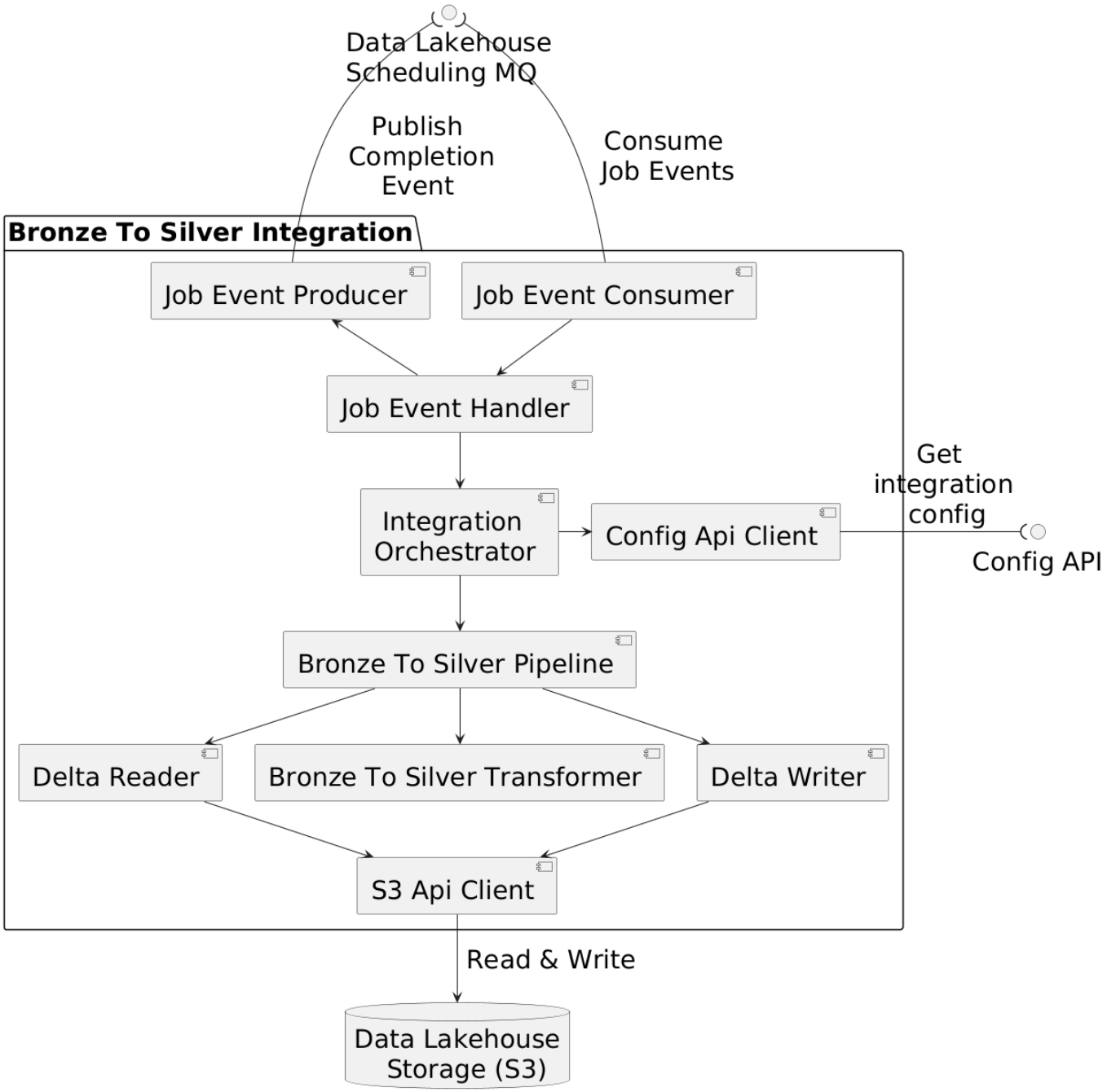
Component	Responsibility
Job Event Consumer	Listens for ingestion job events.

Component	Responsibility
Ingestion Orchestrator	Manages ingestion workflows by calling the required Source Connectors based on the job event with the required configuration.
Config Api Client	Retrieve configurations from the Config Api. Client code is partly generated using OpenApi Generator.
Source Connectors	Fetches data from APIs, databases, etc. Each datasource has its own connector (multiple possibilities such as GitHubConnector, JiraConnector etc.)
Delta Writer	Stores ingested data in the Bronze Layer using Delta Lake.
S3 Api Client	Provides read and write access to files stored in S3.

Interfaces

Interface	Direction	Purpose
Data Lakehouse Scheduling MQ	**Used**	MQ used for scheduling jobs in the data lakehouse. Workers subscribe to job events and perform jobs for which they are responsible in the data lakehouse. This building block consumes ingestion job events and executes the required ingestion job.
Config API	**Used**	Retrieves ingestion configurations.
Data Lakehouse Storage (S3)	**Used**	S3-compatible object store containing the data lakehouse data. Data is stored in files in Apache Parquet format in S3.

Whitebox: Bronze to Silver Integration



Purpose/Responsibility

- Cleans and normalizes raw Bronze Layer data and processes it to the Silver Layer.

Internal Structure

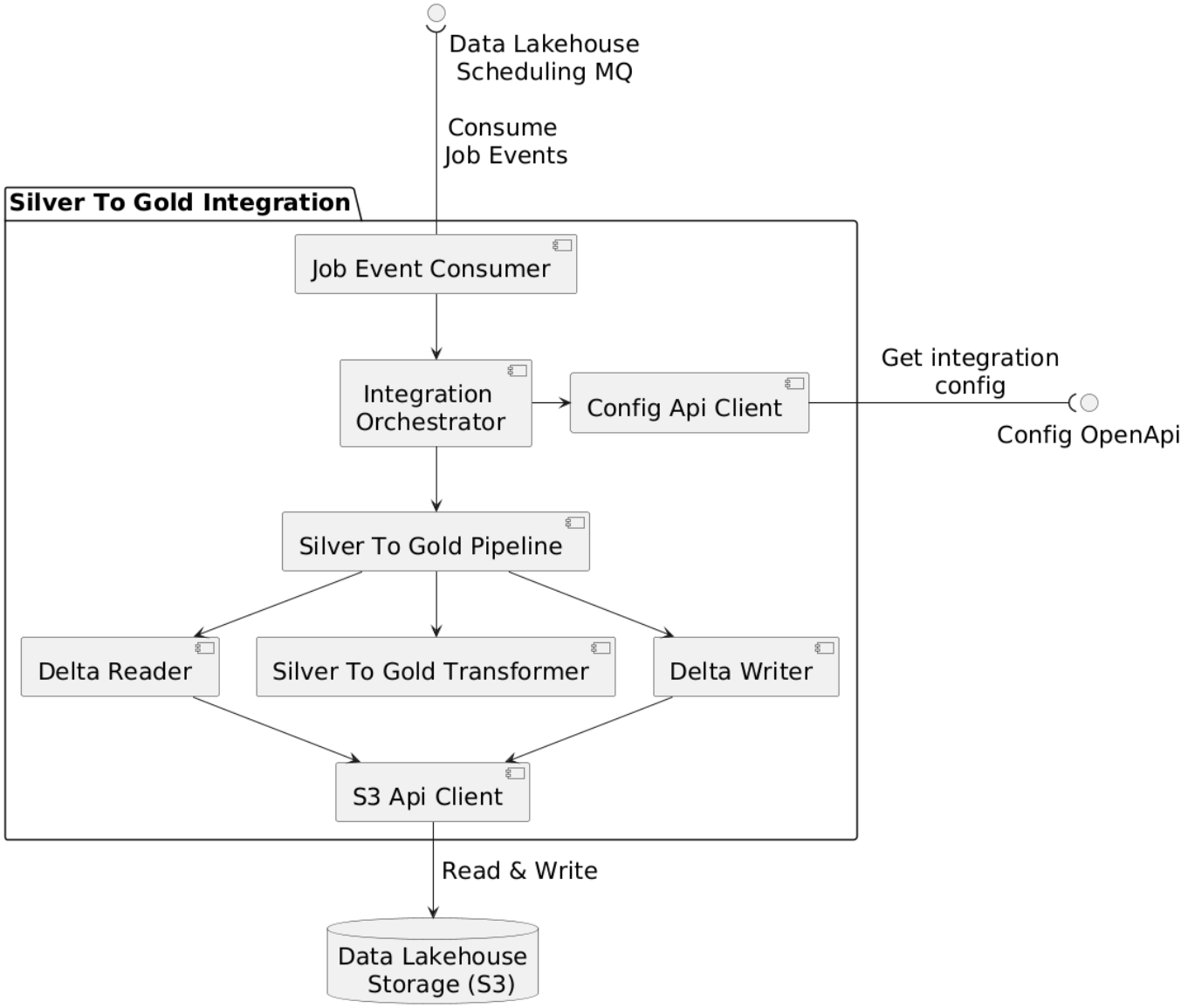
Component	Responsibility
Job Event Consumer	Listens for Bronze To Silver Integration Jobs Events.
Integration Orchestrator	Manages the integration by calling the Bronze To Silver Pipeline with the required configuration.
Bronze To Silver Pipeline	Calls data retrieval, required transformation services and triggers the storage of the transformed data to the Silver Layer.

Component	Responsibility
Bronze To Silver Transformer	Applies cleaning, normalization, and validation rules to raw data.
Delta Reader	Reads required data from the data lakehouse by calling the correct Delta Lake tables.
Delta Writer	Writes transformed data back to the Silver Layer in the Data Lakehouse.
S3 Api Client	Provides read and write access to files stored in S3.
Config Api Client	Retrieve configurations from the Config Api. Client code is partly generated using OpenApi Generator.

Interfaces

Interface	Direction	Purpose
Lakehouse Scheduling MQ	**Used**	MQ used for scheduling jobs in the data lakehouse. Workers subscribe to job events and perform jobs for which they are responsible in the data lakehouse. This building block consumes bronze to silver integration job events and executes the required integration job.
Config API	**Used**	Retrieves transformation configurations.
Data Lakehouse Storage (S3)	**Used**	S3-compatible object store containing the data lakehouse data. Data is stored in files in Apache Parquet format in S3.

Whitebox: Silver to Gold Integration



Purpose/Responsibility

- Aggregates data from the Silver Layer into the Gold Layer to gain business-oriented insights.

Internal Structure

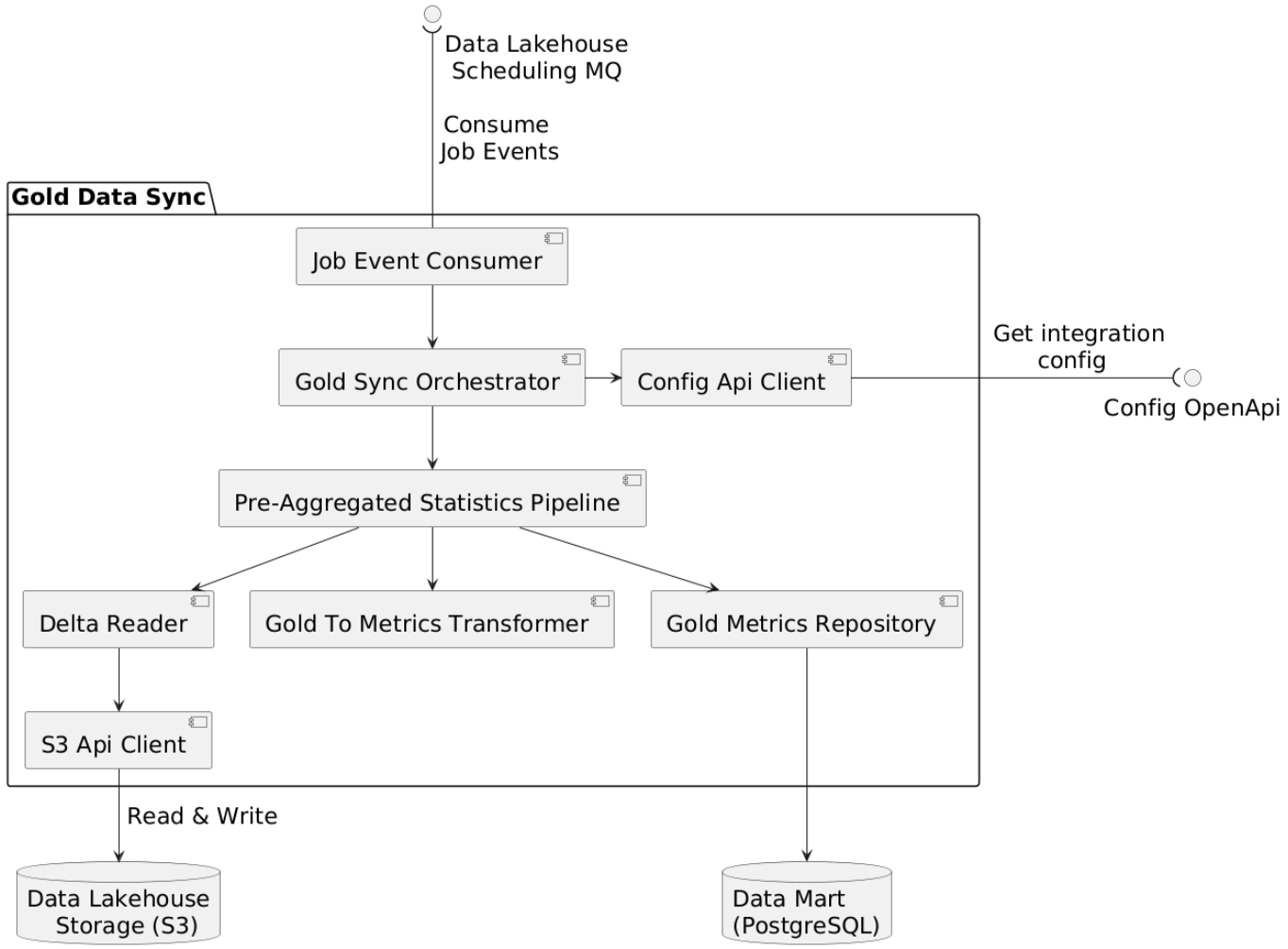
Component	Responsibility
Job Event Consumer	Listens for Silver To Gold Integration Jobs Events.
Integration Orchestrator	Manages the integration by calling the Silver To Gold Pipeline with the required configuration.
Silver To Gold Pipeline	Calls data retrieval, required transformation services and triggers the storage of the transformed data to the Gold Layer.
Silver To Gold Transformer	Manages the transformation of silver data to gold data by performing aggregations.

Component	Responsibility
Delta Reader	Reads required data from the data lakehouse by calling the correct Delta Lake tables.
Delta Writer	Writes transformed data back to the Gold Layer in the Data Lakehouse.
S3 Api Client	Provides read and write access to files stored in S3.
Config Api Client	Retrieve configurations from the Config Api. Client code is partly generated using OpenApi Generator.

Interfaces

Interface	Direction	Purpose
Lakehouse Scheduling MQ	**Used**	MQ used for scheduling jobs in the data lakehouse. Workers subscribe to job events and perform jobs for which they are responsible in the data lakehouse. This building block consumes silver to gold integration job events and executes the required integration job.
Config API	**Used**	Retrieves transformation configurations.
Data Lakehouse Storage (S3)	**Used**	S3-compatible object store containing the data lakehouse data. Data is stored in files in Apache Parquet format in S3.

Whitebox: Gold Data Sync



Purpose/Responsibility

- Synchronizes **pre-aggregated statistics** from the **Gold Layer** into PostgreSQL for **fast analytics queries**.

Internal Structure

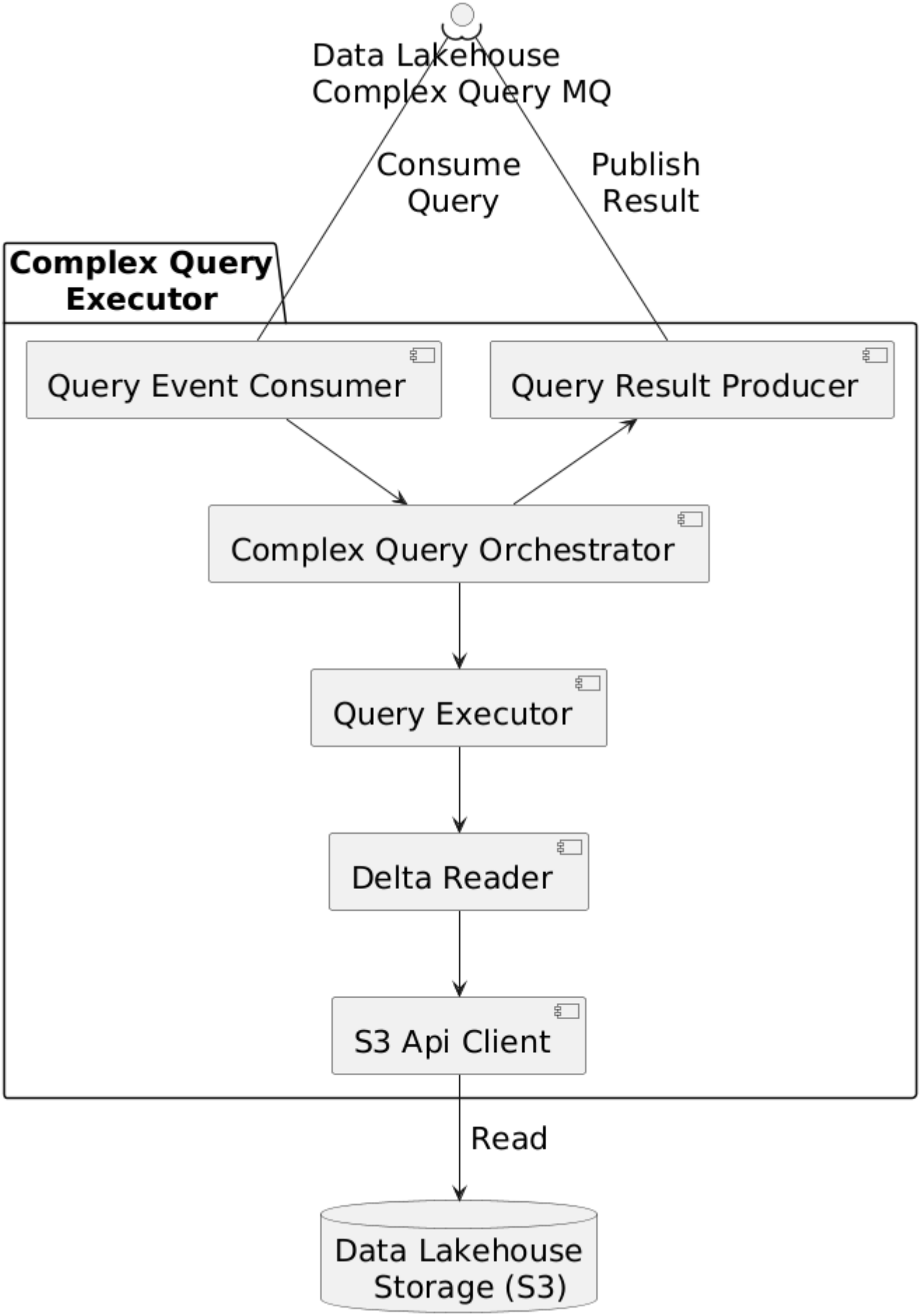
Component	Responsibility
Job Event Consumer	Listens for synchronization job events.
Gold Sync Orchestrator	Manages the syncing pipeline by calling required business logic with the required configuration.
Pre-Aggregated Statistics Pipeline	Calls data retrieval, required transformation services and triggers the storage of the transformed data to PostgreSQL.
Gold Metrics Repository	Layer responsible for communicating with the database and providing access to the data in the PostgreSQL (Gold Layer) DB.
Config Api Client	Retrieve configurations from the Config Api. Client code is partly generated using OpenApi Generator.
Delta Reader	Reads required data from the data lakehouse by calling the correct Delta Lake tables.

Component	Responsibility
S3 Api Client	Provides read and write access to files stored in S3.

Interfaces

Interface	Direction	Purpose
Data Lakehouse Scheduling MQ	**Used**	MQ used for scheduling jobs in the data lakehouse. Workers subscribe to job events and perform jobs for which they are responsible in the data lakehouse. This building block consumes gold data sync job events and syncs the gold data in the data lakehouse with PostgreSQL (Gold Layer).
Config API	**Used**	Retrieves ingestion configurations.
Data Lakehouse Storage (S3)	**Used**	S3-compatible object store containing the data lakehouse data. Data is stored in files in Apache Parquet format in S3.
PostgreSQL (Gold Layer)	**Used**	Holds pre-aggregated analytical metrics for fast, frequent reads.

Whitebox: Complex Query Executor



Purpose/Responsibility

- The Complex Query Executor processes long-running advanced analytical queries on the data lakehouse beyond the pre-aggregated metrics in PostgreSQL (Gold Layer).
- Queries are executed asynchronously and results are published once available.

Internal Structure

Component	Responsibility
Query Event Consumer	Listens for incoming complex query requests from the Lakehouse Complex Query MQ.
Complex Query Orchestrator	Manages query execution by delegating queries to the appropriate components.
Query Executor	Holds the business logic for executing Spark-based analytical queries against the Data Lakehouse.
Query Result Producer	Publishes the processed query results back to the Lakehouse Complex Query MQ.
Delta Reader	Reads required data from the data lakehouse by calling the correct Delta Lake tables.
S3 Api Client	Provides read and write access to files stored in S3.

Interfaces

Interface	Direction	Purpose
Data Lakehouse Complex Query MQ	**Used**	MQ for executing complex queries on the data lakehouse asynchronously. Queries can be pushed to the MQ. After the query has been executed, the result will be published to the MQ and can be retrieved by consuming the result event. This building block publishes query events and consumes their result.
Data Lakehouse Storage (S3)	**Used**	S3-compatible object store containing the data lakehouse data. Data is stored in files in Apache Parquet format in S3.

6. Runtime View

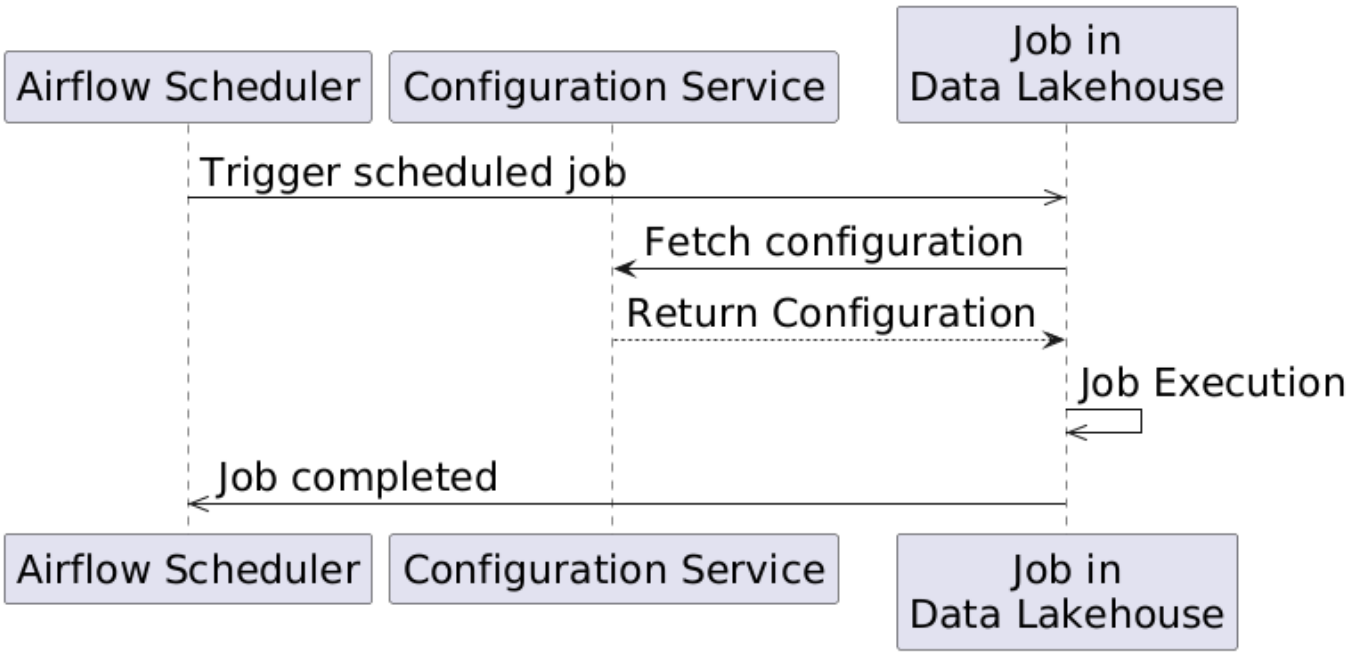
6.1 Scheduling of Jobs in the Data Lakehouse

Scheduling differs between single-tenant and multi-tenant mode:

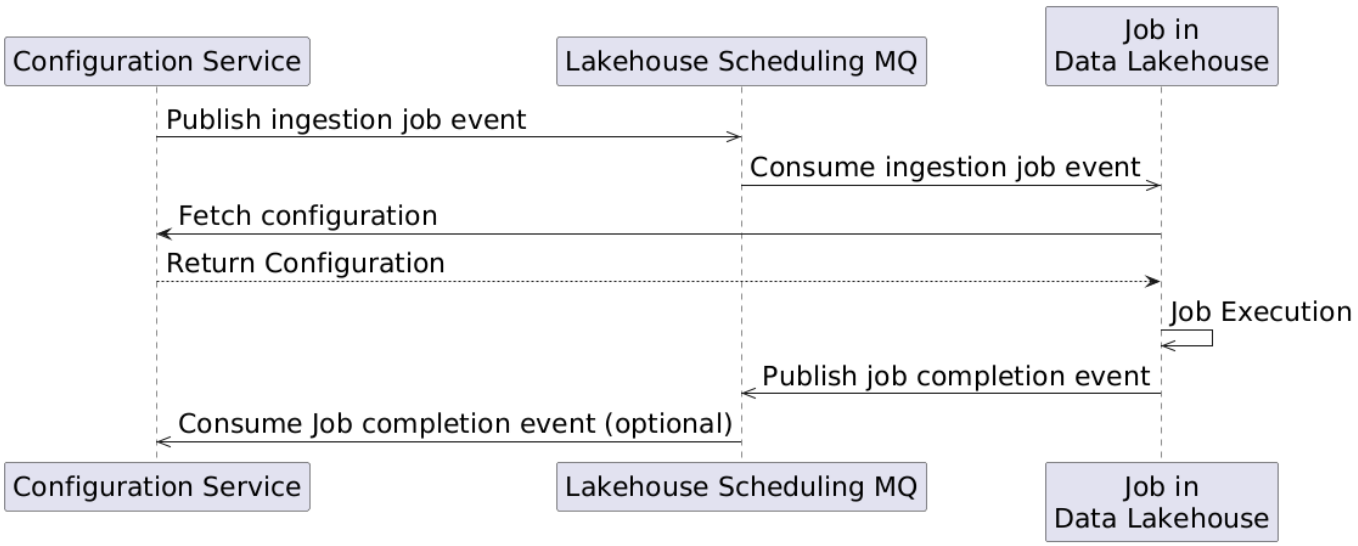
- Single-Tenant Mode (Airflow): Jobs are scheduled and started via Apache Airflow.
- Multi-Tenant Mode (Data Lakehouse Scheduling MQ):
    - Scheduling is managed by the Configuration Service.

- If a data ingestion job needs to be executed, the Configuration Service publishes a job event with the job type *data ingestion* and necessary information to identify the correct configuration data
- The data ingestion service consumes the job event and performs the data ingestion task
- A job completion event is published after the job has finished
- The job completion event can be consumed and used by the scheduling service for subsequent processing (start another job after data ingestion completion if required)

**Sequence Diagram Single-Tenant Mode - Airflow



Sequence Diagram Multi-Tenant Mode – MQ

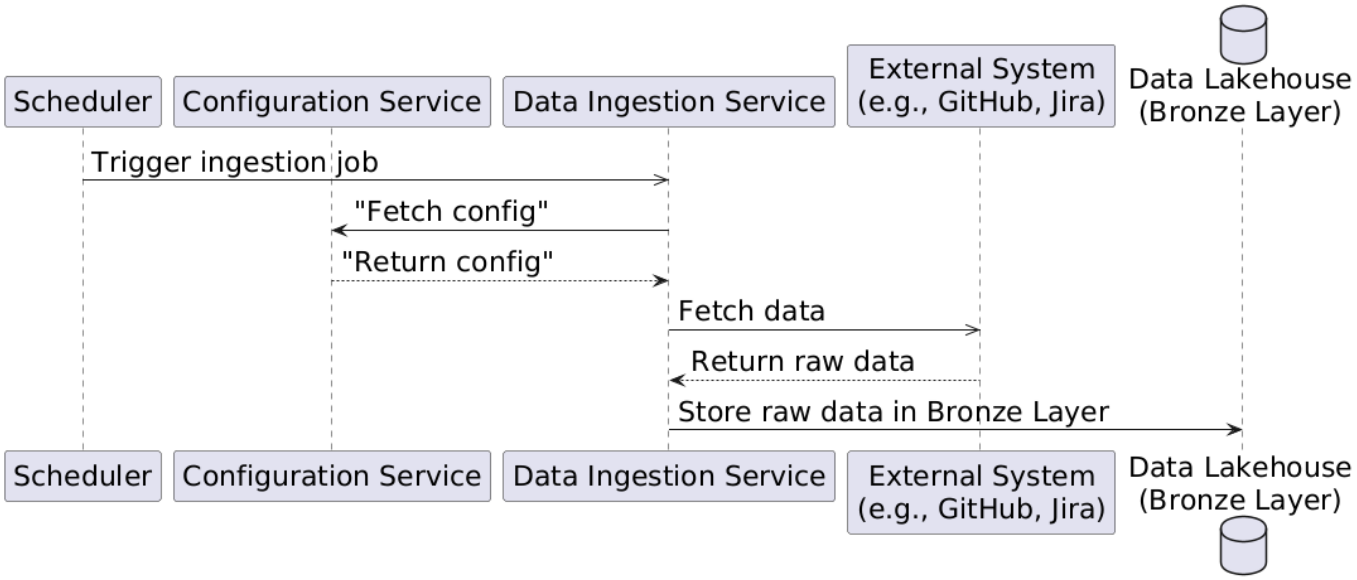


6.2 Data Ingestion Workflow

Description

The Data Ingestion Workflow is responsible for ingesting software development data of various sources to the system and storing this raw data in the Bronze layer of the data lakehouse. Required configuration is retrieved by using the API of the Configuration Service. Dependent on the trustworthiness of the connected data source a virus scan ensures that the data to be ingested is not compromised.

Sequence Diagram



6.3 Data Integration

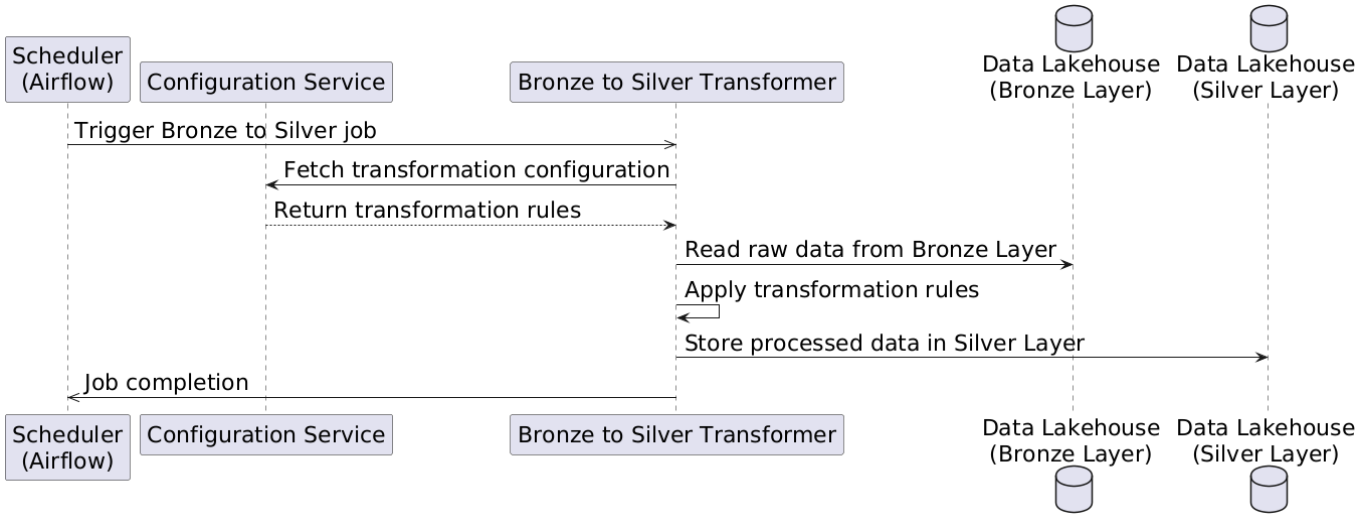
Description

According to the Medallion Architecture there exists two major integration steps:

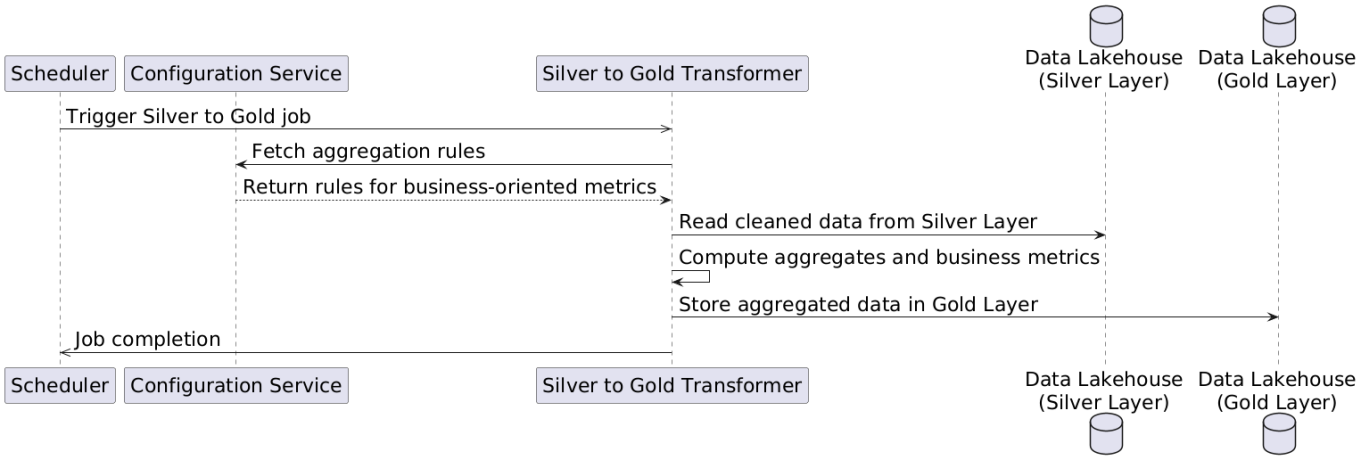
1. Bronze to Silver Integration: Raw data from various sources is cleaned, normalized and transformed into a unified format
2. Silver to Gold Integration: Business-oriented aggregates are computed given cleaned data of the Silver layer and loaded to the Gold layer

The integration steps differ by their transformation rules, but they are similar from a high-level point of view.

Sequence Diagram Bronze to Silver Integration



Sequence Diagram Silver to Gold Integration

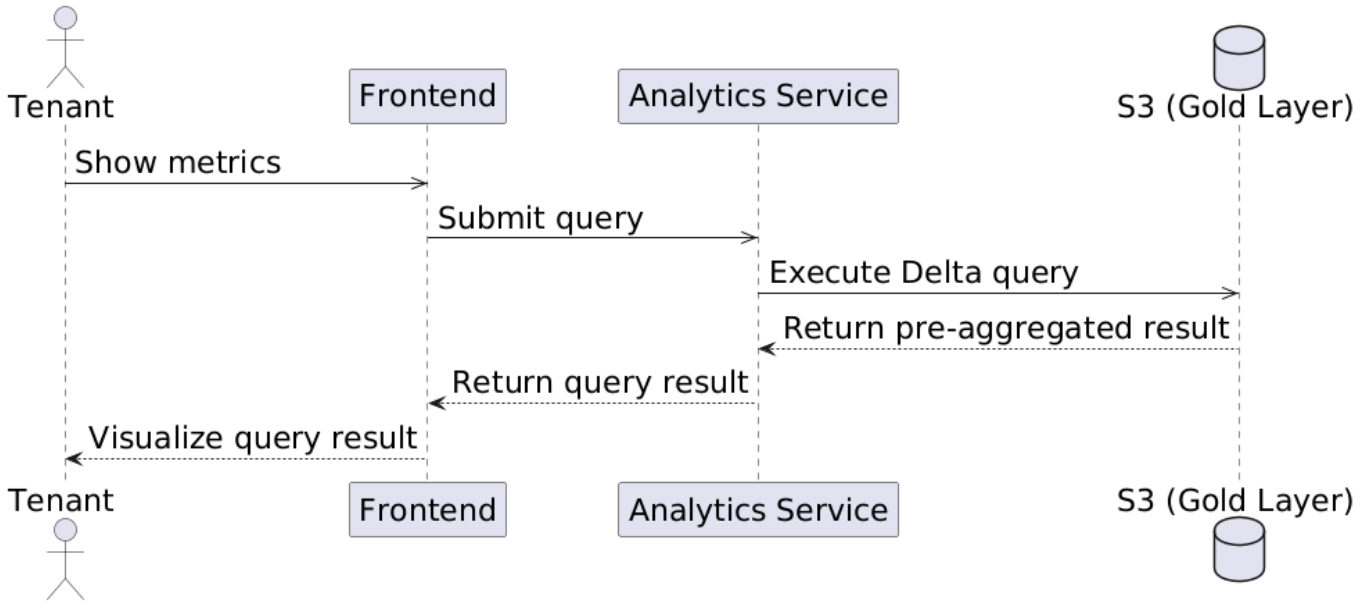


6.4 Fast Query Execution

Description

The Fast Query Execution targets low-latency frequent queries to support performance critical application such as dashboards or reporting. Data is retrieved from the Gold layer of the data lakehouse directly from S3. Retrieved are pre-computed aggregates such as metrics. This approach offers quick response times for frequently accessed data.

Sequence Diagram

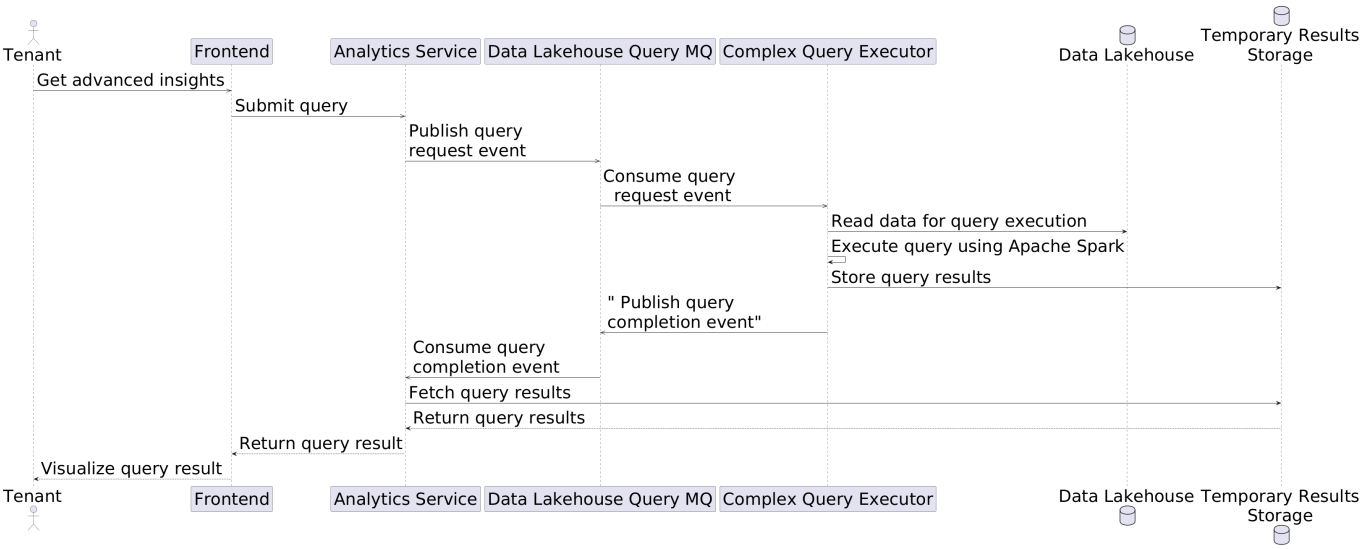


6.5 Complex Query Execution

Description

The Complex Query Execution functionality allows users to perform advanced long-running and resource-intensive analytical tasks. The asynchronous and if required distributed query processing allows correlations on large datasets of various sources and historical analysis.

Sequence Diagram



6.6 Configuration Update

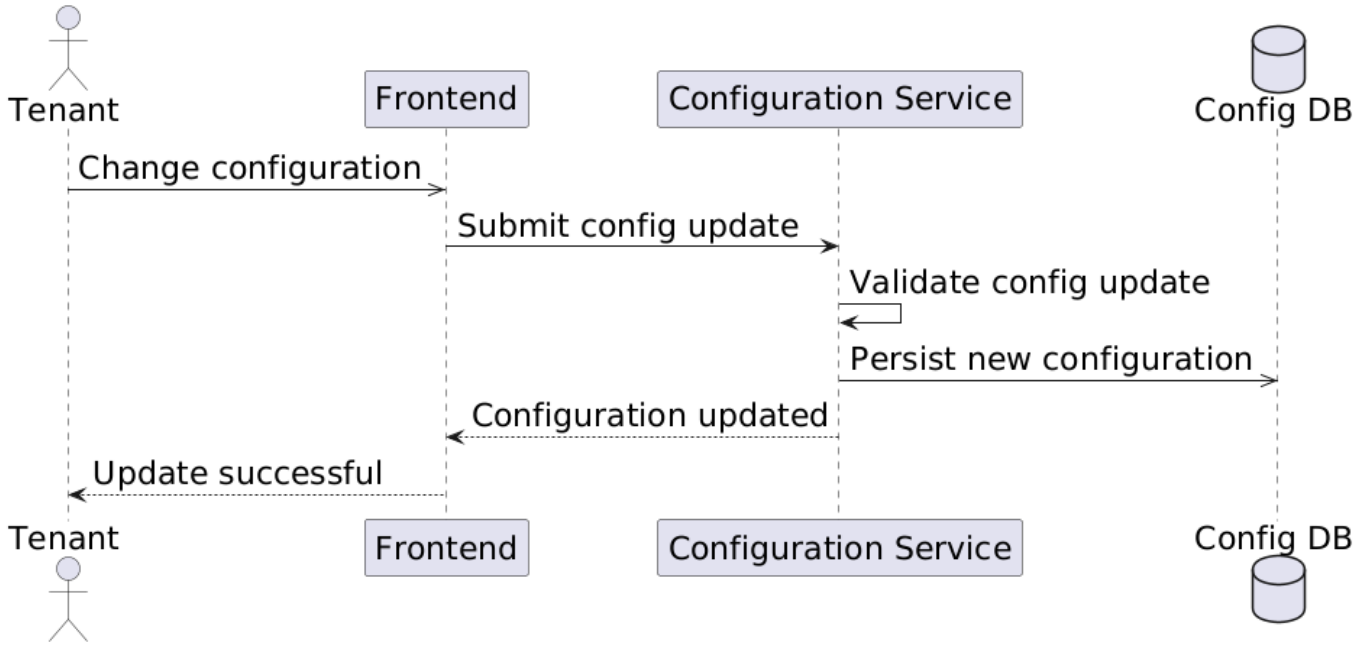
Description

The Configuration Update Workflow offers tenants the possibility to update the configuration of jobs running in the data lakehouse.

Configuration changes can include:

- Data ingestion settings
  - Define which data sources to include
  - Define required credentials such as API keys
- Data integration transformation rules
  - Define which columns are to be dropped
  - Partly control the creation of the unified format and identity recognition
- Scheduling setting: Control job schedules

Sequence Diagram

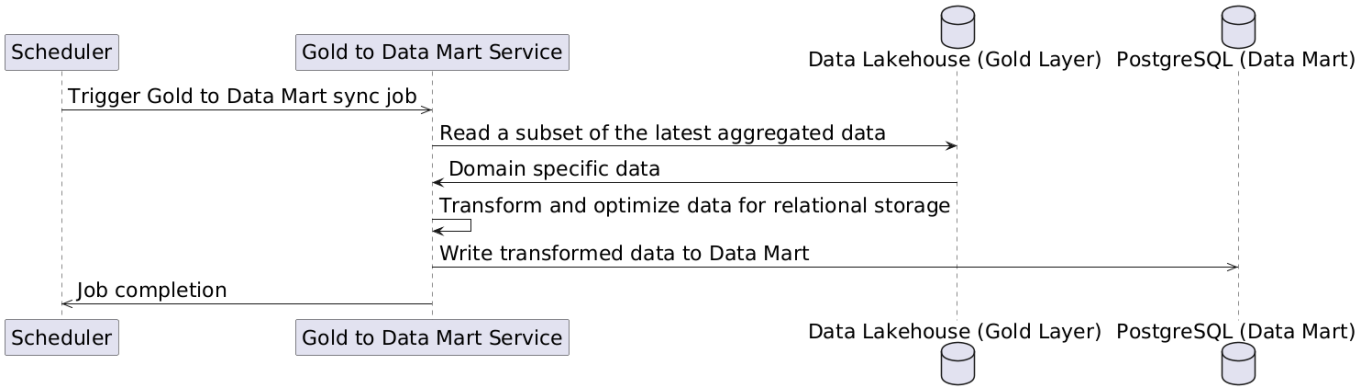


6.7 Data Sync from Gold to Data Mart

Description

The data mart is designed to be optional as is this synchronization job. The Data Sync from Gold to Data Mart Workflow ensures that frequently accessed business-oriented aggregates from the Gold layer are synchronized with a PostgreSQL that behaves as a data mart. This offers time critical applications to fastly access frequently read data such as metrics from a separate data store as query execution in a data lakehouse can be rather slow.

Sequence Diagram



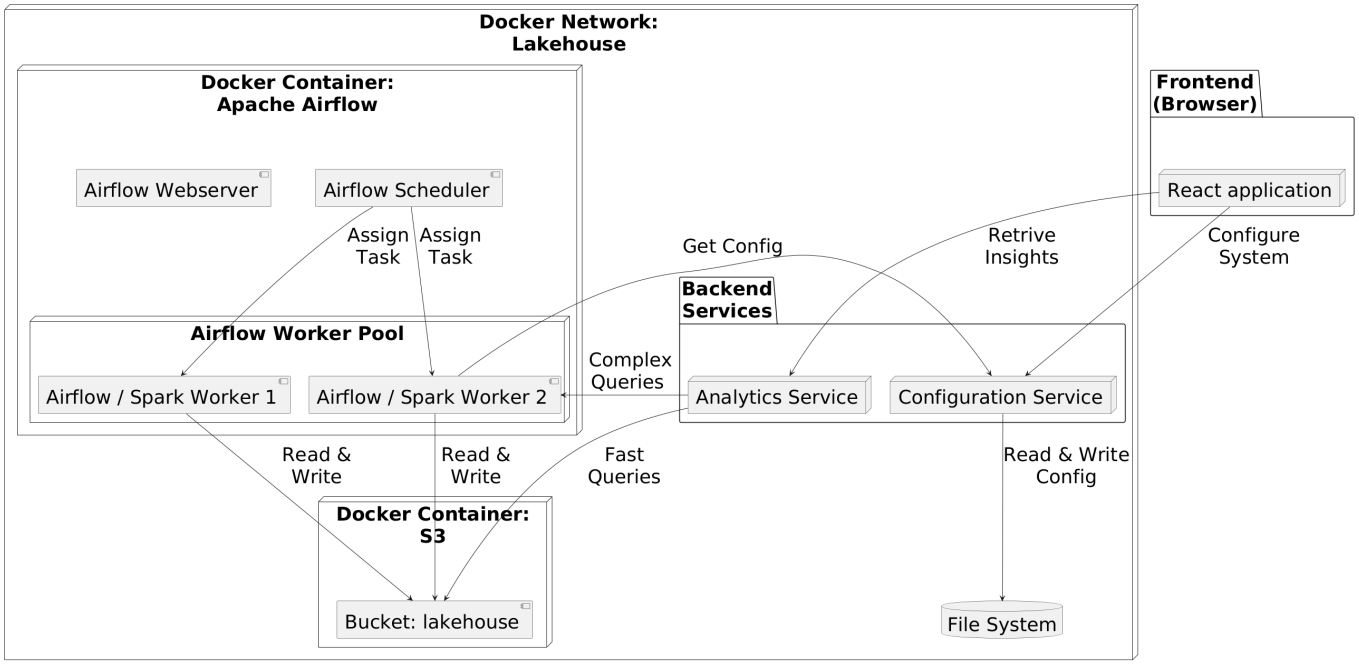
7. Deployment View

The system is designed for modular, independent and containerized deployments using Docker Compose. OnPremises installation or a deployment to a cloud computing environment are both possible options.

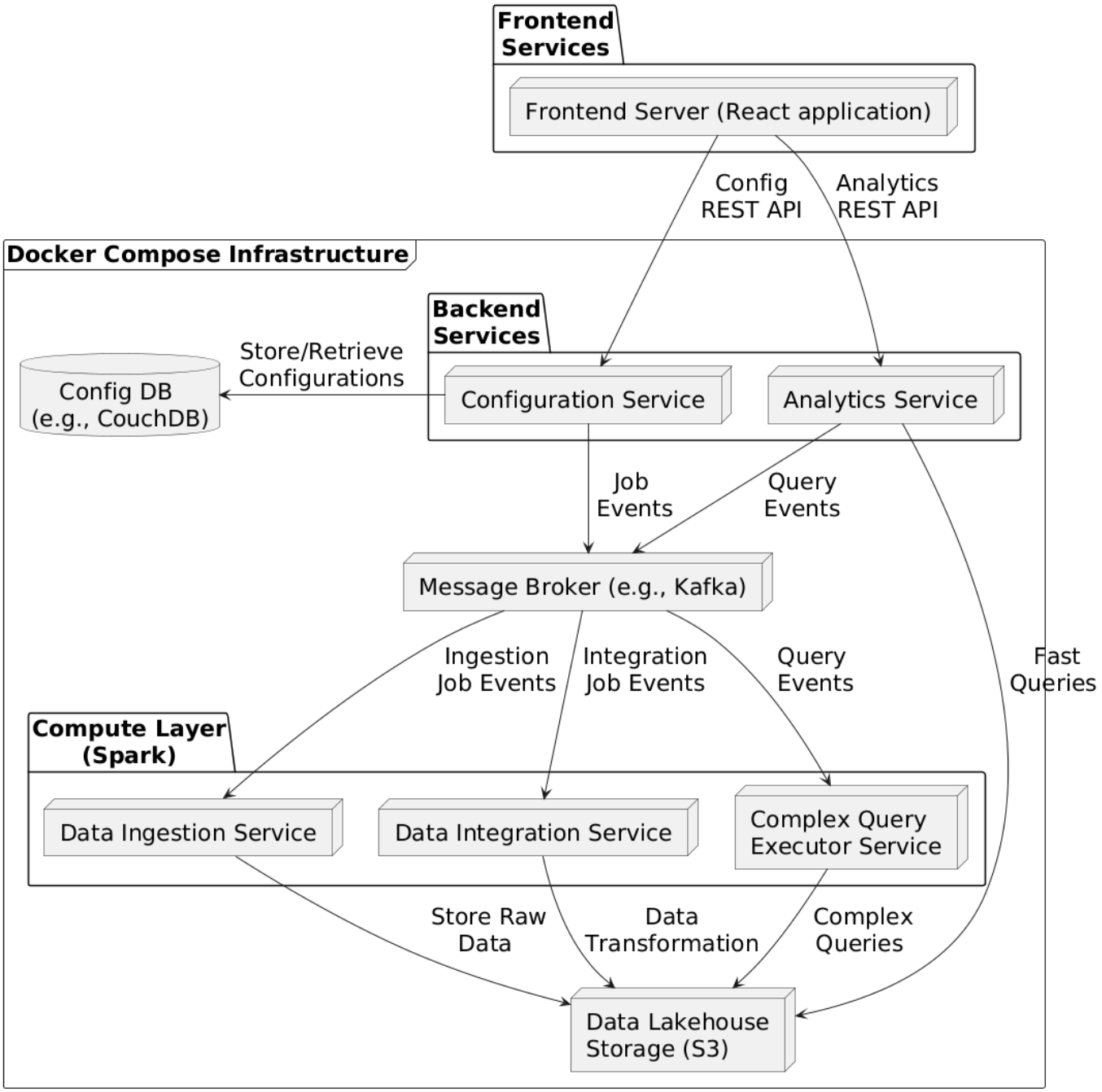
The deployment differs significantly between the single-tenant and the multi-tenant solution.

Deployment View

Single-Tenant Solution



Multi-Tenant Solution



Quality and/or Performance Features

- **Flexibility of Storage**: The system can operate based on a HDFS based file system or based on a S3 like cloud object storage. The Storage solution can be chosen based on the computing environment in which the system should run (OnPremises installation vs cloud vs hybrid)
- **Event-Driven Execution**: Asynchronous, scalable job orchestration via Apache Kafka.
- **Decoupled Architecture**: All infrastructure elements are independently of each other allowing horizontal scaling.
- **Multi-Tenancy & Security**:
  - Analytics Service and Configuration Service enforce access control.
  - Configuration Service manages tenant-specific configurations.
  - Data isolation is handled logically using Delta Lake. Physical isolation is possible but dependent on the choice of the Data Lakehouse Storage technology.

Mapping of Building Blocks to Infrastructure

Building Block	Infrastructure Component
Frontend	React app served via Nginx running in a Docker container
Analytics Service	Node.js / Express backend service running in a Docker container
Configuration Service	Node.js / Express backend service running in a Docker container
Data Ingestion	Python/Spark using service running in a Docker container
Bronze To Silver Integration	Part of the Python/Spark based *Data Integration Service* that runs in a Docker container
Silver To Gold Integration	Part of the Python/Spark based *Data Integration Service* that runs in a Docker container
Gold Data Sync	Part of the Python/Spark based *Data Integration Service* that runs in a Docker container
Complex Query Executor	Python/Spark using service running in a Docker container
Kafka Message Broker	Kafka message broker running in Docker that offers the Data Lakehouse Scheduling MQ and Data Lakehouse Query MQ topics
Apache Airflow (Single-Tenant Mode)	Apache Airflow running in a Docker container
Data Lakehouse Storage	SeaweedFS object store or S3 (cloud) based. Stores data using Delta Lake.
Data Mart	PostgreSQL container
Config DB	CouchDB container

8. Cross-cutting Concepts

8.1 *Data Vault-based Silver Layer*

Data Vaults consist of three entity types - Hubs, Links and Satellites:

- **Hub** entities do only contain business keys with some additional information (metadata). Their aim is to ease identifying business objects as they keep all business keys separately from contextual information. As metadata, often load date and record source are included. The load date is the timestamp when the data was loaded into the data warehouse. The record source identifies the original source of the ingested data. Including this metadata allows for better tracing problems.
- **Links** represent the relationship between these business keys by connecting Hubs. Describing the relationship (Links) between business keys (Hubs) is essential in the data vault model, but still both relationships and business keys need to be enriched with contextual information.

- **Satellites** hold the information describing business objects or the relationship between business objects and are associated to Hubs or Links.

Config File excerpt

The algorithm to transform Bronze data to a Data Vault offers the possibility to define the mapping of data sources to Hubs, Links and Satellites in a single YAML-file.

```
Sources:
  - name: commit
    entities:
      - hub_name: hub_commit
        business_keys:
          - sha
        satellite_name: sat_commit
        attributes:
          - commit_message
          - commit_author_date
        attribute_renames:
          commit_author_date: commit_authored_date
      - hub_name: hub_user
        business_keys:
          - author_login
        satellite_name: sat_user_author
        attributes:
          - author_id
          - author_name
    links:
      - link_name: link_commit_author
        hubs:
          - name: hub_commit
            business_keys:
              - sha
          - name: hub_user
            business_keys:
              - author_login
```

Mapping a Commit to Data Vault Entities

Each data source can contain multiple entities. Consider a `commit` from GitHub or GitLab. Conceptually, this `commit` can be decomposed into a primary entity: `entity_commit`.

This entity may include fields such as:

- `sha`: a unique identifier for the commit (used as a **business key**)
- `commit_message`: contextual information about the commit

Based on this business object, a **Hub** table `hub_commit` is created. It contains the business key `sha` to uniquely identify each commit. The contextual information such as `commit_message` is stored in an associated **Satellite** table called `sat_commit`, which includes:

- A foreign key reference to `hub_commit`
- Contextual attributes (e.g., `commit_message`)
- Metadata like `load_date` and `record_source`

Mapping User Entity

The original `commit` object typically contains additional nested information. One such business object is the `user_entity`, representing the author of the commit.

This may include:

- `author_name`: the name of the commit author
- `author_login`: the user's unique login or ID

A separate **Hub** table `hub_user` is created to store the business key `author_login`. Descriptive attributes such as `author_name` are stored in an associated **Satellite** called `sat_user_author`.

Modeling Relationships with Link Tables

To model the relationship between the `commit` and its `author`, a **Link** table is introduced: `link_commit_author`.

This table:

- References both `hub_commit` and `hub_user` via their business keys
- Establishes a clear connection between business objects
- Includes metadata such as `load_date` and `record_source`

This structure enables flexible and robust integration of development data from heterogeneous systems like GitHub, GitLab, and Jira using the Data Vault modeling pattern.

---

9. Architecture Decisions

The following chapter discusses key architecture decisions using the **Architecture Decision Record (ADR)** format.

9.1 Design Decisions

9.1.1 Data Lakehouse Architecture

Context

Software development data is available from various sources in large volumes (big data), requiring efficient data storage and processing. The application must support:

- Data ingestion to retrieve data from multiple external data sources in different serialization formats.
- Data integration to combine and normalize the data from different systems.

- Data analytics to gain insights from data correlation and present them to end users.

Data ingestion and data integration may differ from tenant to tenant as some data sources may not be available for all tenants or they require different configurations.

To tackle these challenges the following alternatives were considered:

Alternative	Pros	Cons
Data Warehouse	- Optimized structured querying - Traditional BI applications	- Expensive - Less flexible - Requires schema-on-write
Data Lake	- Scalable - Low-cost storage - Schema-on-read	- No ACID transactions - Harder schema management
Data Lakehouse	- ACID transactions - Scalable - Supports schema evolution - Combines advantages of data warehouses and data lakes	- Additional complexity in setup and orchestration

Decision

A data lakehouse architecture is used to

- Separate data ingestion and integration steps to ensure flexibility for different tenants
- Support schema evolution while maintaining data consistency especially at the highest integration level
- Enable horizontal scalability to handle large data volumes

Status

Approved

Consequences

- Technical Impact
    - Ensures scalable processing of data
    - Supports schema evolution and transactional consistency
    - Requires an engine for large-scale data processing such as Apache Spark
    - Need of a storage framework to support schema evolution such as Delta Lake
- Operational Impact
    - More complex job scheduling and orchestration
    - Increased complexity of monitoring

9.1.2 Customizable Data Lakehouse Configuration

Context

Different tenants may require different configuration in the following domains:

- **Data ingestion**: Different tenants use different systems or may use only a subset of the systems that are supported (e.g., GitHub, Jira, Confluence).
- **Data integration**: As different data sources are supported and these data sources may be configured differently some integration rules need to be adjustable (e.g. integration of different data format)
- **Scheduling and Orchestration**: Tenants may define different schedules for ingestion and integration jobs
- **Access control**: API keys and credentials of tenants differ and require separate and secure storage

To allow customizable configurations per tenant the system must be able:

- To perform CRUD operations on the configuration of a tenant.
- To adapt job scheduling based on the tenant's specific settings.
- To access the configuration from multiple jobs
- To maintain consistency of configuration data (all jobs should see the same configuration at the same time)

To address this, the following alternatives were considered:

Alternative	Pros	Cons
Centralized static configuration in config files	- Simple setup - Low operational overhead	- No flexibility - Not scalable - Cannot support tenant-specific configurations such as different API keys
Separate configuration Storage	- Separation of configuration and real data, configuration management independent of jobs in the data lakehouse, if config is separated from the rest of the data lakehouse multiple versions can exist enabling different configurations for multiple tenants	- Synchronisation of configuration and jobs needed, Communication problems, Complexity to decide when to use which configuration

Decision

Create a microservice, the *Configuration Service*, that stores and manages tenant-specific configurations separately of the data lakehouse.

This approach ensures:

- Different configurations for multiple tenants
- Tenants can access and update their configuration concurrently.
- Flexible configuration updates without affecting the data lakehouse.

Status

Approved

Consequences

- Technical Impact
    - Configuration queries are handled efficiently through a (REST) service for low-latency access.
    - Scalable configuration handling as each tenant can access and update it`s configuration independently and concurrently
    - Configuration management independent of jobs in the data lakehouse
        - Configuration changes do not affect jobs running in the data lakehouse
        - If a job accesses the configuration and the configuration is altered directly after the job has accessed it, the job is still running with the old configuration
        - Jobs in the data lakehouse only need read access to the configuration
    - Complexity of secure storage of credentials can be handled separately by the Configuration Service.
    - Requires a clear versioned interface of the Configuration Service for jobs running in the data lakehouse. Jobs must be able to process the configuration that the Configuration Service provides.
- Organizational Impact
    - Configuration Service as single point of truth for configurations of the data lakehouse
    - Easier administration and management of configurations via a structured API.
- Operational Impact
    - Operational Overhead due to maintaining a separate service (Configuration Service).
    - Long-running jobs in the data lakehouse may still run with an old configuration. If required long-running jobs must be informed about configuration changes via a MQ or poll their config status.

9.1.3 Configuration Management in Separate Database

Context

The Configuration Service as introduced in 9.1.2 is responsible for storing and managing tenant-specific configurations separately from the data lakehouse.

To fulfill this requirement the Configuration Service must:

- Store and retrieve tenant-specific configurations efficiently to provide low-latency responses to readers (jobs running in the data lakehouse).
- Ensure transactional consistency to avoid conflicts in concurrent configuration updates.
- Securely store credentials such as API keys.
- Isolate sensitive data of tenants such as credentials from other tenants and ensure no unauthorized access to sensitive data is possible

The following options were considered as storage for configuration data:

Alternative	Pros	Cons
Store configurations inside the Data Lakehouse (Delta Lake/S3)	- Reuse of an already existing storage - Schema evolution supported	- Slow configuration retrieval - Not optimized for efficient frequent low-latency reads - CRUD operations of tenants on data in the data lakehouse
Use a Separate Database (e.g., relational, document-based)	- Fast low-latency data access - Transactional consistency (ACID guarantees) -Allows Querying of parts of the configuration using a query language	- Requires maintaining an additional database
Use a Distributed Key-Value File Storage (S3)	- High availability - Simple	- Limited query capabilities - Not ideal for managing structured configuration settings - Lacks ACID transactions
Store configuration files at the disk of the Configuration Service	- Simple and fast	- Not scalable - No transactional support of a database system -No schema enforcement -Difficult to offer configuration changes at runtime

Decision

The Configuration Service has as a microservice the single responsibility for managing configuration data. For a single-tenant solution runtime updates are not a strict requirement as the solution is initially configured. It is expected that changes happen only rarely afterward. Therefore, it is acceptable to redeploy the configuration service for once if configuration changes are required.

If the solution is enhanced by multi-tenancy, configuration should be stored in a separate database managed by the Configuration Service. The Configuration has the responsibility for managing this database and providing secure access control to it. This strategy allows concurrently updating configuration data of multiple tenants at runtime. Other tenants are not affected by a tenants configuration changes. As this involves changes of configuration data at runtime, additional content validation measures to validate configuration updates must be introduced.

This approach ensures:

- Efficient low-latency access for frequently read data
- Separation of configuration and real data as configuration data is stored independently of the data lakehouse.
- Consistency of configuration data

- Scalability as multiple tenants can access and modify configurations concurrently.

Status

Approved

Consequences

- Technical Impact:
    - Fast, optimized configuration reads improve ingestion and integration job performance. The Configuration Service must offer low-latency access to configuration data by a clearly-defined (REST) interface.
    - Read/write separation:
        - Jobs running in the data lakehouse only need read access to configuration data.
        - In single-tenant mode the Configuration Service has to be redeployed if configuration changes.
        - In multi-tenant mode the Configuration Service handles updates of the configuration data to ensure consistency. It should offer a Management API with which users can alter configuration data.
        - The Configuration Service is responsible to control authorized access to sensitive tenant-specific data
    - Credentials need to be stored securely. This complexity must be tackled using database mechanisms (dependent on the used database) or by the Configuration Service
- Organizational Impact:
    - Improved multi-tenant data handling allowing different configurations per tenant without affecting other tenants.
    - Single Responsibility principle
        - The Microservice Configuration Service acts as single source of truth for all configurations.
        - Configuration handling can be developed independently of the data lakehouse
    - Simplifies isolation of configuration data from other tenants and secure configuration retrieval.
- Operational Impact:
    - In multi-tenant mode, overhead due to maintaining a separate database.
    - Database as potential single point of failure if the database is deployed non-replicated -> Deployment as cluster for high-availability
    - Additional backup and disaster recovery procedures for the separate Config DB required

---

9.1.4 Query Processing

Context

To present analytical insights to end users, the system must support two types of queries:

- **Fast Queries**
    - Queries that return pre-aggregated data such as metrics efficiently to end users.
    - Query runtime is critical to allow applications such as dashboards.
    - Metrics can be pre-aggregated and do not need to include the latest data from every data source. Regularly updating metrics is sufficient.
- **Complex Queries**
    - Long-running asynchronous query processing supporting advanced correlations.
    - May require batch processing and distributed computing.
    - Time-intensive computations such as correlating multiple data sources or historical analysis.

The following alternatives were considered for query handling:

Alternative	Pros	Cons
Perform all queries on a separate database: Loading required data to a database with high read performance and perform queries exclusively against this database, e.g., a PostgreSQL that works as a data mart or a PostgreSQL that mirrors the Gold layer of the Medallion architecture	- Efficient and easy fast query handling - Simple	- Does not support asynchronous complex queries - Not scalable for large-scale analytics - No batch processing
Perform all queries on the data lakehouse: Handle all queries in the data lakehouse using Spark.	- Execution of both fast and complex queries in the data lakehouse leveraging Spark - Scalable processing - No additional complexity due to a separate storage	- High latency for real-time queries - No optimization for frequent queries
Hybrid Approach: Fast queries against data marts, complex queries against the data lakehouse	- Optimized for both real-time and complex queries - Efficient separation of queries against pre-aggregated frequently accessed data and rarely used complex queries	- Additional complexity in query routing - Synchronization between Data Mart and Data Lakehouse necessary

Decision

While a hybrid architecture using a separate relational database for fast queries was initially considered, a walkthrough of the software architecture with an experienced software architect led to the conclusion that this additional complexity should only be introduced if strictly required.

A dedicated relational database for fast queries introduces additional complexity into the architecture. It is dependent on the actual use case whether this complexity is really necessary or if a well-designed Gold layer is sufficient to meet the desired performance requirements. For that reason, in the solution an approach is adopted where all queries are executed against the data lakehouse. If strict performance requirements cannot be met with a Gold layer in the data lakehouse, the solution should be extended by separate fast and complex query processing. This dedicated layer for fast query processing is modeled optionally and involves the mirroring of pre-aggregated Gold layer data that is often accessed to data marts based on business requirements.

Status

Approved

Consequences

- **Technical Impact**
    - The system avoids the additional complexity of introducing a separate database for fast query processing. It relies entirely on Spark-based query processing against the data lakehouse or, if necessary, direct Parquet-based data access.
    - A dedicated building block, the Analytics Service, is required that handles secure data access and authorization to Gold layer data.
    - The Analytics Service does not need to differentiate between fast and complex query processing, which eases the building block's implementation.
    - However, if query performance becomes a bottleneck, the Analytics Service may need to be extended by separate fast and complex query processing.
    - Beforehand, query optimization techniques should be applied to the Gold layer, such as pre-aggregations, materialized views, and other query optimization techniques.
- **Organizational Impact**
    - Users interact with the system through interfaces provided by the Analytics Service. The Analytics Service provides an interface that enables secure access to Gold layer data.
    - This interface can be accessed directly by users or used as a basis for dashboards.
    - The way users like developers retrieve data from the data lakehouse is different from traditional SQL usage with which they are familiar.
    - Documentation and knowledge transfer must ensure that users understand the differences in data access and the data model of the data lakehouse.
- **Operational Impact**
    - System complexity is reduced, because there is no need to provision or monitor a separate relational database.

    - Keeping the data in the data lakehouse also has the advantage that no synchronization pipelines are required between the Gold layer and external systems, which reduces maintenance costs.
    - All data remains centralized within the lakehouse.
    - This further reduces data duplication, consistency, or data staleness issues and has a positive impact on costs.
    - Observability can focus on the performance of Spark jobs.

9.1.5 Scheduling and Orchestration in Single-Tenant and Multi-Tenant Mode

Context

Multi-tenancy adds additional complexity to the system. Therefore, for prototypical implementations it may be beneficial to ignore this additional complexity at first and implement a single tenant system and add multi-tenancy support later. Some tenants may also require an isolated OnPremises deployment for which multi-tenancy support is not a requirement and handling of a single-tenant system might be easier.

The data lakehouse requires scheduling and orchestration of data ingestion, integration, and transformation jobs. These steps differ between the single-tenant and multi-tenant solution:

- **Single-Tenant Mode**
    - A single tenant has control over the entire system and has access rights to connected data sources. Data ingestion and transformation schedules can be completely controlled by the tenant.
    - The system can rely on a centralized workflow orchestrator to schedule and orchestrate jobs.
    - Less security concerns as jobs and data do not need to be isolated from other tenants
- **Multi-Tenant Mode**
    - Multiple tenants run jobs in the data lakehouse concurrently.
    - As multiple tenants share the same infrastructure, resources need to be allocated equally.
    - For scalability reasons a scalable, distributed job orchestration mechanism is required.
    - Different tenants may have conflicting scheduling requirements.
    - Jobs and data of tenants must be isolated from unauthorized access of other tenants.
    - Issues in multi-tenant systems such as tenant resource starvation may arise (one tenant consuming too many resources).

Two possible solution designs were considered:

- Central workflow orchestrator:
    - There exist tools on the market (Apache Airflow) to schedule and orchestrate jobs running on Spark
    - Apache Airflow as industry-proven open source solution with a user-friendly user interface
- Event-driven scheduling and orchestration:
    - Jobs are triggered based on events from a *Data Lakehouse Scheduling MQ*. Job orchestration is implemented via ordering of the messages in the MQ.
    - Scheduling is realized by a component that publishes a job event if this job needs to run. As the configuration service also holds scheduling configurations, it is easy to realize this functionality as part of the configuration service.

- If required jobs inform scheduler via MQ about their completion

Alternative	Pros	Cons
Central workflow orchestrator for single-tenant and multi-tenant mode (e.g., Apache Airflow)	- No differentiation between single-tenant and multi-tenant mode - Adapt existing industry-proven solutions such as Apache Airflow	- Existing solutions lack support of strict tenant-level isolation (Apache Airflow) - Difficult to control a tenants access to resources as a tenant should not consume too many resources. - Expensive implementation if existing systems can not be used. - It may makes sense to hold data ingestion schedules in Apache Airflow what contradicts the central configuration handling via the Configuration Service.
Centralized workflow orchestrator for single-tenant mode and an event-driven approach for multi-tenant mode	- Leverage existing solutions in single-tenant mode - Ensures tenant isolation in multi-tenant mode	- Additional complexity in managing two orchestration mechanisms -Jobs need to be implemented differently whether they are orchestrated by a MQ or by Apache Airflow.
Fully event-driven scheduling and orchestration	- Highly scalable - Works well for single and multi-tenancy - Ensures tenant isolation in multi-tenant mode - Moderate implementation effort	- Unnecessary complexity for single-tenant setups (need of a queue) - Single-tenant solution could leverage benefits of existing centralized orchestration engines. - Lack of support of advanced features such as stopping of currently running jobs.

Decision

Try realizing the single-tenant system at first and realize scheduling via Apache Airflow. Then add multi-tenancy support via an MQ. If centralized workflow engines get real multi-tenancy ensuring strict tenant-level data isolation realize multi-tenant architecture also with a centralized workflow engine.

- **Single-Tenant Mode**
  - Jobs are scheduled using a centralized workflow orchestrator (Apache Airflow).
  - Fine-grained control over job execution by leveraging an industry-proven workflow orchestration solution
  - Scheduling using Cron-Expressions in Airflow
- **Multi-Tenant Mode**

- Event-driven scheduling and orchestration using a message queue
- A scheduling component, the Configuration Service, publishes job events to a MQ (Lakehouse Scheduling MQ).
- Events published by the scheduler include information to identify a tenants configuration (e.g. tenantId)
- The job events are executed by workers running on Spark using the configuration gathered from the configuration service
- Jobs are processed independently per tenant ensuring data isolation.

Status

In Discussion

As the system is difficult to implement, an implementation of the single-tenant system at first using Apache Airflow and later enhancing it via a MQ may make sense. On top of that, according to polls multi-tenancy support is in discussion for such centralized workflow orchestration tools, though this feature is complicated to implement.

Consequences

- Technical Impact:
  - Executors may require a different implementation (with MQ, without MQ)
- Organizational Impact:
  - Developers and data engineers must understand two different scheduling approaches.
- Operational Impact:
  - Deployment differs between single-tenant and multi-tenant mode.
  - In multi-tenant mode
    - Monitoring required to ensure jobs do not block each other.
    - Requires resource allocation policies to prevent tenant resource starvation.

9.2 Technology Choices

9.2.1 Data Lake Storage System

Context

Data Lakehouse Storage Frameworks store information in specialized data storage formats, such as Apache Parquet or ORC. Consequently, data lakehouses require a scalable storage system in which these files can be stored physically. The modern data lakehouse often uses a low-cost object store such as an Amazon S3-compatible system as physical storage as suggested by Armbrust et al. (2021). But also OnPremises

deployments based on HDFS are possible. This architecture decision discusses the impact of the choice of the physical data storage and compares the two storage systems.

The following alternatives were considered for data storage:

Alternative	Pros	Cons
Cloud Object Store (S3)	- Scalable - Cost-effective - Widely supported - Compatible with Delta Lake - Easy expansion and dockerized deployments	- Network latency - Requires full S3 API compatibility
HDFS	- Performance advantages due to data locality - High scalability	- Higher operational complexity - Higher costs compared to S3 - More complex deployment and management

Decision

The solution adopts an S3-based approach, due to cost-benefits, the simplicity of this approach, simple local dockerized deployments and because Delta Lake is designed to be used with S3. For the development phase of the solution, a dockerized deployment is seen as beneficial in contrast to having to maintain machines with HDFS installations.

Status

Approved

Consequences

- **Technical Impact**
    - The adoption of an S3-compatible object store enables cloud-native deployments.
    - Either a cloud provider's S3 solution or an on-premises installation on a server or cluster can be used.
    - Most S3-compatible object store solutions also offer Kubernetes-based or dockerized deployments.
    - An architectural dependency on the storage framework is introduced.
    - The chosen storage framework must support S3, which Delta Lake does.
    - Storage frameworks such as Delta Lake require that the S3-compatible object store fully supports the S3 API to ensure correct transactional processing.
- **Organizational Impact**

- Local dockerized deployments of the storage solution can simplify the development process, as a preconfigured infrastructure setup can be provided.
- Infrastructure management is simplified for teams with limited DevOps resources.

- **Operational Impact**
  - An S3-compatible deployment at a cloud provider enables flexible, cost-effective scaling of storage.
  - Storage can scale up or down on demand.
  - The use of S3 requires a reliable network connection between the object store and Spark.
  - In distributed setups, this connection must be monitored.
  - S3 introduces more latency compared to HDFS, although this is partly mitigated by Delta Lake.

---

10. Quality Requirements

10.1 Quality Tree


Capacity
Handles large data volumes efficiently
Processes data in chunks
Ensures consistency in analytics results
Faultlessness
Error handling for data ingestion
Error handling for data integration
Ensures analytical results are available
Availability
Ensures configurations are available
Frequently accessed data is replicated
Handles processing failures without data corruption
Reliability
Fault Tolerance
Graceful recovery from unexpected errors
Retry mechanism for temporary failures
Offers disaster recovery mechanism
Recoverability
Rollback mechanism for incorrect data ingestion
Self-healing jobs
Can share infrastructure with other services
Co-existence
Supports integration of new data sources
Compatibility
Interoperability
Offers standardized APIs for external tools
Supports different storage technologies
Configurable data sources per tenant
Configurable ingestion & integration scheduling per tenant
Adaptability
Configurable data integration per tenant
Support for schema evolution
Utility
Supports Auto Scaling based on workloads
Flexibility
Scalability
Scales without performance degradation when new tenants are added


Deployable in Cloud & OnPremises infrastructures
Installability
Supports containerized deployments via Docker Compose
Replaceability
Is installable on all commonly used operating systems
Components can be replaced independently
Ensures data of a tenant is isolated from other tenant`s access
Role-based access control
Confidentiality
No correlations across tenants
Credentials are processed securely
Prevents unauthorized modifications
Security
Integrity
Ensures consistency of stored data
Ensures stored data is not compromised by viruses
Accountability
Logs all data accesses & job executions
Ensures auditability
Enforces authentication on REST API requests
Authenticity
Enforces authentication between backend systems
Ingestion Services can authenticate against data sources
Configuration management as module
Modularity
Independent deployment of services
Standardized API generation & documentation
Composable data ingestion pipelines
Reusability
Composable data integration pipelines
Maintainability
Supports Logging
Analyzability
Supports Tracing across components

11. Risks and Technical Debts

The architecture itself does not include known technical debts, so far. However, the Configuration Service, Analytics Service, Frontend and Gold layer have not yet been implemented.

12. Glossary

Term	Definition
Amazon S3 API	*An API for interacting with Amazon S3-compatible object storages, used for storing data lakehouse layers in this project.* *See https://aws.amazon.com/s3/*
Apache Airflow	*Platform to schedule and monitor workflows, used in this project for Spark jobs.* *See https://airflow.apache.org/*

Term	Definition
Apache Kafka	*Distributed event-streaming platform suitable to create fast asynchronous data pipelines and stream based applications. Kafka is utilized in this project for scheduling and orchestration of jobs in the data lakehouse.* *See https://kafka.apache.org/*
Apache Parquet	*Column-oriented data file format used to store and query data from the data lakehouse efficiently.* *See https://parquet.apache.org/*
ADR	*Architecture Decision Record - format to document architecture decisions.* *See https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions*
CRUD	*Abbreviation for the operations Create, Read, Update and Delete on a dataset*
Delta Lake	*An open-source storage layer enabling ACID transactions in Spark. Used for managing the data lakehouse in this project.* *See https://delta.io/*
Docker	*Platform for developing, shipping, and running containerized applications, used in this project to containerize components for deployment.* *See https://www.docker.com/*
Docker Compose	*Util for defining and running multi-container Docker applications. Used in this project to orchestrate component deployment.* *See https://docs.docker.com/compose/*
Express	*A Node.js web application framework used to build APIs for the Analytics and Configuration Services.* *See https://expressjs.com/*
Mecois	*The product name and team name of this project.* *See https://www.mecois.com/*
MQ	*Message Queue for asynchronous event-based processing*
Node.js	*A JavaScript runtime environment. Used as the runtime environment for the Analytics and Configuration Services.* *See https://nodejs.org/*
OpenAPI Generator	*Code generator that is able to generate API clients and server stubs from an OpenAPI specification.* *See https://github.com/OpenAPITools/openapi-generator*
PostgreSQL	*Relational database used for storing configurations and pre-aggregated data for fast querying.* *See https://www.postgresql.org/*
PySpark	*Python API for Apache Spark, used in this project for data ingestion, transformation, and aggregation in the data lakehouse.* *See https://spark.apache.org/docs/latest/api/python/*

Term	Definition
React	*JavaScript library for building single page applications, used in this project for the user interface.* *See https://react.dev/*
Swagger UI	*Tool to visualize OpenApi specifications* *See https://swagger.io/tools/swagger-ui/*
TypeScript	*Typed programming language that builds on JavaScript. Used for both frontend and backend services.* *See https://www.typescriptlang.org/*

Credits - Software architecture template (arc42)

This architecture documentation was created with arc42, the template for documentation of software and system architecture.

Template Version 8.2 EN. (based upon AsciiDoc version), January 2023

References

Ajiga, D., Okeleke, P. A., Folorunsho, S. O., & Ezeigweneme, C. (2024). Enhancing software development practices with AI insights in high-tech companies. *Computer Science & IT Research Journal*, *5*(8), 1897–1919. https://doi.org/10.51594/csitrj.v5i8.1450

Amazon Web Services. (2024). *Amazon Simple Storage Service API Reference*. Retrieved December 31, 2024, from https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html

Amazon Web Services. (2025). *Security best practices on Amazon MWAA - Amazon Managed Workflows for Apache Airflow*. Retrieved April 20, 2025, from https://docs.aws.amazon.com/mwaa/latest/userguide/security-best-practices.html

Amazon Web Services, Inc. (2025). *AWS CLI*. Retrieved April 28, 2025, from https://aws.amazon.com/cli/?nc1=h_ls

Armbrust, M., Ghodsi, A., Xin, R., & Zaharia, M. (Eds.). (2021). *Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics*.

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., Torres, J., van Hovell, H., Ionescu, A., Łuszczak, A., Świtakowski, M., Szafrański, M., Li, X., Ueshin, T., Mokhtar, M., Boncz, P., Ghodsi, A., Paranjpye, S., Senster, P., ... Zaharia, M. (2020). Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proceedings of the VLDB Endowment*, *13*(12), 3411–3424. https://doi.org/10.14778/3415478.3415560

Atlassian. (2025). *Atlassian Data Lake*. Retrieved April 26, 2025, from https://www.atlassian.com/platform/analytics/what-is-atlassian-data-lake#why-use-the-atlassian-data-lake

Begel, A., & Zimmermann, T. (2014). Analyze this! 145 questions for data scientists in software engineering. In P. Jalote, L. Briand & A. van der Hoek (Eds.), *Proceedings of the 36th International Conference on Software Engineering* (pp. 12–23). ACM. https://doi.org/10.1145/2568225.2568233

Belcastro, L., Marozzo, F., Talia, D., & Trunfio, P. (2017). Big Data Analysis on Clouds. In A. Y. Zomaya & S. Sakr (Eds.), *Handbook of Big Data*

Technologies (pp. 101–142). Springer International Publishing. https://doi.org/10.1007/978-3-319-49340-4_4

Buse, R. P., & Zimmermann, T. (2010). Analytics for software development. In G.-C. Roman & K. Sullivan (Eds.), *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 77–80). ACM. https://doi.org/10.1145/1882362.1882379

Camacho-Rodríguez, J., Agrawal, A., Gruenheid, A., Gosalia, A., Petculescu, C., Aguilar-Saborit, J., Floratou, A., Curino, C., & Ramakrishnan, R. (2024). LST-Bench: Benchmarking Log-Structured Tables in the Cloud. *Proceedings of the ACM on Management of Data*, *2*(1), 1–26. https://doi.org/10.1145/3639314

Capraro, M., & Riehle, D. (2016). Inner Source Definition, Benefits, and Challenges. *ACM Computing Surveys*, *49*(4), 1–36. https://doi.org/10.1145/2856821

Czerwonka, J., Nagappan, N., Schulte, W., & Murphy, B. (2013). CODEMINE: Building a Software Development Data Analytics Platform at Microsoft. *IEEE Software*, *30*(4), 64–71. https://doi.org/10.1109/MS.2013.68

Databricks. (2017). *Top 5 Reasons for Choosing S3 over HDFS*. Retrieved March 14, 2025, from https://www.databricks.com/blog/2017/05/31/top-5-reasons-for-choosing-s3-over-hdfs.html

Databricks Blog. (2022). *Data Warehouse Modeling on Databricks*. Retrieved March 17, 2025, from https://www.databricks.com/blog/2022/06/24/data-warehousing-modeling-techniques-and-their-implementation-on-the-databricks-lakehouse-platform.html

Dixon, J. (2010). *James Dixon's Blog. Pentaho, Hadoop, and Data Lakes*. Retrieved December 11, 2024, from https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/

Docker Inc. (2025). *Docker Compose*. Retrieved April 25, 2025, from https://docs.docker.com/compose/

Dueñas, S., Cosentino, V., Gonzalez-Barahona, J. M., Del Castillo San Felix, A., Izquierdo-Cortazar, D., Cañas-Díaz, L., & Pérez García-Plaza, A. (2021). GrimoireLab: A toolset for software development analytics [Journal Article Santiago Dueñas, Daniel Izquierdo-Cortazar, Luis Cañas and Alberto Pérez García-Plaza are employees of Bitergia, the main contributor to GrimoireLab. Journal Article Santiago Dueñas, Daniel Izquierdo-Cortazar, Luis Cañas and Alberto Pérez García-Plaza are employees of Bitergia, the main contributor to GrimoireLab.]. *PeerJ. Computer science*, *7*, e601. https://doi.org/10.7717/peerj-cs.601

Franch, X. (2022). Software analytics tools: an intentional view. In *Proceedings of the 15th International iStar Workshop (iStar 2022), co-located with 41th International Conference on Conceptual Modeling (ER 2022): virtual event, Hyderabad, India, October 17, 2022* (pp. 29–34). CEUR-WS.org. http://hdl.handle.net/2117/378210

Friedman, A., Dhupelia, R., & Jackson, B. (2023). The Atlassian Data Lake: consolidating enriched software development data in a single, queryable system. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 265–266. https://doi.org/10.1109/MSR59073.2023.00045

Giebler, C., Gröger, C., Hoos, E., Schwarz, H., & Mitschang, B. (2019). Leveraging the Data Lake: Current State and Challenges. In C. Ordonez, I.-Y. Song, G. Anderst-Kotsis, A. M. Tjoa & I. Khalil (Eds.), *Big Data Analytics and Knowledge Discovery* (pp. 179–188, Vol. 11708). Springer International Publishing. https://doi.org/10.1007/978-3-030-27520-4_13

GitHub Repository scality/cloudserver. (2025). *Zenko CloudServer*. Retrieved April 27, 2025, from https://github.com/scality/cloudserver

GitHub Repository seaweedfs/seaweedfs. (2025). *SeaweedFS*. Retrieved April 27, 2025, from https://github.com/seaweedfs/seaweedfs

Gonzalez-Barahona, J. M., Izquierdo-Cortazar, D., & Robles, G. (2022). Software Development Metrics With a Purpose. *Computer*, *55*(4), 66–73. https://doi.org/10.1109/MC.2022.3145680

Google Cloud. (2025). *Interoperability with other storage providers*. Retrieved March 16, 2025, from https://cloud.google.com/storage/docs/interoperability?hl=en

Haelen, B., & Davis, D. (2024). *Delta Lake Up & Running: Modern Data Lakehouse architectures with Delta Lake* [Haelen, Bennie (VerfasserIn) Davis, Dan (VerfasserIn)]. O'Reilly Media Inc. https://learning.oreilly.com/library/view/-/9781098139711/?ar

Harby, A. A., & Zulkernine, F. (2022). From Data Warehouse to Lakehouse: A Comparative Review. *2022 IEEE International Conference on Big Data (Big Data)*, 389–395. https://doi.org/10.1109/BigData55660.2022.10020719

Harby, A. A., & Zulkernine, F. (2025). Data Lakehouse: A survey and experimental study [PII: S0306437924001182]. *Information Systems*, *127*, 102460. https://doi.org/10.1016/j.is.2024.102460

Hassan, A. E., & Xie, T. (2010). Software intelligence: The Future of Mining Software Engineering Data. In G.-C. Roman & K. Sullivan (Eds.), *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 161–166). ACM. https://doi.org/10.1145/1882362.1882397

Hirsch, J., Riehle, D., Wolter, T., & Keskinoglu, T. (2025). *MECOIS - Engineering intelligence and developer empowerment*. Retrieved February 28, 2025, from https://www.mecois.com/

Huijgens, H., Rastogi, A., Mulders, E., Gousios, G., & van Deursen, A. (2020). Questions for data scientists in software engineering: a replication. In P. Devanbu, M. Cohen & T. Zimmermann (Eds.), *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and*

Symposium on the Foundations of Software Engineering (pp. 568–579). ACM. https://doi.org/10.1145/3368089.3409717

Inmon, W. H. (1996). The data warehouse and data mining. *Communications of the ACM*, *39*(11), 49–50. https://doi.org/10.1145/240455.240470

Inmon, W. H. (2005). *Building the data warehouse* (4th edition). Wiley.

International Organization for Standardization. (2023). *ISO/IEC 25010:2023(en): - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Product quality model*. https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-2:v1:en

Ishwarappa & Anuradha, J. (2015). A Brief Introduction on Big Data 5Vs Characteristics and Hadoop Technology [PII: S1877050915006973]. *Procedia Computer Science*, *48*, 319–324. https://doi.org/10.1016/j.procs.2015.04.188

Jamal, A., Fleiner, R., & Kail, E. (2021). Performance Comparison between S3, HDFS and RDS storage technologies for real-time big-data applications. *SACI 2021*, 000491–000496. https://doi.org/10.1109/SACI51354.2021.9465594

Karna, H., Vicković, L., & Gotovac, S. (2019). Application of data mining methods for effort estimation of software projects. *Software: Practice and Experience*, *49*(2), 171–191. https://doi.org/10.1002/spe.2651

Kimball, R., & Ross, M. (2013). *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling* (3rd edition). Wiley.

Kreps, J. (2014). *Questioning the Lambda Architecture*. Retrieved March 12, 2025, from https://www.oreilly.com/radar/questioning-the-lambda-architecture/

Lewis, N., Bez, J. L., & Byna, S. (2025). I/O in Machine Learning Applications on HPC Systems: A 360-degree Survey [ACM Computing Surveys (CSUR), Vol. 1, No. 1, Article 1, January 2025]. *ACM Computing Surveys*, *18*, 11. https://doi.org/10.1145/3722215

Linstedt, D. (2015). *Building a Scalable Data Warehouse with Data Vault 2.0: Implementation Guide for Microsoft SQL Server 2014* (1. Aufl.). Elsevier Reference Monographs. http://www.sciencedirect.com/science/book/9780128025109

Marz, N. (2011). *How to beat the CAP theorem*. Retrieved March 12, 2025, from http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html

Mesnier, M., Ganger, G. R., & Riedel, E. (2003). Storage area networking - Object-based storage. *IEEE Communications Magazine*, *41*(8), 84–90. https://doi.org/10.1109/MCOM.2003.1222722

Microsoft Azure. (2025). *Azure Blob Storage*. Retrieved March 16, 2025, from https://azure.microsoft.com/en-us/products/storage/blobs

Microsoft Learn. (2024). *What is Azure Synapse Analytics?* Retrieved April 19, 2025, from https://learn.microsoft.com/en-us/azure/synapse-analytics/overview-what-is

Miloslavskaya, N., & Tolstoy, A. (2016). Big Data, Fast Data and Data Lake Concepts [PII: S1877050916316957]. *Procedia Computer Science*, *88*, 300–305. https://doi.org/10.1016/j.procs.2016.07.439

Negash, S., & Gray, P. (2008). Business Intelligence. In F. Burstein & C. Holsapple (Eds.), *Handbook on Decision Support Systems 2: Variations* (1. Aufl., pp. 175–193). Springer-Verlag. https://doi.org/10.1007/978-3-540-48716-6_9

Nogueira, I. D., Romdhane, M., & Darmont, J. (2018). Modeling Data Lake Metadata with a Data Vault. In B. C. Desai, S. Flesca, E. Zumpano, E. Masciari & L. Caroprese (Eds.), *Proceedings of the 22nd International Database Engineering & Applications Symposium on - IDEAS 2018* (pp. 253–261). ACM Press. https://doi.org/10.1145/3216122.3216130

Nygard, M. (2011). *Documenting Architecture Decisions*. Retrieved February 8, 2025, from https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions

Paras, J., Kraft, P., Power, C., Das, T., Stoica, I., & Zaharia, M. (Eds.). (2023). *Analyzing and Comparing Lakehouse Storage Systems*.

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Perens, B., et al. (1999). The open source definition. *Open sources: voices from the open source revolution*, *1*, 171–188.

Powers, M., & Zhu, R. (2023). *Adding and Deleting Partitions in Delta Lake tables*. Retrieved April 27, 2025, from https://delta.io/blog/2023-01-18-add-remove-partition-delta-lake/

Puonti, M., & Raitalaakso, T. (2019). Data Vault Mappings to Dimensional Model Using Schema Matching. In P. Doucek, J. Basl, A. M. Tjoa, M. Raffai, A. Pavlíček & K. Detter (Eds.), *Research and Practical Issues of Enterprise Information Systems: 13th IFIP WG 8.9 International Conference, CONFENIS 2019, Prague, Czech Republic, December 16–17, 2019, Proceedings* (1st ed. 2019, pp. 55–64, Vol. 375). Springer International Publishing; Imprint Springer. https://doi.org/10.1007/978-3-030-37632-1_5

R. P. L. Buse & T. Zimmermann. (2012). Information needs for software development analytics. *2012 34th International Conference on Software Engineering (ICSE)*, 987–996. https://doi.org/10.1109/ICSE.2012.6227122

Rehman, K. U. u., Ahmad, U., & Mahmood, S. (2018). A Comparative Analysis of Traditional and Cloud Data Warehouse. *VAWKUM Transactions on Computer Sciences*, *15*(1), 34. https://doi.org/10.21015/vtcs.v15i1.487

Rique, T., Perkusich, M., Dantas, E., Albuquerque, D., Gorgônio, K., Almeida, H., & Perkusich, A. (2023). On Adopting Software Analytics for Mana-

gerial Decision-Making: A Practitioner's Perspective. *IEEE Access*, *11*, 73145–73163. https://doi.org/10.1109/ACCESS.2023.3294823

Scheibel, W., Blum, J., Lauterbach, F., Atzberger, D., & Döllner, J. (2024). Integrated Visual Software Analytics on the GitHub Platform [PII: computers13020033]. *Computers*, *13*(2), 33. https://doi.org/10.3390/computers13020033

Serra, J. (2021). Data Lakehouse defined. *James Serra's Blog*, *2021*. Retrieved April 19, 2025, from https://www.jamesserra.com/archive/2021/01/data-lakehouse-defined/

Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The Hadoop Distributed File System. In M. G. Khatib (Ed.), *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST): 6-7 May 2010, Lake Tahoe, Nevada, USA* (pp. 1–10). IEEE. https://doi.org/10.1109/MSST.2010.5496972

The Apache Software Foundation. (2022). *HDFS Architecture Guide*. Retrieved March 14, 2025, from https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

The Apache Software Foundation. (2024a). *Apache Hudi*. Retrieved December 31, 2024, from https://hudi.apache.org/

The Apache Software Foundation. (2024b). *Apache Iceberg*. Retrieved December 31, 2024, from https://iceberg.apache.org/

The Apache Software Foundation. (2024c). *Apache ORC*. Retrieved December 31, 2024, from https://orc.apache.org/

The Apache Software Foundation. (2024d). *Apache Parquet*. Retrieved December 16, 2024, from https://parquet.apache.org/

The Apache Software Foundation. (2025a). *Apache Hadoop*. Retrieved March 14, 2025, from https://hadoop.apache.org/

The Apache Software Foundation. (2025b). *PySpark Overview — PySpark 3.5.5 documentation*. Retrieved April 28, 2025, from https://spark.apache.org/docs/latest/api/python/index.html

The Apache Software Foundation. (2025c). *What is Airflow?* Retrieved April 19, 2025, from https://airflow.apache.org/docs/apache-airflow/stable/index.html

The Linux Foundation. (2025). *Best practices — Delta Lake Documentation*. Retrieved April 27, 2025, from https://docs.delta.io/latest/best-practices.html

Union.ai. (2025). *Core concepts - Flyte Docs*. Retrieved April 20, 2025, from https://www.union.ai/docs/flyte/user-guide/core-concepts/

Wang, S., Huang, L., Gao, A., Ge, J., Zhang, T., Feng, H., Satyarth, I., Li, M., Zhang, H., & Ng, V. (2023). Machine/Deep Learning for Software Engineering: A Systematic Literature Review. *IEEE Transactions on Software Engineering*, *49*(3), 1188–1231. https://doi.org/10.1109/TSE.2022.3173346

Weintraub, G. (Ed.). (2023). *Optimizing Cloud Data Lake Queries*.

Zagan, E., & Danubianu, M. (2021). Cloud DATA LAKE: The new trend of data storage. *HORA 2021*, 1–4. https://doi.org/10.1109/HORA52670.2021.9461293

Zenko. (2025). *Cloudserver*. Retrieved April 27, 2025, from https://www.zenko.io/cloudserver/