Master Thesis

Building a Low-Code Platform for versatile Data Integration

Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science

to the Department of MNI
Technische Hochschule Mittelhessen
(University of Applied Science)

by

Timon Pellekoorne

August 12, 2024

First examiner: Prof. Dr. Frank Kammer

Second examiner: Prof. Dr. Harald Ritz

Declaration of Independence

I hereby declare that I have composed the present work independently and have not used any sources or aids other than those cited, and that all quotations have been clearly indicated. The thesis has not been submitted to any other examination authority in the same or a similar form and has not been published. In addition, I agree that my thesis will be subjected to the THM internal plagiarism check.

Gießen, on August 12, 2024 Timon Pellekoorne

With the increasing amount of data and the growing number of software products in use, companies are faced with the major challenge of processing and merging the versatile data from the various systems. In view of the high costs of data warehouse solutions, especially in the case of complex challenges in companies, this thesis proposes a concept for a data integration platform that enables companies to merge and prepare data from current and possible future systems in a simple, time-saving and cost-effective manner. It is based on a mixture of microservices architecture and event-driven architecture, with services consisting partly of open source software solutions and partly of software solutions developed by the company itself. The concept was developed as part of the work at a local energy supplier, with the goal of creating a cross-domain platform for data integration. Parts of the implementation of the concept are also described in this thesis. The final evaluation shows that the concept developed meets the challenging requirements of the company, while also highlighting limitations that should be investigated in future work.

Mit der zunehmenden Datenmenge und der wachsenden Anzahl an eingesetzten Softwareprodukten stehen Unternehmen vor der großen Herausforderung, die vielfältigen Daten aus den unterschiedlichen Systemen aufzubereiten und zusammenzuführen. Angesichts der hohen Kosten von Data-Warehouse-Lösungen, insbesondere bei komplexen Herausforderungen in Unternehmen, wird in dieser Arbeit ein Konzept für eine Datenintegrationsplattform vorgeschlagen, die es Unternehmen ermöglicht, Daten aus aktuellen, aber auch aus möglichen zukünftigen Systemen einfach, zeitsparend und kostengünstig zusammenzuführen und aufzubereiten. Dabei wird auf eine Mischung aus Microservices-Architektur und ereignisgesteuerter Architektur gesetzt, deren Services zum Teil aus Open-Source-Softwarelösungen und zum Teil aus eigenentwickelten Softwarelösungen bestehen. Das Konzept wurde im Rahmen der Arbeit bei einem lokalen Energieversorger entwickelt, wobei das Ziel einer domänenübergreifenden Plattform zur Datenintegration verfolgt wurde. Teile der Umsetzung des Konzepts werden ebenfalls in dieser Arbeit beschrieben. Die abschließende Evaluierung zeigt, dass das entwickelte Konzept den anspruchsvollen Anforderungen des Unternehmens gerecht wird, wobei auch Grenzen aufgezeigt werden, die in zukünfitigen Arbeiten untersucht werden sollten.

Contents

List of Figures iii

List of Tables v

Listings vii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Approach . . . 2
- 1.4 Limitation . . . 3

2 Fundamentals 5
- 2.1 Data Warehouse . . . 5
- 2.2 ETL Process . . . 6
  - 2.2.1 Extraction . . . 6
  - 2.2.2 Transform . . . 7
  - 2.2.3 Load . . . 8
  - 2.2.4 ETL-Tools . . . 8
- 2.3 Open Source Software . . . 8
- 2.4 Application Programming Interfaces . . . 9
- 2.5 Container . . . 9
- 2.6 Low-Code Platform . . . 10
- 2.7 SQL . . . 11

3 Related Work 13

4 Requirements 17
- 4.1 User Requirements . . . 18
- 4.2 System Requirements . . . 19
- 4.3 Business Requirements . . . 20

5 Concept 21
- 5.1 Architecture . . . 21

5.2 Services . . . 23
5.2.1 Data Integration . . . 24
5.2.2 Core Database . . . 28
5.2.3 Plausibility . . . 30
5.2.4 Semantic Layer . . . 33
5.2.5 Model Synchronizer . . . 35
5.2.6 API-Layer . . . 36
5.3 Graphical User Interface . . . 38
5.3.1 Data integration . . . 38
5.3.2 Data modeling . . . 41
5.3.3 Plausibility check . . . 46
5.4 Summary . . . 48

6 Implementation 49
6.1 Prerequisites . . . 49
6.2 Data Integration . . . 49
6.2.1 Software Selection . . . 50
6.2.2 Airbyte . . . 50
6.2.3 Integration . . . 52
6.3 Core Database . . . 52
6.3.1 Software Selection . . . 52
6.4 Semantic Layer . . . 54
6.4.1 Software Selection . . . 54
6.4.2 Dynamic Modeling . . . 54
6.5 Model Synchronizer . . . 60
6.5.1 Database Changes . . . 60
6.5.2 Realization . . . 61
6.6 Graphical User Interface . . . 61
6.6.1 Data Integration . . . 62
6.6.2 Data Modeling . . . 65

7 Evaluation 69
7.1 Develop Scenarios . . . 69
7.2 Perform Scenario Evaluations . . . 71
7.3 Reveal Scenario Interactions . . . 71
7.4 Overall Evaluation . . . 72

8 Conclusion 75
8.1 Discussion . . . 75
8.2 Limitations and Future Research . . . 76

Bibliography 77

List of Figures

2.1 Basic Architecture of a Data Warehouse [Sch16] 6
2.2 Container Structure 10

5.1 Event-Driven Architecture 22
5.2 Overview of the Concept and its Microservices 24
5.3 Data Integration Approaches [Mas24] 26
5.4 Design of the ETL-Service 26
5.5 Temporal Tables in SQL Server 2016 [McD24] 29
5.6 Design of the Core Database 29
5.7 Basic CI/CD Architecture [Las23, Git24c] 31
5.8 Design of the Plausibility Service 32
5.9 Design of the Semantic Layer 35
5.10 Design of the Model Synchronizer 36
5.11 Design of the API Gateway 37
5.12 Wireframe of the Source System Type Selection 39
5.13 Wireframe of the Connection Configuration Step 39
5.14 Wireframe of the Connection Testing Step 40
5.15 Wireframe of the Data Selection Step 40
5.16 Wireframe of the Transformation Step 41
5.17 Four Types of Joins 42
5.18 Wireframe for Joining Imported Tables 43
5.19 Wireframe for Adding new Measures 44
5.20 Wireframe for the Formula Editor 45
5.21 Wireframe for Creating new Data Views 45
5.22 Wireframe for Choosing the Runner 46
5.23 Wireframe for Setting Runner Configuration 47
5.24 Wireframe for Choosing Data 47
5.25 Wireframe for Setting Destination Data Model 48

6.1 Architecture Overview of Airbyte [Air24a] 51
6.2 Database Schema for Storing Dynamic Data Models 59

List of Tables

4.1 Excerpt from the Functional Requirements of the Specifications Sheet . 17

5.1 Architecture Characteristics Rating [Ric20] 21

6.1 Overview of React Flowchart Libraries (Accessed: 04-08-2024) 65

7.1 Scenario Evaluations 71
7.2 Scenario Interactions per Component 71

Listings

2.1 Example of a Uniform Resource Identifier 9
2.2 SQL SELECT Statement 11

5.1 Simple SQL Join Statement 42
5.2 Measures in SQL 44
5.3 Creating a View in SQL 45

6.1 Creating a Cube 55
6.2 Adding Columns 55
6.3 Adding Measures 56
6.4 Adding Joins 56
6.5 Adding Cubes to View 57
6.6 Adding Dimensions to View 57
6.7 Load Data for Dynamic Cube Creation 58
6.8 Render YAML File via Jinja Template Engine 58
6.9 Implement Schema Version for Cube 59
6.10 Consuming Messages from the Postgres Channel 61
6.11 Execution of the Data Model Transformation 61
6.12 Example of Rendering an Input Field 63
6.13 Example for the Use of Generic Types 63
6.14 Interface for Creating New Sources 64
6.15 Creating a Custom Node 66
6.16 Add Connectors to Custom Node 66
6.17 Create Node Objects of Cubes 67
6.18 Create New Join via Flow Chart 67

.

1 Introduction

1.1 Motivation

The technological progress of recent years is leading to an ever-increasing global volume of data [Hel09]. In 2021, a statistic published by the International Data Corporation (IDC) estimated the volume of data created, captured, copied, and consumed worldwide for the year 2025 at 181 zettabytes [Idc21]. This is 15 times the volume in 2015. Due to this constantly growing volume of data and the associated dependency on this data, the availability and analysis of data is becoming increasingly important [Pau21]. In addition to the increasing amount of data, the number of software applications in companies is also growing. In 2022, companies worldwide use an average of 130 software applications provided by various service providers in the cloud, compared to an average of eight in 2015 [Bet22]. This development presents companies with the challenge of combining data from a large number of applications for optimal analysis. As a result, the demand for business intelligence (BI) products for collecting and processing data is also growing [Hel09], as data integration is an important factor in data analysis [Hlu21].

Even companies in the energy sector are confronted with the challenges of this trend, especially in times of constantly growing energy costs [Des24]. With the rise of new technologies such as remotely transmitted meters, temperature meters, CO2 meters and many more, new software applications are also being used in these companies. The analysis of the data generated in the various systems can be used to increase energy efficiency and thus offer long-term economic benefits by lowering the costs of fuel imports/supply, energy production and reducing emissions from the energy sector [HM20]. This can also offer customers advantages that make a company more competitive. Due to the high demand for BI products, there are numerous solutions for integrating and analyzing heterogeneous data, but these so-called *Data Warehouses* are very cost-intensive, especially when it comes to the challenging requirements of a company, where additional development costs are required by the provider. Such specific requirements arise mainly in the area of data cleansing, plausibility checks and modeling of the various data and cannot be covered by the standard functions that the manufacturers supply in these areas.

In order to counteract these disadvantages of conventional data warehouse systems and to overcome the challenges described, this thesis designs and implements a system that enables companies to combine data from a wide variety of systems via a graphical user interface, to cleanse and plausibilize it and finally to model it.

1.2 Objectives

Although the thesis was created in the environment of an energy supply company, such a system can also be used in other areas. In the medical field, for example, there is also a need to combine data from various hospital information systems such as *MR*, *PACS*, *LIS* and so on in one system [Lyu15]. Also, the implementation of development concepts of smart cities require data integration platforms for storing and analyzing the emerging data [Har19] and even in such special fields as mass spectrometry, one of the major challenge is how to handle, integrate, and model the data that is produced [Pas14].

The objective of this thesis is therefore to design a system that offers companies a cost-effective alternative to proprietary software solutions and at the same time meets the challenging requirements of different companies. The system should offer a good compromise between a complete, self-developed individual solution, which implies a high development effort, and a standard software that cannot fulfill all challenging requirements. The key component should be an easy-to-use user interface that allows the entire process, from the definition of external source systems to data cleansing and modeling, to be managed by one user via this interface. These objectives give rise to the research question on which this thesis is based:

How to build an individual easy-to-use platform for versatile data-integration for a company with challenging requirements?

1.3 Approach

The basic architecture of the system is that of a conventional data warehouse. In order to offer a cost-effective alternative that does not have the disadvantages of self-developed individual software and that there are already existing free software solutions in the area of data integration that cover most of the required functions, the basic approach of the system is to combine existing open source software components with the development of a customized application. This approach combines the advantages of standard software through the open source products and the advantages of individual software through the extension of these and the development of an own application for the individual

configuration and administration of the data according to the requirements of the company.

To ensure that a concept can be developed and implemented according to this approach, a detailed overview of related work is first provided (Chapter 3). Based on the knowledge gained from this and the specific requirements of the energy supplier, requirements are then derived that the concept to be created must fulfill (Chapter 4). Based on these requirements, the concept of the system is then presented (Chapter 5), the implementation of which is described in part in Chapter 6. To ensure that the system meets the requirements, it is evaluated on the basis of specific scenarios (Chapter 7). Finally, Chapter 8 discusses the developed concept as well as the limitations and future research.

1.4 Limitation

In order for companies to derive the described benefits from the integration of data, an analysis platform is required via which users can analyze the merged and processed data. Such a platform must give the user the option of displaying and filtering data using a wide variety of visualizations. The design of such a platform is not part of this thesis. However, there are some analysis platforms such as *Superset*, *KNIME* etc. that can be used as an analysis platform for the system developed in the thesis. Stephan [Ste24] describes a concept and implementation of an analysis platform based on open source software in his thesis, which was used as the analysis platform for the system described in the thesis.

2 Fundamentals

2.1 Data Warehouse

The term *Data Warehouse* first appeared in the 1990s. This term was coined by Inmon, who defined a data warehouse in his book as a subject-oriented, integrated, non-volatile, and time variant collection of data in support of management decisions [Inm96]. According to these described properties, a data warehouse is [Bau13]:

1. A system that was not only created to fulfill a specific task, but can evaluate entire subject areas across domains (subject-oriented).
2. A system consisting of a database made up of integrated data from different data sources (integrated).
3. A system that is based on a stable database whose integrated data is not changed or removed (non-volatile).
4. A system in which the data is stored over a long period of time in order to enable time-related analyses (time variant).

Data warehouses are therefore used in companies in which a large number of systems are used in the individual departments and specialist areas that must not be affected by data analysis or processing. This is where the redundant data storage of data warehouses becomes useful. In contrast to conventional information systems, a special feature of data warehouses is that the data is no longer modified or removed after integration. Only new data is added to the system without overwriting old data. [Kö14, Bau13]

The architecture of a data warehouse can be traced back to a basic architecture (see Figure 2.1) consisting of the staging area, cleansing area, core and marts layers. [Sch16]

Staging Area. The data received from external sources is stored unchanged in the staging area. The structure of the data in the staging area therefore always corresponds to that of the source system.

Cleansing Area. The cleansing area is there to clean up incorrect data from the staging areas and to transform the data from external sources into a standardized form.

Core. The core is the central point in which the data from the staging and cleansing area is merged.

Marts. Data marts form specific views of the data in the core, i.e. they are a subset of the data. These views can reduce the complexity of data queries.

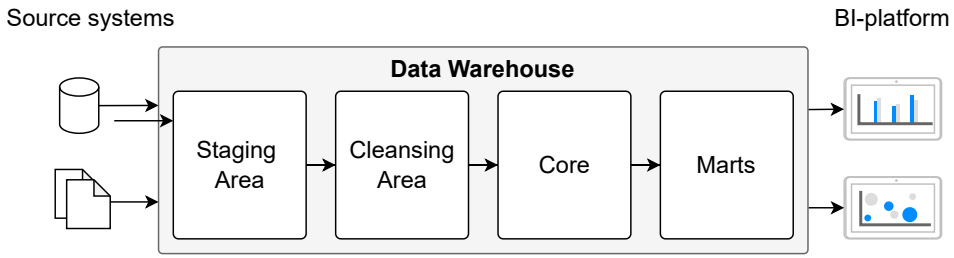

Figure 2.1: Basic Architecture of a Data Warehouse [Sch16]

2.2 ETL Process

The process of loading data from external sources at certain intervals (periodically or manually by a user) into the staging area, then transforming this data into a standardized format and storing it permanently in the target database for later analysis is described as an ETL process, where ETL stands for *extract*, *transform* and *load*. [Kö14] While the staging, cleansing area and the core are located in the data warehouse, the extraction of the data usually takes place outside the data warehouse. [Sch16]

2.2.1 Extraction

During extraction, the data is taken from the various source systems. There are various approaches for selecting the extraction period. [Bau13]

Periodically. The data is read out at regular intervals. The interval length can be selected according to the requirements of the source systems.

Event-driven. The data is read out when a specific event happens. This can be a certain number of changes or even exceeding a certain value.

Request-driven. The data is read out as required, for example by a user for new analysis.

Immediately. If data needs to be up-to-date, it can be imported immediately so that the data in the data warehouse is as up-to-date as in the external source systems.

In addition to the choice of extraction period, the choice of extraction mechanism is also important in this phase of the ETL process. A distinction is made between the two mechanisms of *full extraction* and *delta extraction*. [Sch16]

Full Extraction. All data is extracted from the external source systems.

Delta Extraction. Only a subset of the data is extracted from the external source systems. The subset is limited to a specific time period, e.g. the period between the last extraction and the current one.

2.2.2 Transform

Transforming the data is the most complex part of the ETL process and includes the tasks of *integration*, *data cleansing*, *versioning* and *aggregation*. [Sch16]

Integration involves converting the data from the source systems into a standardized form. A distinction can be made between three different transformations. [Ros13]

Syntactic transformations. The syntactic transformation refers to a standardization of the extracted data with regard to data types, data formats, etc.

Structural transformations. The structural transformation refers to different structures of the data and their modeling.

Semantic transformations. Semantic transformations involve the standardization of designations, scales used or similar.

The *data cleansing* process is responsible for ensuring data quality by detecting and correcting unwanted, inconsistent and incorrect data. [Alo17] The correction of incorrect data can only be automated in a few cases, e.g. when correcting character strings such as an address by means of a similarity-based comparison with an address register. In other cases, the system can either randomly compare values with values from the real world and replace them in the event of a discrepancy or use 'empirical values' from a database to provide an indication of possible incorrect data. [Bau13] The consistency check can be rule-based, e.g. by using regular expressions. However, this requires domain-specific knowledge. [Bau13] In addition, irrelevant records and duplicates can be filtered out, or default values can be inserted to increase completeness. [Sch16]

If data records are changed, *versioning* ensures that a new version is created for the new data records and the previous version is marked as completed [Sch16].

Aggregation involves summarizing the data from the core at a higher level in order to make this data usable for specific analyses in the data marts. When summarizing, additive key figures are added up, whereas a suitable aggregation function must be defined for non-additive key figures, such as the average for percentage values. [Sch16]

2.2.3 Load

Once the data has been extracted from the source systems and then transformed, it can be loaded into the target system. Since a large amount of data has to be loaded — especially when initially loading a data source — bulk loaders are used, which utilize a method that can enter a large amount of data into a database system. [Ros13, Bau13]

2.2.4 ETL-Tools

For the development of ETL processes, so-called ETL tools are often used. These combine all the required steps of the ETL process in a separate system [Ros13] and therefore allow it to be implemented quickly and easily via a graphical interface [Sch16]. ETL tools offer so-called *connectors* for extracting data from a source system and loading it into a target system. A connector is responsible for the software of a specific source or target system. For example, most ETL tools offer a MySQL connector for connecting MySQL databases. ETL tools represent a separate system and are not directly integrated into the data warehouse, which allows the tools to operate independently of the source and target systems, reducing the load on these systems. Such ETL tools are also available as *open source software* and are therefore freely available.

2.3 Open Source Software

The main characteristic of *open source software* is that the source code is freely available [BSI24]. But not only that, there are ten points from the *Open Source Initiative* that define whether software is open source. They require, in addition to public source code, that the software be freely redistributable, freely modifiable, and licensed under an open source license. [Ope24] This means that open source software can be used and developed as an entire application, as part of an application, or as the basis for an application. For example, open source software can be used as the server of an application that is then called via an API by a custom-developed client.

2.4 Application Programming Interfaces

Application programming interfaces (API) are documented interfaces that enable an application to use the services and functions of other applications, operating systems or other software programs. Using these APIs, software developers can create applications by using the APIs of different software libraries and combining them into one application. [JRC19] The most commonly used architectural style for implementing APIs is the REST architectural style [Pos23]. REST was developed by Fielding [Fie00] in 2000 as part of his dissertation and stands for *Representational State Transfer*. This architectural style defines the six design rules with the REST constraints, whereby it must be a client-server architecture that is stateless and cacheable, has a uniform interface, uses a layered system and sends program code on demand [Sub19]. The core of REST consists of uniquely identifiable resources that can be queried, created, edited or deleted using the *HTTP* methods *GET*, *POST*, *PUT* and *DELETE*. A resource is something that can be described uniquely, such as a customer or a file. These resources are uniquely identified via *Uniform Resource Identifier (URI)*, which consists of a schema, which is in REST HTTP or HTTPS, an authority, a path and optionally also query parameters [Wir19]. The query of a customer with ID 1 with the authority *company.com* is shown as an example in Listing 2.1.

Listing 2.1: Example of a Uniform Resource Identifier

```
http://company.com/customers/1
```

2.5 Container

Containers are a form of virtualization at operating system level in which the resources of a single operating system are divided into isolated groups, so-called containers. The technology appeared in 1979 with the introduction of the *chroot* command in *UNIX* operating systems. This command was intended to enable the isolation of file system resources for each process, which is the basis of OS-level virtualization. Building on this command, more and more technologies came onto the market until the best-known software, *Docker*, finally appeared in 2013. Docker is not just a container runtime to execute containers, but an entire ecosystem for containers with tools and standards for creating and managing containers. Even though Docker is currently still the leading container technology [Dat23], more and more providers have appeared over time. For this reason, Google, Docker, IBM, Microsoft and others founded the Open Container Initiative (OCI) in 2015, which published the open container standard in 2016. This includes the *Container running standard* and the *Container image standard*. [Hua23]

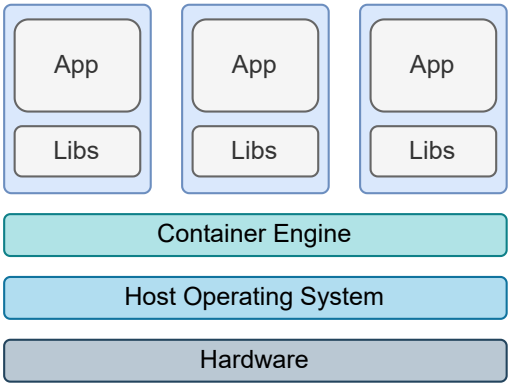

Figure 2.2: Container Structure

A container consists of the application with all its required dependencies and can therefore be executed in isolation from other processes in the user space of the host operating system. As can be seen in Figure 2.2, all containers use the same host operating system. This offers the advantage that containers are therefore smaller as they do not require their own operating system, they are available more quickly and are cheaper and easier to migrate. [Hua23]

The basis of every container is the container image. This provides the basis of the container by means of a read-only file system, which cannot be changed. For the management of files in a container, it also has an overlay file system, which is mapped on the host system within the Docker directory. This separation of the file systems makes it possible to derive any number of containers from the container images. [Kof21] Because a container is isolated, it can be run on virtual machines, physical servers or in the cloud without any adaptations to the host operating system. [Hua23]

2.6 Low-Code Platform

The tendency of developers to switch from low-level languages to high-level languages during development has existed since programming languages have existed [Pin23]. The first concepts for developing applications without extensive programming knowledge therefore emerged as early as the 1980s [Mar82]. However, the term 'low-code' was first mentioned in 2014 by a market research company [Boc21].

A *low-code platform (LCP)* – often also called *low-code application platform (LCAP)* or *low-code development platform (LCDP)* – is a software development tool that enables users with little practical knowledge to efficiently create applications with complex business logic. They aim to close the gap between experienced software developers and people with little or no experience in software engineering. To achieve this goal,

low-code platforms offer the possibility to design the functionality of the application via graphical user interfaces, using mechanisms such as drag and drop. [Sá24] LCPs are particularly strong in the areas of database applications, mobile applications, process applications and request-handling applications [Ihi20]. Especially in the area of database applications, low-code platforms can help users to make queries to a database and model and edit data without having to learn more complex query languages such as *SQL*.

2.7 SQL

The term *SQL* stands for *Structured Query Language* and is the language for accessing relational databases. SQL was developed by IBM and standardized by ANSI (American National Standards Institute) in 1986. One year later, this standard was adopted by the ISO (International Organization for Standardization) [Mar93]. The language was initially intended to be used by an end user to access the data in a database. Today, however, graphical user interfaces are used for this purpose, with SQL being used more as a language for database programming [Sch17]. SQL statements are then written to access relational databases. These are divided into three classes [Mar93]:

Data-Definition-Language (DDL). Statements belonging to this class are used to create, modify or delete data structures.

Data-Control-Language (DCL). These statements can be used to assign access rights to the database. In this way, the access rights of several users can be managed so that some only have read-only rights to the database, for example.

Data-Manipulation-Language (DML). Statments from this class are used to query, change, add or delete data.

SQL is a descriptive language, which makes it easier for end users to formulate statements in SQL, as a statement is structured in such a way that it describes the desired result [Kau23]. In Listing 2.2, for example, a SELECT statement, which belongs to the Data Manipulation Language class and is for querying data, is used to select the surname and first name of a customer from the Customer table, where the customer number is 1.

Listing 2.2: SQL SELECT Statement

```
SELECT surname, firstname FROM customer WHERE customer_number = 1;
```

3 Related Work

The search for related literature was conducted on the databases AIS, IEEXplore, Sciencedirect, ACM and the online library of the Technische Hochschule Mittelhessen. The keywords "data integration platform", "etl" and "data warehouse" were used for the search strings. Only publications published after the year 2000 were taken into consideration. The following criteria were used to further limit the publications: (1) It is about the conception or implementation of a proprietary system. (2) It is a market analysis of current data integration systems. After applying these criteria, nine publications were considered relevant and examined more closely.

A great need for data integration platforms can be found in the biological and medical field. Four of the selected papers are assigned to this area and include the design and implementation of a data integration platform. Jayaratne et al. [Jay19] propose an open data integration platform for patient, clinical, medical and historical data available across multiple health information systems. This should enable the centralization of data sets as well as the possibility of connecting analysis and reporting solutions. The data integration platform is implemented as a web application based on a three-tier architecture. The first tier is the user interface, the second tier is the business logic of the application and the third is the data access layer, which also contains the data integration module. This module supports both regular and on-demand synchronization of external data. The database schema is fixed here, and it is not intended to allow users to modify their own data. The system is also limited in terms of modularity and expansion of new external source systems, as the connectors to the clinical data have all been implemented specifically for the systems used in the monolith of the three-tier architecture and therefore simple expansion of connectors by e.g. a *software development kit (SDK)* or similar is not intended. Winters-Miner et al. [WM15] describe in their work the development of an IT infrastructure designed to intended to provide validated clinical data for researchers and managers in the field of medicine. For the implementation of data integration, they choose two components in their architecture, the Medical Data Warehouse, a central database with all data stored in either relational or multidimensional data structures, and a Data Collection Module, which is responsible for the collection from external source systems. In addition to the components, five other components are implemented for analyzing the data, but these are not relevant to the topic of this thesis. The exact architecture of the individual components is not described

further in this paper, which is why the extensibility of these cannot be estimated. In addition, this work focuses on the integration of special clinical information systems, which is why the integration of various types of data is limited. Modeling of the integrated data by a user is also not provided for in this system. Lyu et al. [Lyu15] present the design and implementation of another clinical data integration and management system. This system is based on a hadoop platform, which can connect to multiple heterogeneous source systems. For the integration of the data, they use a module that is implemented with the Apache Camel framework and receives the data using the HL7 standard, which is a group of international standards for the exchange of healthcare data. This module sends these messages to the database – which is the Hadoop distributed file system – via the predefined workflows, which contain routing rules and transformations. In addition to the module for data integration, this system also offers a module for data management, whereby this module only offers high-performance data queries and no user-related data modeling. Like the previous system, this one is also specially designed for the clinical context, which is made particularly clear by the use of the HL7 standards. The last paper that comes into question as related literature in the field of biology or medicine is a computational platform by Pasculescu et al. [Pas14]. They present the platform *CoreFlow*, which was developed to handle, integrate and also model data. On the presented platform it is possible to upload data into a relational database and to process, correct or model it using user-defined scripts. The platform was primarily designed for programmers to enable them to perform fast data analysis. The execution of the scripts is implemented by using pipelines that use a hierarchical organization of owners, projects, topics and tasks.

Another field in which two of the selected papers fall is that of urban infrastructure. Harris and Sartipi [Har19] propose a data integration architecture further urban development initiatives through this integration platform. This integration platform acts as a central system that receives data from all source systems. The system is based on an event-driven architecture in which the core of the system is the so-called data hub — a cluster of several brokers. Other applications can then act as consumers or producers and either integrate data or consume it for analysis. The proposed architecture is therefore only part of an overall data integration platform, as it is not possible to integrate data automatically without the need to develop your own producers. In addition, it does not offer the possibility of data modeling. The second paper in this topic comes from Chen et al. [Che22] They proposed a plan for building a cloud-based big data platform that follows the innovative development of urban rail. The architecture of the cloud-based big data platform consists of five layers and follows the everything-as-a-service approach from cloud computing. At the bottom is the infrastructure-as-a-service layer, which provides comprehensive virtualization services such as data processing, storage, network and security. Above this is the platform-as-a-service layer, which contains general application components such as a database, middleware, etc. On top of this layer is

the data-as-a-service layer, which provides data services such as data cleaning, data standardization, data integration, etc. The aim of these services is to ensure that every application can query data without having to consider the source from which it comes. The data is standardized for this purpose. This is done by mapping the IDs of the business systems to the master data IDs using a table. In addition to standardization, the data objects of various system-related elements such as vehicles, signals, etc. are combined, refined and modelled. This is already done with regard to the domain and restricts the use of the platform with regard to use in another domain. The fourth layer is the software-as-a-service, which contains various intelligent application services for operation. The last layer is the display layer which shows data and analysis results. The big data platform provides a common standard interface so that data consumers can easily query the data. This interface can be extended via a low-code interface so that new service interfaces can be generated when new requirements arise. In this way, the system offers users the opportunity to create their own views of the data and thus to model it in a way.

Another system for integrating data from multiple data sources is presented by Sarnovsky et al. [Sar17], who are aiming to *'design and develop the cross-sectorial scalable environment, which will enable the data collection from different sources and support the development of predictive functions to help the process industries in optimizing of their production processes'*. The system is based on Apache Hadoop and uses Apache NiFi for the implementation of data integration, which supports a variety of modules for transformation and integration, but can also be extended with custom modules. This enables great flexibility in the integration of different data types from external data sources. In addition, Apache NiFi offers the possibility to define the workflows for the integration via a graphical user interface.

As the last paper to be considered, Nie et al. [Nie21] present a design of a big data integration platform based on a hybrid hierarchy architecture. The architecture is based on a classic data warehouse architecture, in which the data from the various data sources is loaded into a data warehouse. However, to increase query performance, they divide the data sources into different topics according to the business logic and then add intermediate data sources with this data. Thus, when extending an external data source, only the topic needs to be extended, which significantly increases the scalability and performance of the system. This approach severely limits the use of the system in different domains, as the subject areas must be defined during implementation.

As the research has shown, there are many approaches for the development of a data integration platform. Almost all of them show a major limitation with regard to easy data modeling by a user. Only Chen et al. offer a low-code approach by creating new interfaces via an API. The search in the literature for systems that offer this possibility proved to be difficult. No paper could be identified that describes the design

or development of a system that allows data to be modeled using a low-code platform. Only Hoseini et al. [Hos24] mention the semantic layer of *AtScale* in their work, in which they provide an overview of semantic data management and semantic modeling. AtScale is a product of the company of the same name that takes the approach of providing data integration through data modeling in a canvas in which users can create relationships and hierarchies between heterogeneous data sources from different data stores.

4 Requirements

To be able to answer the research question raised in Section 1.2, a system is necessary that fulfills the requirements of a data integration platform as well as the special requirements of a company. In order for the system to achieve this, requirements are defined in this chapter on the basis of which the system is then conceptualized.

The requirements of a data integration platform can be derived from the requirements of data warehouses and their ETL processes, whereas a different methodology is needed to meet the company-specific requirements. Since, as already mentioned, the development of the concept arose from the need of an energy supplier for such a system and accordingly a specification sheet was created at the beginning, the requirements were derived from the general requirements for data integration platforms and the functional requirements listed in the specification sheet (see table 4.1 for an excerpt).

No.	**Functional requirements**
1	It is possible to create, remove and configure connections to source systems via an interface in order to automatically query data from them.
2	The raw imported data is available unchanged for error detection.
3	It is possible to transform or plausibilize data when loading into the target system.
4	Data can be entered manually into the Core Database or existing data can be adapted via an interface.
5	The imported data can be modeled by the user into a business view.

Table 4.1: Excerpt from the Functional Requirements of the Specifications Sheet

In order to counteract the fact that there was only one steakholder's specification sheet, the use cases from other domains that emerged in the literature research were also taken into account when selecting the requirements listed there. In this way, the requirements offer a range that finally makes the system usable for many domains.

4.1 User Requirements

Combining different data sources (U1). Combining data from different sources is a key component of a data warehouse [Sch16]. As described in Chapter 2, the ETL process is responsible for this. Based on this, a user should be able to use the system to define external source systems of all kinds, such as a MySQL database, an Excel file, etc., as an interface in order to load the data from these sources into the system. The user should be able to precisely define the required data from the source.

Manual and automatic data import (U2). Both from the extraction phase of the ETL process and from the requirements of the specification sheet, it can be deduced that the data from the various data sources should be imported into the system manually or automatically at specific intervals configured by the user [Bau13]. The user should be able to define these times for the various data sources and decide how the data should be imported, whether all data from the source system should be imported and the existing data overwritten or whether only new data should be appended to the data already imported.

Source independent data storage (U3). It should be possible to store all possible data, regardless of its source and whether structured or unstructured, uniformly in the system's central database [Bau13]. Such a transformation can be useful both for the actual values, e.g. to parse data types, but also for the structure of the data, e.g. the name of a column, because this has hardly any meaning in the source system.

Plausibility check of data (U4). In addition to the transformation of the imported data, it is also necessary to check the plausibility of the data for subsequent analysis. On the one hand, this is a specific requirement of the energy supplier with regard to incorrect meter data; on the other hand, plausibility checks are an important part of data quality, which in turn is an essential task of data integration platforms [Alo17]. With regard to the various stakeholders interested in such software the system should make it possible to execute all possible algorithms for the plausibility check of data during an import on the imported data.

Overwrite data manually (U5). In addition to the modification of data by plausibility algorithms, it should also be possible to manually adjust or create certain entries by a user so that their completeness can be guaranteed for analyses. This requirement arises directly from the specification sheet, but represents a useful function overall and is therefore also generally interesting.

All original values are available (U6). All data loaded into the system from external sources should be available in its original form and accessible to users. This means that even if plausibility checks or manually overwritten values are possible, it should still

be an option to view the data taken from the external sources for further analysis of possible faulty data. This is important for the required error detection and fulfills the task of versioning in the data warehouse.

User-defined modeling of imported data (U7). Translating the technical view into the business view is an important part of data warehousing [Sch16]. Therefore, the user should be able to create a user-defined view of all imported data from different source systems. To do this, they must be able to join different imported tables, dynamically define primary keys, and dynamically create additional attributes and additional views.

Easy to use via a low-code platform (U8). As described in Section 2.6, LCPs are intended to bridge the gap between experienced software developers and people with little or no experience in software engineering. This makes it possible for many more employees to use the system's functions. In addition, the research question asks for an easy-to-use platform. Therefore, the system should provide the all the functions required by the user requirements via a graphical user interface. This should make it as easy as possible to perform all functions, even without experience in software development or a deeper understanding of IT. These requirements are therefore also the basis for the other user requirements, as these are always to be executed via a graphical user interface.

4.2 System Requirements

In addition to the requirements that a user has of the system, there are also two system requirements that are essential for the conception and implementation of the required system.

Secure and efficient data storage (S1). The data stored in a company is sensitive, which can also be derived from the requirements of the specification sheet. Therefore, all data imported from source systems should be stored securely. It should be possible to secure access to this data as well as the import process. Another requirement that arises from dealing with the rapidly increasing volume of data is the efficient storage of data. It should be possible to add only new data during import and remove duplicate data.

Scalability and modularity (S2). As mentioned in the introduction, the amount of data created is increasing rapidly. The system should therefore be easily scalable in order to respond flexibly and efficiently to growing data volumes. In addition, the scalability must also ensure the availability of the system. Good scalability of such an extensive system also requires a certain degree of modularity in order to be able to

expand or replace individual services. Here too, the modularity of the system should meet future requirements so that the system remains adaptable in any case.

4.3 Business Requirements

Contrary to the user and system requirements, the business requirements are always dependent on the company. For this reason, two general business requirements are derived, which every company should pursue.

Cost-effectively development (B1). The system should be designed in such a way that it can be implemented cost-effectively despite the wide range of functions.

Time-saving development (B2). The analyses that are finally based on the combined, plausibilized data can have a direct impact on the company. In the case of the energy supplier, for example, it needs certain analyses in order to comply with certain regulations by a certain date. The analyses can also save costs or provide advantages over competitors, and since this system forms the basis of the analyses, this leads to the requirement for time-saving development.

5 Concept

The concept is based on the requirements mentioned in Chapter 4, as well as on the findings of the research of related work described in Chapter 3. For a more detailed consideration of this concept, the selected architecture is first examined in more detail in order to then explain the individual selected services of the architecture closer.

5.1 Architecture

For the selection of the basic architecture of the system, special attention must be paid to the system requirement **S2**, as the software architecture has a strong influence on this and must therefore fulfill it.

The *architecture characteristics rating* by Richard [Ric20], which rates known architectural styles in several categories on a star scale from 1 to 5 stars, was used for a first selection of possible suitable software architectures. The rating includes a total of eight architectures, each of which has its strengths in different areas. Due to the requirement **S2**, the characteristics of scalability and modularity were taken into account for the preselection of suitable architectures (see table 5.1).

Here, the architecture should have an average rating of greater than or equal to four in order to meet the requirement **S2**. With this restriction, the architecture styles *Layered Architecture Style*, *Pipeline Architecture Style*, *Microkernel Architecture Style*, *Service-Based Architecture Style* and *Orchestration-Driven SOA* are excluded and are

Architecture Style	**Modularity**	**Scalebility**	**Average**
Layered Architecture Style	⋆	⋆	1
Pipeline Architecture Style	⋆⋆⋆	⋆	2
Microkernel Architecture Style	⋆⋆⋆	⋆	2
Service-Based Architecture Style	⋆⋆⋆⋆	⋆⋆⋆	3.5
Event-Driven Architecture Style	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	4.5
Space-Based Architecture Style	⋆⋆⋆	⋆⋆⋆⋆⋆	4
Orchestration-Driven SOA	⋆⋆⋆	⋆⋆⋆⋆	3.5
Microservices Architecture	⋆⋆⋆⋆⋆	⋆⋆⋆⋆⋆	5

Table 5.1: Architecture Characteristics Rating [Ric20]

therefore not explained in more detail in this section. The remaining three will be considered in more detail for the final selection.

Event-Driven Architecture Style. The event-driven architecture (EDA) style is an asynchronous distributed architecture style in which individual components receive and process events asynchronously based on events sent over an event bus (see Figure 5.1) [Ric15].

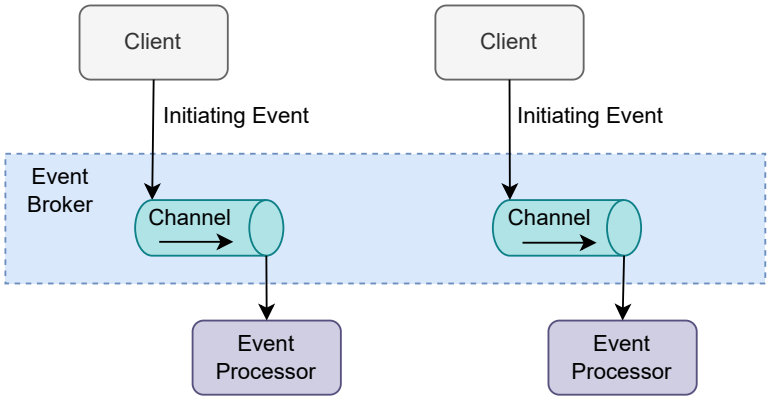

Figure 5.1: Event-Driven Architecture

This architecture style allows highly scalable and high-performance applications to be built [Ric20], which meets the **S1** and **S2** requirements. An event-driven architecture is also suitable for the two user requirements **U2** and **U4**, since both the data import is triggered by an event, which is triggered either manually by the user or periodically by the system, and the plausibility check of the data, which is to be checked each time the data is imported. Especially for requirement **U2**, the event-driven approach is advantageous because it offers a high advantage for processing a high number of data arriving at a high input rate. [Bru10]

However, this style has a disadvantage in terms of modularity, especially with regard to the use of open source software. Since communication takes place via event buses and the individual event processors have to be connected to these, the EDA offers a restriction with regard to the selection of various open source tools as individual services and also to their interchangeability. Another disadvantage is the complexity in development due to the asynchronous nature of the architecture style [Ric20].

Space-Based Architecture Style. The space-based architecture style is designed for applications that needs a high scalability. The higher scalability is achieved by representing a service with one or more processing units. A unit holds the required application data in the cache and replicates it to all other processing units of the service. This allows the number of processing units to be dynamically adjusted and thus scaled. Storing the data in the cache also optimizes performance. However, it is not well suited

for large database applications with large amounts of data and is also complex and expensive to implement, which is contradictory to **B1**. [Ric15]

Microservices Architecture. As the name suggests, the microservice architecture consists of small, independent services that each perform a specific task and thus work together as a complete system [New15]. The special aspect of these services is that they all represent an independent deployed unit and are therefore completely decoupled from each other. Communication between the services takes place via a remote access protocol such as REST, AMQP, etc. [Ric15] With this concept, the microservice architecture offers a high degree of modularization, as the service granularity means that a single service is used for a task, which can be connected via a defined interface, but can also be exchanged, which in turn leads to easy replaceability and thus meets the requirements **S2**. In addition, the completely decoupled services provide technological freedom in the development of the individual services, which means that the technologies can be selected to be more problem-specific [Wol18]. The latter characteristic is particularly advantageous for the goal of using open source software, as the selection of open source tools is greater. In contrast to the other architecture styles presented, the development is significantly less complex, as the development of the individual services is isolated from each other and changes usually only affect the individual service [Ric15].

It turns out that both the event-driven and the microservice architecture styles offer their advantages in order to fulfill the requirements mentioned. The architecture chosen as the basis for the concept of the system described in this thesis is therefore a mixture of both architectural styles. The basis is the microservice architecture style, which in some cases uses the event-driven style for the communication of services in order to eliminate bottlenecks, e.g. in case of a backlog of requests [Ric20].

5.2 Services

For a more precise definition of the individual services of the system, it is necessary to take a look at the important concept of a service within a microservice architecture. There are two basic rules: Firstly, there should be a low level of dependency between the various services in order to ensure *loose coupling*, and secondly, all components of a microservice should offer a *high level of cohesion* to ensure that they belong together. [Wol18, New15] Following these two rules, the architecture shown in Figure 5.2 was defined.

For a more detailed explanation of this concept, the individual services are described below. Three aspects are presented here. Firstly, it is explained why the part of the system represents an own microservice, secondly, the development of the service

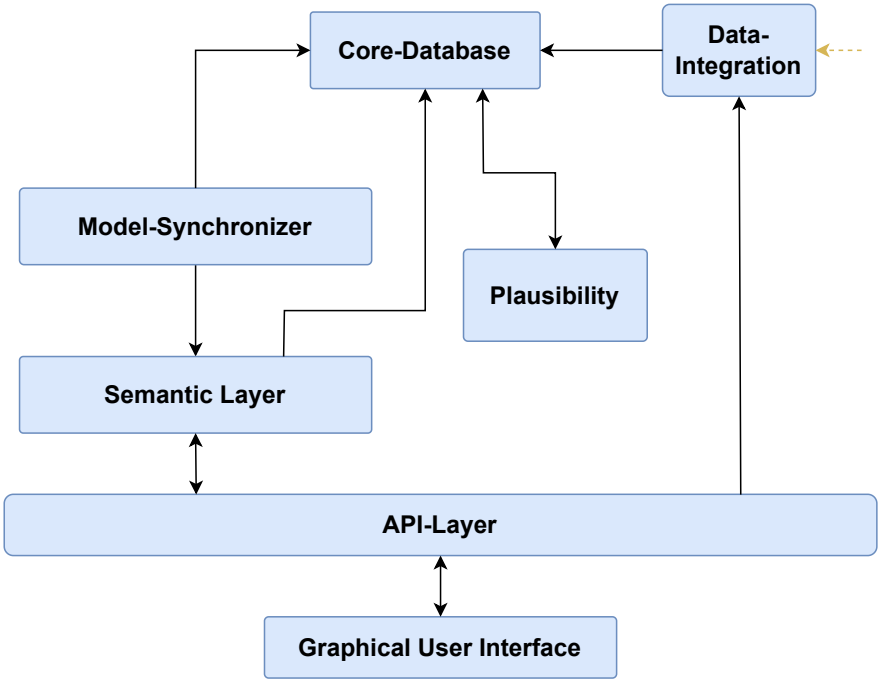

Figure 5.2: Overview of the Concept and its Microservices

structure is looked at and finally the final design of the microservice. In order to meet the requirements **B1** and **B2**, the concept is based on the use of open source software. However, the choice of software is not determined by the concept. Rather, the concept describes the individual components that can then be implemented using open source software.

5.2.1 Data Integration

Reasons for an own microservice

As described above, all components of a microservice should offer a high level of cohesion. With regard to the requirements from Chapter 4, **U1 - U3** represent just such a cohesion. The extraction of data from external sources, the following transformation and the final loading form a common process known as the ETL process. Such an ETL process does not necessarily have to be a separate service, but can also form a service together with the central database. However, the requirements **S2** speak against this approach. As ETL tools are often used for the implementation of ETL processes, such tools can be integrated into the system as an open source solution and exchanged if necessary; this is not the case if it is implemented together with the central database. In addition, outsourcing the data integration can also reduce the traffic on the central database [Sch16].

Deriving the design

There are three different approaches to integrating data from source systems into a target system: the *materialized approach*, the *virtual approach* and the *hybrid approach*. Basically, they consist of the same structure, the *data layer*, which contains all source systems, a *wrapper or charger layer*, which enables access to the source systems and the extraction of data from them, and the *warehouse or mediator layer*, which enables the retrieval of data via a global schema. [Mas24]

In the following, these approaches are considered in more detail with regard to the requirements of the system described in this thesis in order to derive the design of the service. An overview of all approaches can be seen in Figure 5.3.

Materialized approach. The materialized approach describes the approach of a classic data warehouse. Here, the data is extracted from the source systems and physically integrated into the data warehouse, which then provides the user with a unified view of the data. Users who want to query this data submit their queries to the data warehouse and not to the source system. An ETL tool is responsible for integrating the data in the materialized approach. [Mas24]

Virtual approach. With the virtual approach, the data is not transferred from the source systems to a target database but remains in the source systems. In order to provide the user with a uniform view of the data, a mediator is used instead of a data warehouse. This only holds the view of the data, but not the data itself. A wrapper is provided for each source system so that the mediator can query data from the data sources in a standardized language [Mas24]. In contrast to the materialized approach, the virtual approach has the advantage that fewer data is generated as it is not repoduced.

Hybrid approach. The hybrid approach was developed for more flexible data integration systems. This consists of a combination of the materialized and the virtual approach. Depending on requirements, the data is loaded either from a data warehouse or directly from the data sources. [Mas24]

With regard to the requirements set out in Chapter 4, the virtual approach can't be used. Since it should be possible to edit the data or add new data (**U5**), but these may not be edited in the source systems, a separate database is required in which entries can be written. With the hybrid approach, however, this would be possible. Here, new entries could be added to the data warehouse. However, the hybrid approach has a disadvantage compared to the materialized approach in terms of modularity. With the materialized approach, all data is transferred via the ETL tool. This makes it easier to scale and exchange (**S2**). There are also many ETL tools on the market, which makes

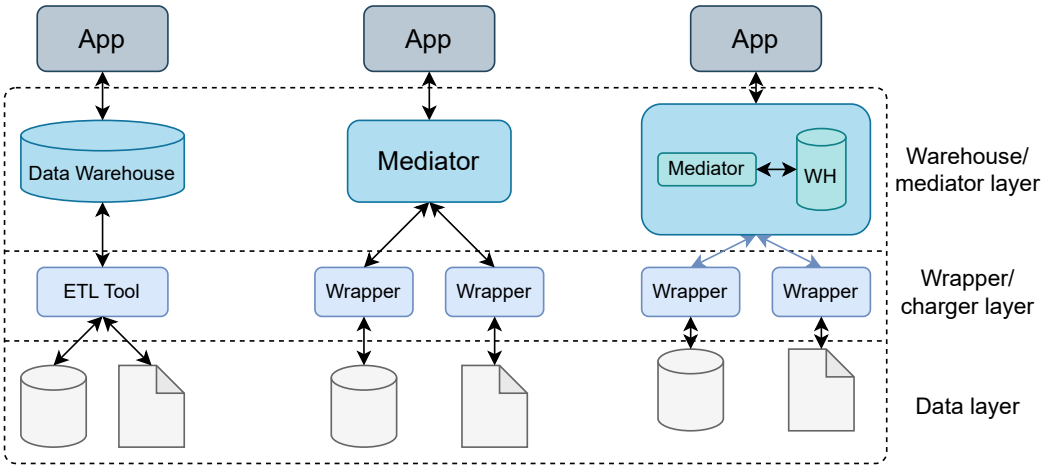

Figure 5.3: Data Integration Approaches [Mas24]

it easy to find an open source solution for the entire process and the integration process can be more clearly separated as a service. This benefits the microservice architecture. In addition, ETL tools often offer many connectors and a simple extension of additional connectors, which is not the case with wrappers. In this case, the wrappers would have to be re-implemented for each external source.

For these reasons, the materialized approach was chosen for the design of the Data Integration service, which basically implements an ETL-Tool.

Design

The Data Integration service consists of several components to map the functional scope of an ETL tool as shown in Figure 5.4.

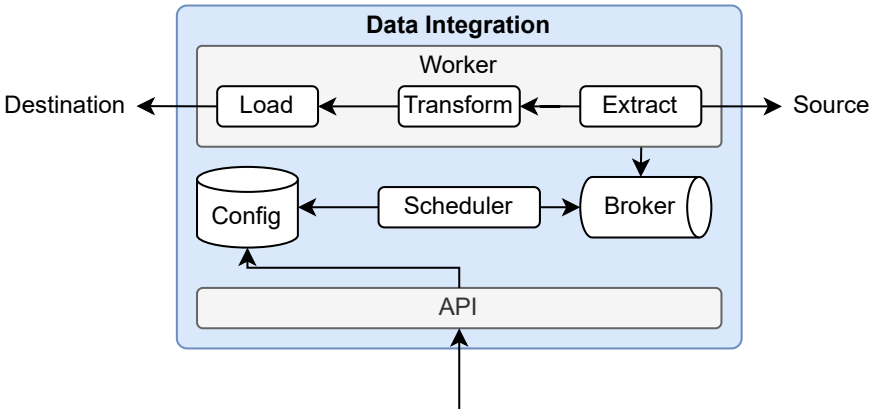

Figure 5.4: Design of the ETL-Service

The Data Integration service implements an event-driven architecture. This is because the execution of the ETL process is always triggered by a specific event and is then executed asynchronously. This architecture also ensures that all processes are eventually executed and are not lost, meaning that there is no loss of data to be integrated [Ric20].

API-Layer. To allow a user to configure the ETL process via a graphical interface and define stored transformation functions, the service requires a REST API that provides these functions.

Config-Database. Saving the information sent by the user through the API, the service must have its own database. This contains all the configurations for the ETL process, such as information about the data sources, transformation functions to be executed or the choosen extraction period. The *Scheduler* can then trigger the ETL processes based on this data.

Scheduler. To ensure that the ETL process is executed at the extraction period selected by the user, a scheduler is required that triggers the process at the times stored in the database. This task is performed by the Scheduler. Since the architecture of the data integration service is an event-driven architecture, the Scheduler acts as a producer. It creates jobs based on the information from the database, which it then passes on to the *Broker*.

Broker. The Broker acts as middleware between the producer and the conusmer. In this case, it receives the events from the scheduler and passes them on to the conusmer, which in this case is the *Worker*.

Worker. Once the jobs to be processed are available in the Broker, the Worker can take them and process them one after the other. For a simpler explanation, only one Worker is shown in Figure 5.4, but there can also be several workers processing the jobs in parallel. The worker itself has then implemented the ETL process already explained in Section 2.2, which then pulls the data from the external source using the functions stored in the job, transforms it if necessary and finally writes it to the Core Database.

With regard to the requirement **U6**, there is still a restriction in the transformation phase of the ETL process. Since the data should be available in the original data warehouse, the transformations are limited to the syntactic and structural transformations described in Section 2.2.2. There is a separate service for the semantic transformations to meet all requirements (see Section 5.2.3).

5.2.2 Core Database

Reasons for an own microservice

The Core Database represents the heart of the system. It forms the central data storage of the system and therefore has to fulfill several requirements. All data from the source systems should be stored in the database (**U3**), this data should be able to be modified by further services or by a user or new values should be added (**U4** and **U5**). Despite the availability of these options, all originally imported data should be available at all times. These requirements are closely related and should therefore be summarized in a service. In addition to the user requirements, however, it also makes sense to provide the central data storage in a separate service, as this ensures a loose coupling, which helps to fulfill **S2**. As a result, the service remains modular, and the data storage technology can be adapted as requirements increase.

Deriving the design

The core data initially consists of a database, which is why an extensive design for storing the data is not necessary at first glance. The choice of technology is the most important factor here, but this is not considered in this chapter and is only examined for the implementation in Chapter 6. Due to the requirements **U5** and **U6** set out in Chapter 4, however, a special design is required, as data should always be available in its original state, but should also be modifiable.

It is a well-known versioning concept that data is available in its original state as well as in all other states. All changes should be logged in the database and be available. With the 2016 version of SQL Server, Microsoft has provided a similar feature that uses temporal tables to track changes to data [Kon18]. The Microsoft feature is primarily there to enable simple point-in-time analysis. However, the concept behind it can be an inspiration for fulfilling the requirements **U5** and **U6** of the Core Database. System versioning from SQL Server 2016 is implemented with the help of a current and a history table. In the history table, two time columns are added in addition to the data to define the validity period of the data. The current table always holds the current valid value, while the historical table contains all previous values (see Figure 5.5). [Mic23]

One problem that arises from this approach is the overwriting of the original values in the current table. In the system on which the thesis is based, the ETL tool writes to the current table. If a user wants to edit or add values in it, according to Microsoft's approach, the values in the current table would be overwritten and added, and the previous values would be written to the historical table. However, this would then

Temporal Table
(current values)

Item	Value
A	100
B	50
C	200
D	175
E	450

1 m

History Table
(historic values)

Item	Value	Start	End
A	50	2023-09-01	2023-09-02
B	45	2023-08-31	2023-09-04
A	75	2023-09-02	2023-09-08
C	175	2023-09-01	2023-09-09

Figure 5.5: Temporal Tables in SQL Server 2016 [McD24]

result in duplicated values if the added value is finally added via the ETL tool. For the use case of the Core Database, it would make sense to divide tables into a table with the original data and the overwritten data for better analysis, instead of dividing historical and current data. In this way, the ETL could always write to the table with the original data, with changes made by the user ending up in the analysis data table. Since not all the most current values are in one table, a component would be needed that brings this information together and provides the user with the most current values in a standardized way. Here it is interesting to take a look at the hybrid approach to data integration described in Section 5.2.1. In this, a mediator is responsible for providing a uniform view of data from the source systems and those in the data warehouse. This approach can help, as a mediator is responsible for querying the data to determine whether data is read from the table with the original data or from the table with the analysis data, thus enabling a uniform view.

Design

The core database service consists of a database, a mediator and an API-Layer as shown in Figure 5.6.

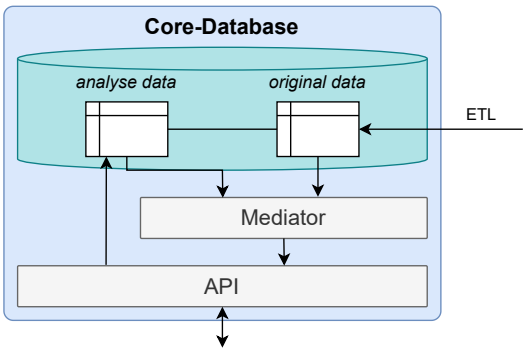

Figure 5.6: Design of the Core Database

Database. The Database contains two groups of tables. Firstly, all tables that are created by the ETL tool and therefore contain the original data from the source systems. This group of tables is described as *original data*. This second group is described as *analyse data*. These tables are a replica of the original tables, but only contain data that has been added or changed by a user or another service such as the Plausibility service (see 5.2.3). In addition, these tables also contain a modification date so that the mediator can deliver the current data to the user.

Mediator. The Mediator is responsible for providing a uniform view of both table groups. It is therefore possible to query all original data via the mediator, in which case it returns the data from the *original data* tables or all current data. In this case, the mediator searches for the most current values from the *original data* and *analysis data* tables and returns them in a merged form.

API-Layer. The API-Layer provides a REST API via which a user or a service can query the data via the mediator or edit or add data from the *original data* tables by adding entries to the *analysis data* tables. When querying the data, the user can decide what type of data they want to query by specifying parameters.

5.2.3 Plausibility

Reasons for an own microservice

The plausibility check of data is usually part of the ETL process. In this case, it belongs to the semantic transformations. Since the ETL tool's transformations are limited to syntactic and structural transformations, it needs a way to check the plausibility of data after it has been imported into the Core Database.

As database management systems have become more powerful and the need for a scalable solution for complex transformation processes has arisen, the so-called ELT (Extract, Load, Transform) approach has developed alongside the ETL approach. Here, the data is extracted from the source system and loaded in its raw form into the target system, where it is transformed on request [Ros13]. Since the Core Database and thus the target system is a separate microservice for the reasons mentioned in Section 5.2.2, the ELT approach is followed, but the transformations are outsourced to a separate service. This creates several advantages:

- Original data is available more quickly in the Core Database, as complex transformations are executed separately.
- The plausibility check service can be scaled separately.

- Plausibility checks can be developed independently of the ETL tool used.

Deriving the design

The plausibility check should always be triggered by the import of new data and the required transformations should be executed depending on the newly imported data. It therefore requires a service that detects changes in the database and executes the transformation process depending on the changes.

There is a *change data capture* procedure for the automatic detection of data changes in a database. All changes to a database are logged with the associated information. Many frameworks that implement such a procedure also offer the option of subscribing to such changes and thus always being notified of the latest changes. [Sch16]

In addition to receiving changes, the service needs the option for users to perform any transformations. The user should be able to specify which transformations are to be executed when changes are made to the database. These requirements have many parallels to *Continuous Integration (CI)* and *Continuous Delivery (CD)* pipelines in the software development process. In these, code changes trigger processes that the user can define themselves, e.g. the execution of special software tests, the validation of code style or the building of containers. For this reason, the architecture of CI/CD tools was considered in order to derive the design of the service. These tools have a similar basic structure as seen in Figure 5.7. First, there is an interface with which it is possible to trigger certain events. Depending on the event, a job is then created that contains all the information required to execute the desired process. Runners are used to execute the content of the job. These can be virtual machines or containers that are specially provided for the execution of the job and execute the desired process with the help of the information stored in the job. [Git24b, Git24c]

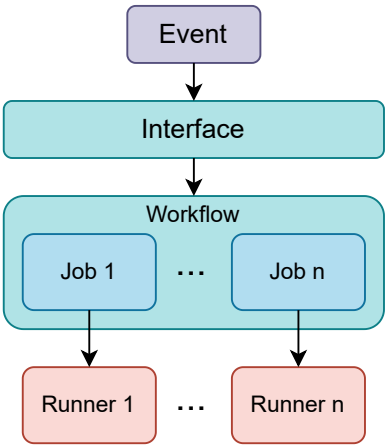

Figure 5.7: Basic CI/CD Architecture [Las23, Git24c]

To get a more detailed look at the architecture of such a tool, Weynand et al. [Wey17] presented the architecture of the continuous integration software *TravisCI* in more detail. As an interface for triggering events, Travis CI contains a listener that listens for changes from *GitHub* and then sends a request to the second component, the gatekeeper, which is responsible for creating the required jobs from this request. The Scheduler is the third component, which is responsible for managing all created jobs, e.g. by grouping and prioritizing them. The last component in the pipeline process is the Worker, which starts a virtual machine for the job planned by the scheduler and finally executes the job.

Design

The plausibility check service is based on this design described by Weynand et al. [Wey17]. Like the Data Integration Service and the architectures of the CI/CD tools, it follows an event-driven architecture and contains the following components:

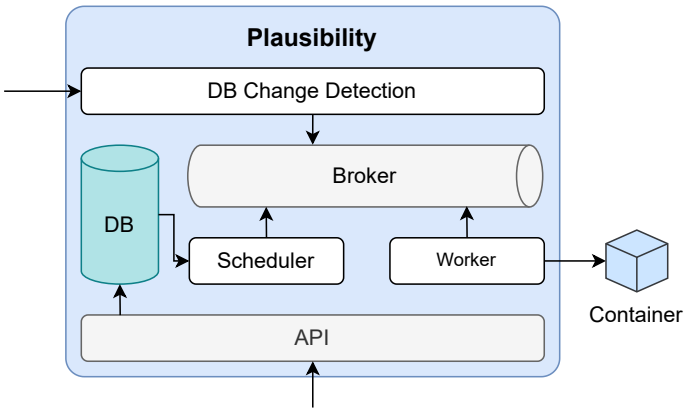

Figure 5.8: Design of the Plausibility Service

API-Layer. The plausibility check service should also provide a REST API for a user. This can be used to create so-called *tasks*, which defines which plausibility checks are to be executed for which database changes. For example, the user can specify that an address correction should be executed if data in the user table changes. In addition, dependencies between tasks can also be defined here if a plausibility check process requires another one to be performed first.

Database. The Database stores the tasks created by the user so that the scheduler can query this information from the database later in the process and finally create the job.

DB Change Detection. Database Change Detection is the service interface for the triggering events. As mentioned at the beginning of the section, an implementation of

the change data capture process is used here, whereby the service receives all changes to a database via this component.

Broker. The changes received by the DB Change Detection component are put in the Broker's queue. This ensures that all changes are processed at some point and are therefore also checked for plausibility.

Scheduler. The Scheduler component takes care of managing the tasks created by the user and the incoming database changes. For this purpose, the Scheduler is attached to the Broker as a consumer and therefore receives all incoming changes to the Core Database. If it receives one, it looks in the service's own database for all tasks created by the user that are dependent on the incoming change and creates jobs for them. It is also the task of the Scheduler to manage dependencies between tasks correctly and to ensure that all tasks are executed at some point. To do this, the Scheduler puts the created jobs in the Broker's queue so that the worker(s) can execute the jobs.

Worker. The Worker consumes the jobs placed in the queue by the Scheduler and creates containers for the execution of the necessary plausibility checks. It is responsible for managing the containers and for writing the results back to the Core Database.

Container. In contrast to the virtual machines used by Travis CI for executing the runners, containers are spawned by the Worker, which has the advantage that they are more lightweight (see Section 2.5) but also that plausibility algorithms can be developed independently of the platform and programming language and then only need to be available as a Docker image. Only the url to the image needs to be stored in the task configuration so that the Worker can download and start the container.

5.2.4 Semantic Layer

Reasons for an own microservice

To allow the user to create user-defined views of the data in the Core Database (**U7**), there needs to be a way to model data. It would be quite possible to add a component to the Core Database that lies as a layer between the API layer and the database. However, this would be to the disadvantage of modularity (**S2**), as the implementation of the data modeling is directly linked to the database, making it more difficult to exchange the database. In addition, the database would not scale independently in this case, which can be a problem, as it may be necessary for performance to cache the user-defined view of the data to enable faster queries. Depending on requirements, such a service can be scaled independently of the Core Database.

Deriving the design

It is a well-known problem that data in data warehouses is often not provided in the business presentation needed by the user due to the various sources. To counter this problem, there are two ways of making data available to the user in the business logic. Firstly, the data from the various source systems can be transformed during the ETL process so that it can be stored in the same entity in the target system. However, this requires complex transformation, depending on how much the schema of the source systems differs. In addition, this type of linking is not very flexible due to the complex transformation steps. The second option is the concept of the semantic layer. The semantic layer provides the link between the database and the platform that users use for analysis [Sch16]. Using this layer, a user can create semantic models that represent the business logic. This is achieved by using a semantic model as a projection and mapping the entities and relationships of the business logic to the database schema [Hos24]. In addition, metrics can be defined in the semantic model that are not available in the raw data, such as the turnover of a company. Furthermore, a semantic layer can provide performance improvements for user queries. It achieves this, for example, by pre-storing users' database queries, which minimizes traffic to the Core Database, or by pre-calculating metrics in order to make them available to the user more quickly. A semantic layer that is independent of the raw data and the analysis platform, as in this case, is referred to as a universal semantic layer. [AtS24]

Design

In conventional universal semantic layers, the semantic models are predefined views of the raw data [AtS24]. With a view to request **U7**, however, it is necessary to create the semantic models and thus the views of the raw data dynamically via a user interface. For this reason, the semantic layer of this system basically requires four components (see Figure 5.9).

API-Layer. Using the service's REST API, a user can add metrics to existing data models, link data models together and create new views from linked data models. Furthermore, all data queries for analyzing the data of the Core Database are made via this interface. This means that the user always queries the created business logic of the data and can benefit from the performance optimizations described.

Config DB. To store all data modeling performed by the user, the service has a configuration database. This only holds the information about the data models, views and metrics created.

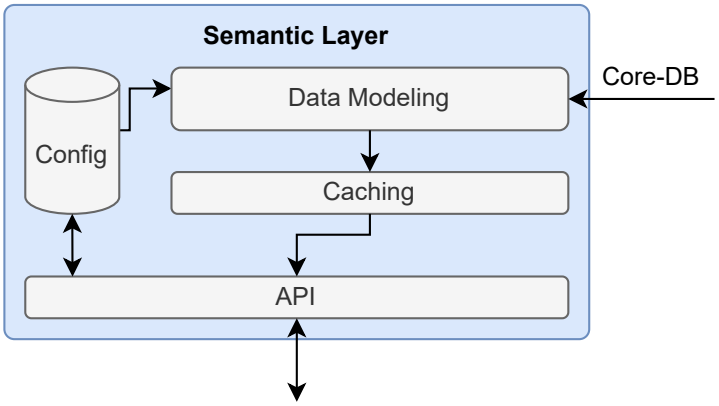

Figure 5.9: Design of the Semantic Layer

Data Modeling. The data modeling component is responsible for the described transformation of the technical view of the Core Database into the business logic defined by the user. This component takes the information stored by the user in the configuration database and dynamically creates the views of the raw data.

Caching. To avoid having to recalculate all views and metrics every time the user makes a request via the REST API, there is a caching component that keeps the data in the cache and can therefore make it available to the user more quickly.

5.2.5 Model Synchronizer

Reasons for an own microservice

With the use of a data integration service and a universal semantic layer, a problem arises with regard to the synchronization of tables newly created by the data integration service and the models provided via the semantic layer. The user would like to have all existing tables of the Core Database available during modeling and not have to create them first when importing from a new source system. This requires a service that registers newly created tables in the Core Database and automatically creates a model for this data in the semantic layer. Since the semantic layer is independent of the Core Database used, the model synchronizer should also be implemented independently of both services to ensure modularity.

Deriving the design

There are two ways to log changes in a database: Trigger and change data capture. The former was used in the plausibility check to track all changes in the Core Database.

The Model Synchronizer, on the other hand, only requires the logging of changes regarding the creation and modification of tables, which in SQL are commands of the data definition language. Triggers that listen to these commands and then create data models in the semantic layer are therefore suitable here. These triggers must therefore be implemented in the Core Database so that they are written to a channel when tables are changed.

Design

In line with the simple derivation of a Model Synchronizer design, it also has a correspondingly simple structure (see Figure 5.10). It consists of a component that receives changes by consuming a channel, then translates these into the structure of the data models of the semantic layer and finally creates the data models by making a request to the interface of the semantic layer.

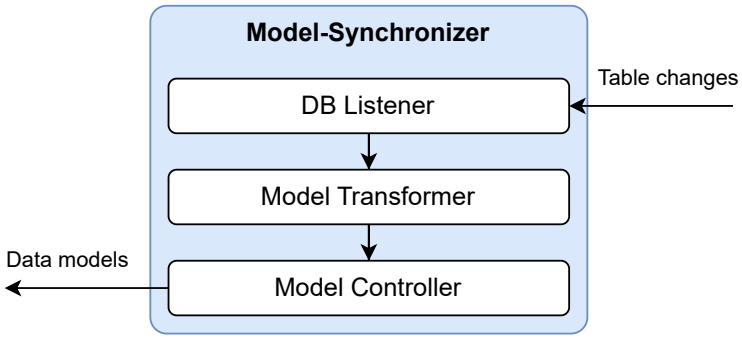

Figure 5.10: Design of the Model Synchronizer

5.2.6 API-Layer

Reasons for an own microservice

By using the microservice architecture, each service offers its own, usually fine-grained interface for certain configurations or queries [Zha18]. So if a user wants to use all the functions of the system, they must use a client that interacts with the many different services. This leads to a strong coupling between the client and the individual microservices, which has disadvantages. The client must have precise information about the structure of the system, i.e. how many services it is divided into. In addition, it must be adapted each time the services are further developed, as breaking changes to the microservices automatically require breaking changes to the client application. Another problem is performance. Since the use of many microservices can result in several services having to be requested when loading a user interface, this can lead to

high latency, which can result in multiple network round trips between client and server. A final major disadvantage is the security of the interfaces. As every microservice must be able to be requested by the user, every service interface must also be publicly available. This means that each microservice is also responsible for the security of the interface. On the one hand, this increases the attack surface of the system, but on the other hand it also increases the development effort. [Mic24]

To deal with these disadvantages, it therefore seems reasonable to provide a separate API layer as a service that manages the communication between the clients and the microservices.

Design

As the described problems with communication between clients and microservices are known, there is a pattern that stands for the design of the required API layer as a single service — the API gateway pattern. The API Gateway forms a central management interface at the boundary of the system [Zha18] and is therefore located between the clients and the microservices. It acts as a kind of reverse proxy that forwards the client's requests to the individual microservices [Mic24]. In addition to managing requests, the API gateway can also offer other advantages such as load balancing, authorization, caching and much more.

According to the API gateway pattern, the API gateway is located between the microservices and the client. And contrary to not using an API gateway, it manages all requests from one or more clients as shown in Figure 5.11.

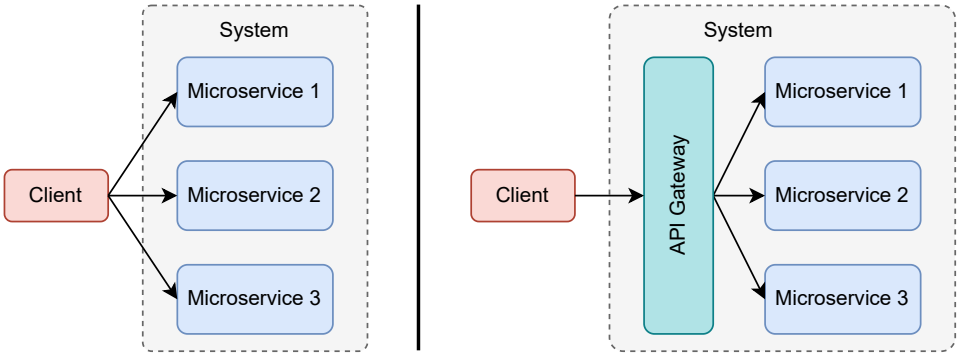

Figure 5.11: Design of the API Gateway

5.3 Graphical User Interface

Due to the described microservices and their API layers, which are available collectively via the API gateway, the system now offers almost the entire range of functions and therefore fulfills the requirements specified in Chapter 4. With **U8**, however, one of the main requirements is not yet fulfilled. In order for users to be able to use the entire range of functions without in-depth IT knowledge, a user interface is required that allows all functionalities to be managed in one application via graphical components.

As the concept of the technical implementation depends strongly on the environment in which the application is to be integrated, this section only deals with the concept of the graphical user interface components. The concept of an exemplary implementation as a web application can be found in Section 6. The following sections therefore present concepts for graphical low-code data modelling, data integration and plausibility checks.

5.3.1 Data integration

There are two variants on the ETL tool market for the low-code configuration of data integration and therefore the ETL process. The most common variant is the use of interactive flow charts to create pipelines, which then define the ETL process. The second variant is the less common, but simpler variant for the user - configuration with the help of a so-called stepper, i.e. the specification of information in several predefined steps. This type of implementation is used by Airbyte, among others.

The use of an interactive flow chart offers advantages when more complex pipelines have to be created by a user, e.g. by combining several transformations one after the other. However, in view of the required range of functions of the system conceptualized in this thesis, the types of transformations are limited. In particular, the transformations that require user interaction are limited to the structural transformations, while the syntactic transformations can be executed automatically by the ETL tool depending on the target system. For this reason, implementing the graphical user interface for configuring the ETL process as a stepper is the more user-friendly option in this case. The structure of the stepper is divided into the following steps:

Type of source system. The first step is to select the type of source system, as different options have to be configured in the next steps depending on this. The types provided depend on the connectors provided by the Data Integration service. Many open source ETL tools offer a wide range of connectors out of the box, but also the option to extend the tool with additional connectors. If an additional connector needs to be added, it must be implemented in the Data Integration service, but also provided in the interface.

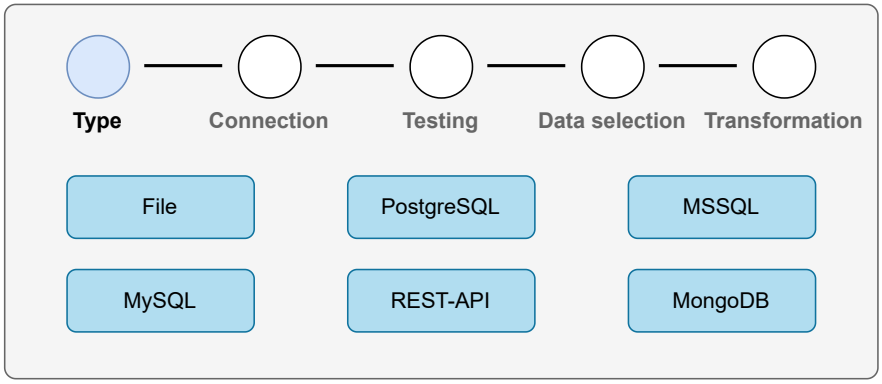

Figure 5.12: Wireframe of the Source System Type Selection

To get a more precise idea of the stepper, a sample type selection of the source systems can be seen in Figure 5.12. Here, for example, the connectors for an automated or manual import of files, from a REST API and from various database management systems are offered. A distinction must be made here between the different systems, as, as already mentioned, further configurations differ primarily in the connection to the database.

Connection. After selecting the type of source system, the connection to it must be configured. A form is therefore rendered here, which provides all possible configurations for the connection to the source system. This can be, for example, the host, the port, the username or a password for authentication. For files, the file format would still have to be specified here, but this becomes obsolete if a database is selected as the source system.

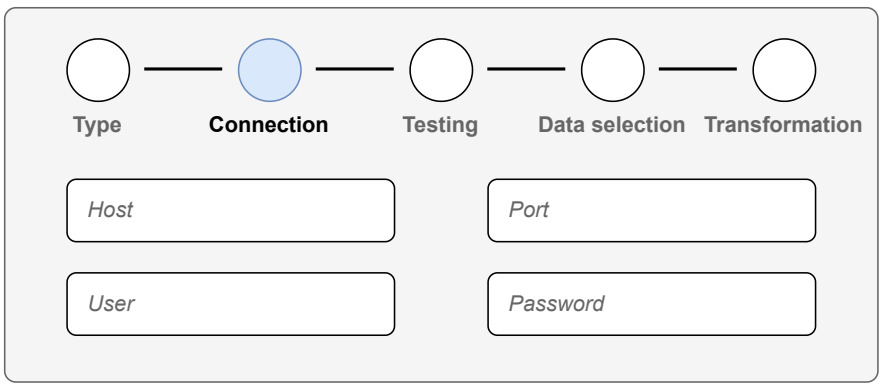

Figure 5.13: Wireframe of the Connection Configuration Step

In addition to configuring the connection, the import time must also be configured in this step as part of the connection between the ETL tool and the source system. Here there should be a choice between manual and a specification of regular times.

Testing. To ensure that an incorrect configuration of the connection is not only noticed on the first attempt to import, the connection should be tested in the step after

configuration. To do this, the system should establish a connection in the background using the specified configurations and display this to the user in this step in the case of success or failure.

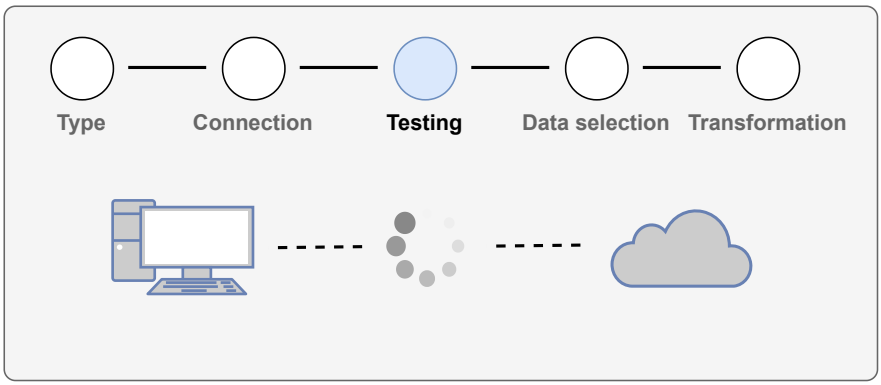

Figure 5.14: Wireframe of the Connection Testing Step

Figure 5.14 shows an example of this step, as the client attempts to establish a connection to the external source system. If this attempt fails, the user should be redirected to the previous step to adjust the incorrect configurations.

Data selection. If the connection to the external source system can be established, the user has the option of selecting the data to be extracted from the source system. They can select entire tables or individual attributes. In addition, they can select the extraction mechanism for the various tables as described in Section 2.2.1.

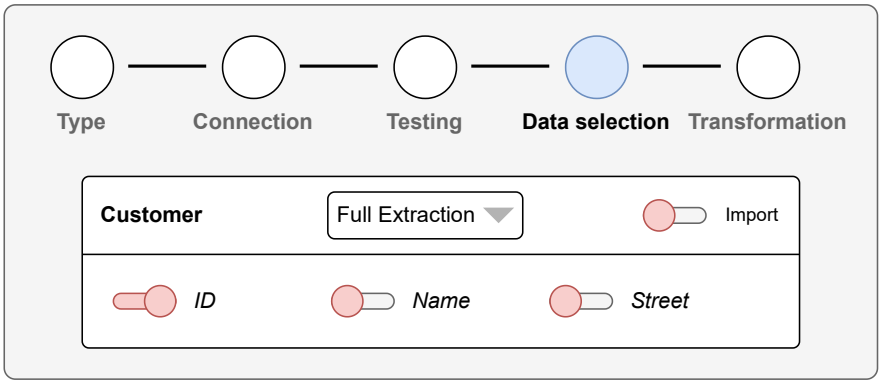

Figure 5.15: Wireframe of the Data Selection Step

In Figure 5.15, for example, the user decides to import the Customer table with the *ID* attribute. The complete data set is to be overwritten with each import by selecting the *Full Extraction* mechanism.

Transformation. Due to the limitations of the transformations in the ETL process, no complex user interface is required for data integration. Similar to step one, the implemented transformations are displayed here, for which further settings can be made if required. Again, the transformation algorithms are added to the ETL tool during

development, which means that the user only has to make a selection on the user interface, but does not have to develop any complicated transformations themselves.

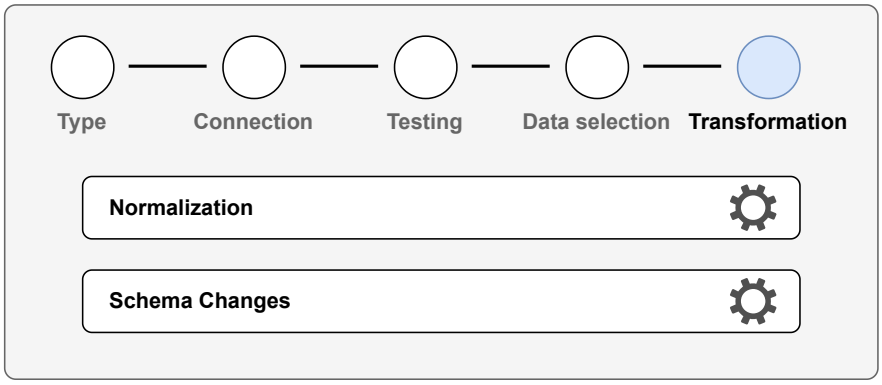

Figure 5.16: Wireframe of the Transformation Step

Figure 5.16 shows examples of the transformations *Normalization* and *Schema Changes*. This allows the user to decide without their own implementation that unstructured or semi-structured data from the source systems is normalized and thus structured if required. When making changes to the schema, the user has the option of renaming columns, for example.

5.3.2 Data modeling

The biggest challenge of the graphical user interface is the provision of data modeling as a low-code approach. Normally, data modeling is an important step in the design process of a database. A suitable data model should be found in close cooperation with the company or customer [Dav16]. However, using a central database that only contains data from external source systems creates challenges in terms of data modeling, which are solved in this system by the semantic layer. This translates the technical views of the various data sources into those of the business logic. The mapping of the technical view into the business logic also normally takes place in the semantic layer during the design process. It would therefore be part of the development of the *Semantic Layer* service. As already described in the service (see Section 5.2.4), this should provide all imported tables as data models and enable a user to connect data models, extend data models and create new views of these models via an interface. These functions should now be available in the graphical user interface in the low-code approach.

Joining data models

In order to develop a graphical user interface for connecting data models, a look is first taken at the connection of tables at database development level, more precisely

at connections with SQL. In SQL, connections between entities are realized by means of so-called joins. For this purpose, both tables to be joined are specified as well as a binding under which the entries are merged (see Listing 5.1).

Listing 5.1: Simple SQL Join Statement

```
SELECT * FROM table_a JOIN table_b on table_a.b_id = table_b.id;
```

In most and simplest cases, the condition is simply a comparison of two values in the respective tables, but it can also be more complex conditions. In addition to the conditions, different joins can also be used in SQL, which ultimately lead to different results when querying data via the joins. To conceptualize a complete low-code solution for the connections between two data models, these types of joins must also be considered. A basic distinction is made between four joins: The inner join, outer join, left join and right join.

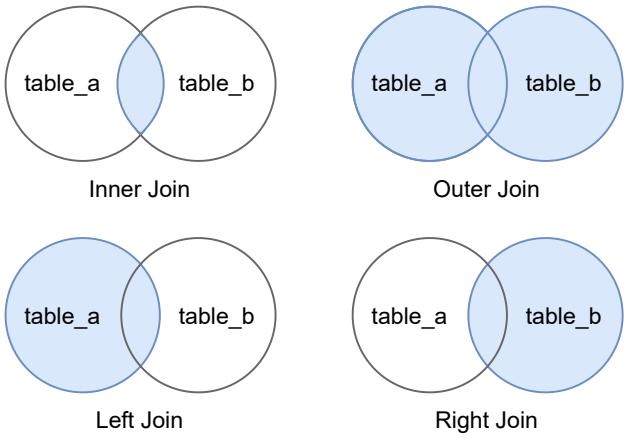

Figure 5.17: Four Types of Joins

As can be seen in the Figure 5.17, the different joins can be used to obtain different intersections of the tables. An inner join therefore only returns data for which there are values in *table_a* and in *table_b* after applying the condition. An outer join returns all entries, a left join returns all entries from *table_a* and the corresponding entries from *table_b* and a right join returns the opposite.

For the graphical user interface, a representation is required that shows the entities and their relationships. The *entity-relationship model (ERM)* exists precisely for such a representation. It contains three basic elements: The *entities*, their *attributes* and the *relationship types* between the entities. The representation of the data models is also inspired by this model. As shown in Figure 5.18, the data models are represented as nodes in an interactive flowchart. In contrast to ERMs, the attributes of the data models are displayed within the nodes for a better overview. As with ERMs, the connections

are shown as edges between the data models. However, not only is an edge drawn between data models, but also between the attributes of the data models. This has the advantage that it is easier to create simple joins, as the condition of the join can be created by drawing a connection between two attributes. The join shown in Listing 5.1 can be created simply by drawing a connection between the attribute *b_id* of *table_a* and the attribute *ID* of *table_b*.

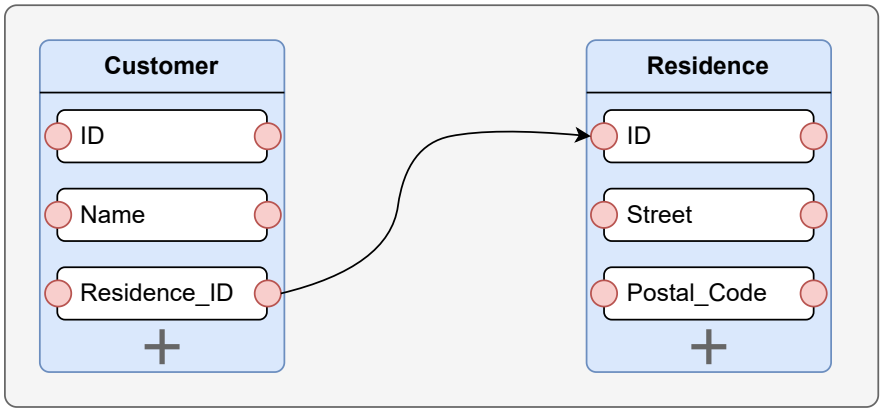

Figure 5.18: Wireframe for Joining Imported Tables

As there are different types of joins as described, this option must also be available to the user in the interactive flow chart. In order to implement this as intuitively as possible, the connections are created as left joins by default. If a line is drawn from the attribute *Residence_ID* to the attribute *ID*, as in the example in Figure 5.18, this is similar to the SQL join *Customer LEFT JOIN Residence*. The implementation of a *RIGHT JOIN* is therefore not required, as in this case the edge must simply be dragged in the opposite direction. The edge has an arrow at the end so that the join direction can be clearly recognized, and the user can follow this direction. The *INNER JOIN* and *OUTER JOIN* can be implemented by dragging connections in both directions and thus displaying an arrow at both ends. If a connection has such a double edge, the user can select whether it is a *INNER* or *OUTER JOIN* when clicking on the edge.

Adding new measures

The addition of new measures should also be available to the user in the low-code approach. For the design of this interface, the possibilities for adding new measures in SQL will be looked at first.

It is important to mention that adding new measures is not about adding new columns to a table, the structure of the imported table remains unchanged. It should only be possible to add further measures based on the existing columns. So in the SQL

environment, it is still only about statements of the data query language (DQL) and not the data definition language (DDL).

To add such new measures, either aggregation functions or calculations as formulas can be used in the SELECT clause in SQL. For example, as shown in Listing 5.2, a discount or the number of all items that exist can be queried.

Listing 5.2: Measures in SQL

```
SELECT price * discount FROM item;
SELECT count(id) FROM item;
```

To make these options available in the user interface, a measure can be added to the interactive flowchart in the data models by clicking the plus at the bottom of a node. The new measure can then be given a name, regardless of whether it contains a formula or an aggregation function.

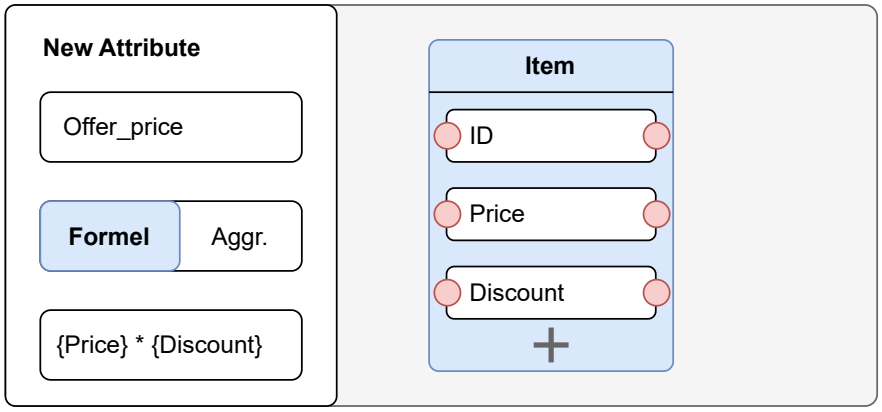

Figure 5.19: Wireframe for Adding new Measures

If a formula is selected, as in the example in Figure 5.19, it is possible to specify a formula. These can be simple calculations, but also complex SQL commands, which can also occur in every SQL statement in the select clause. When entering formulas, the user is supported by a formula editor that opens when the formula is entered (see Figure 5.20).

Create new data views

Thanks to the possibility of joining tables and adding new measures, a user now has the option of combining data from different source systems and adding new values that are useful from a business perspective for analyses or similar. In order to actually display the data in the business logic, it must still be possible to define new views of the data in order to query the data more specifically as required. For example, if customer data

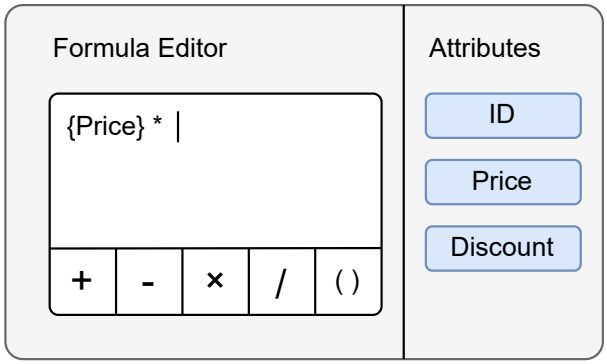

Figure 5.20: Wireframe for the Formula Editor

and building data are in different tables in the technical view, but a department needs a view of the customer with its buildings, it should be possible to define such a view on the graphical interface so that all users of the department can only query the combined view.

Creating a new view on several tables in SQL seems to be complicated for non-IT specialists at first glance. As can be seen in Listing 5.3, a view must first be created in which the desired data is added to the view using SELECT statements.

Listing 5.3: Creating a View in SQL

```
CREATE VIEW customer_data AS SELECT * FROM customer JOIN residence ON
    residence_id = id;
```

As can be seen in the SQL statement, the view is created from a join of two tables. Since the user has already created the joins via the interface, they only need to select the data models from which they are to be created and which attributes of the data models are to be added when creating the new views.

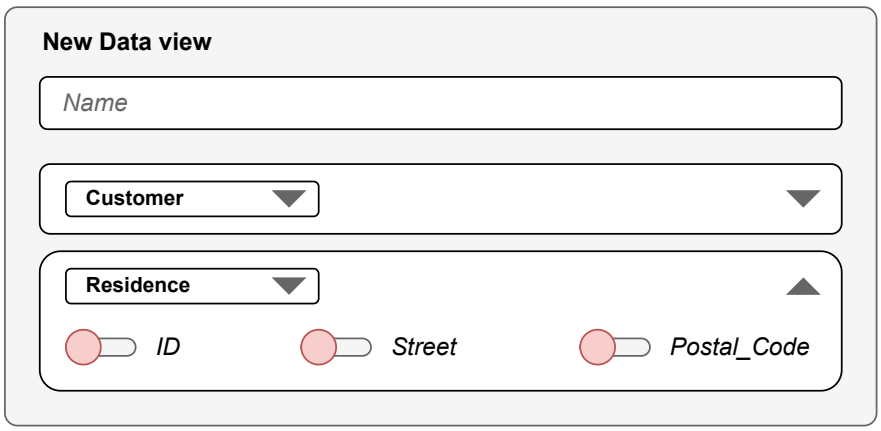

Figure 5.21: Wireframe for Creating new Data Views

As can be seen in Figure 5.21, the new view can first be given a name and then the data models to be added to the view can be selected from a drop-down menu. It is important that only one data model is selected first (in the example here, Customer). Then a second input field is displayed, which only has data models for selection that are directly or indirectly linked to the first table specified. This ensures that the view can also be realized technically. This interface, combined with the previously presented ones for joining data models and adding measures, now gives the user the full ability to translate the technical view in the Core Database into the required business logic using the low-code view.

5.3.3 Plausibility check

While the configuration of the transformation step in the ETL process fits into one step, the requirements for the configuration of plausibility checks by a user via the graphical user interface are more extensive. Although it is also the case here that the user has to choose from selected implemented plausibility algorithms, (s)he has several options for the configuration. In order to keep it as simple as possible for the user and to guide them through the configuration process, a stepper is also used here, which is divided into the following four steps.

Runner Selection. The first step is to select the algorithm for the plausibility check. From a technical point of view, this is the runner that is started by the worker as described in Section 5.2.3 and contains the plausibility check algorithm.

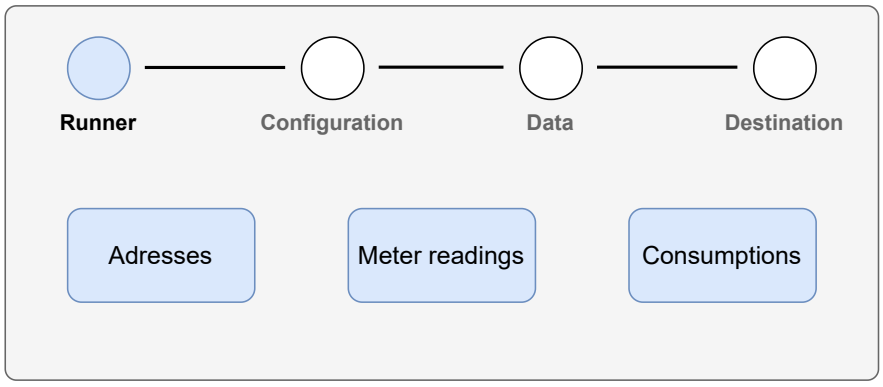

Figure 5.22: Wireframe for Choosing the Runner

Similar to the stepper for the configuration of the ETL process, the structure of the next step also depends on the choice of runner.

Runner Configuration. Depending on the runner selected, there are various configuration options in this section that are required so that the runner can be executed correctly later when the specified tables are changed.

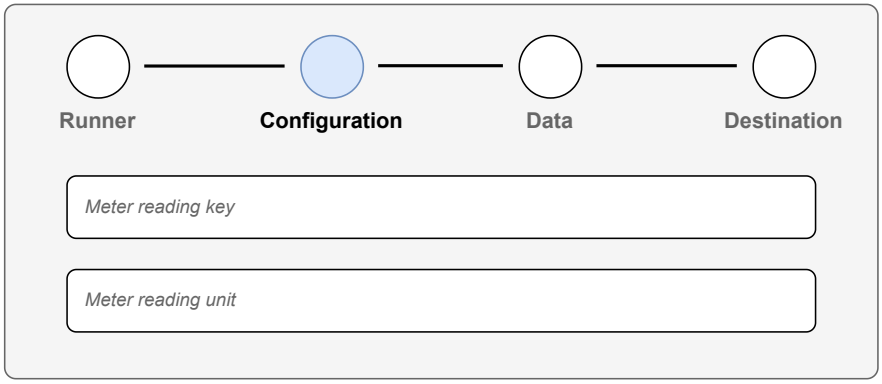

Figure 5.23: Wireframe for Setting Runner Configuration

If, for example, imported meter readings are to be checked for plausibility, it may be necessary to specify the attribute of the data model as *meter reading key* so that the runner knows in which field the value is located.

Data. The user must make two configurations in the data step. On the one hand, (s)he selects the data models for which the created plausibility check is to be executed when changes are made, and on the other hand, (s)he can specify further data models that are to be fetched by the plausibility check service and transferred to the runner. This may be necessary if, for example, a plausibility check requires not only the newly added data but also data from the past. So that not all data is fetched, the user can also specify the required attributes and filters for each table, so that, for example, only the meter readings for one year are fetched from the Core Database.

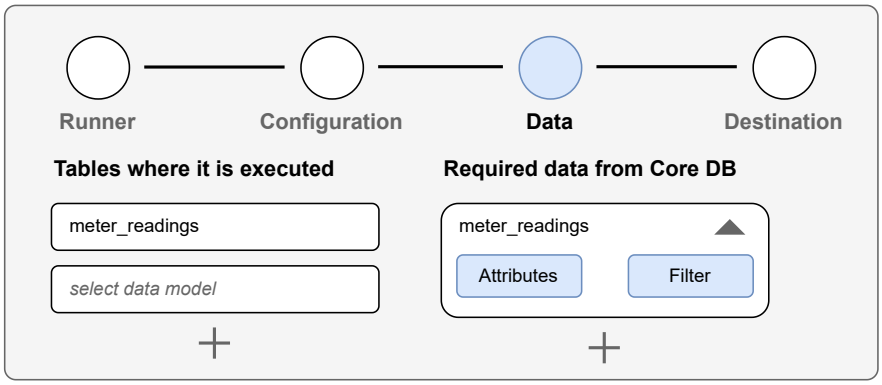

Figure 5.24: Wireframe for Choosing Data

Destination. The last step is to specify a target table. Due to the structure of the Core Database and the requirement that no original data is overwritten, the target table must be specified with the desired structure. This can either be a replica of the original data model or a completely new one if the user wants to have the plausibilized data in a separate data model.

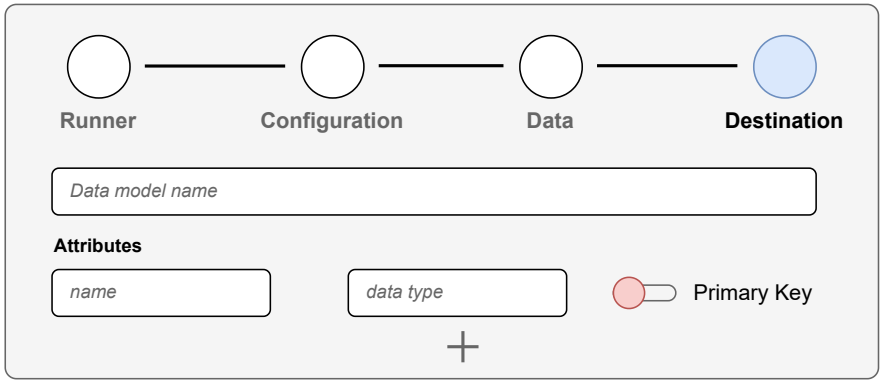

Figure 5.25: Wireframe for Setting Destination Data Model

As shown in Figure 5.25, the user has the option of giving the new data model a name and adding any number of attributes to it. This data model is then created once by the Plausibility service if it does not already exist. Finally, the results are then written to this table.

5.4 Summary

For the development of the concept, the selection of the appropriate architecture was considered first. A mix of the microservice architecture and the event-driven architecture was chosen, as these two offer the best trade-off between scalability, modularity and performance. Since the microservice architecture forms the basic structure of the system and the event-driven architecture is used in individual services or for communication between services, the system was divided into six services. To configure and execute the process of combining the data from the various source systems, a *Data Integration* service was designed that implements the ETL process based on the event-driven architecture. This service then loads the data into the *Core Database* service, which stores the data in a database. Changes in this database are always transferred to the *Plausibility* service. This performs the plausibility checks defined by the user. The fourth service is responsible for translating the data in the Core Database from the technical view into the business view. The concept of a *Semantic Layer* is used, which has this mapping as its task and offers further advantages such as caching. For the synchronization of newly created tables with the semantic layer data model, there is the *Model Synchronizer* service, which receives table changes in a database and translates them into the structure of the semantic layer data models. The last service is the *API-Layer*, which manages the communication between a client and the individual services. In order to offer an easy-to-use platform in the form of a low-code platform, a *Graphical User Interface* was designed in addition to the individual services, which enables a user to operate the functions of the individual services without in-depth IT knowledge.

6 Implementation

As mentioned in Chapter 1, the concept of the low-code platform for versatile data integration was developed for an implementation at an energy supply company. In addition to the complete development of the concept, the implementation was also realised, which is partly discussed in this chapter. It is not about the implementation of every single service, but rather about individual challenges in the implementation of the services or the graphical user interface and the possible open source software selection. How these challenges were solved and which open source tools were used to develop the system is described in the following Sections. First, the conditions under which the software was developed are considered in order to better understand certain decisions in the implementation of the concept.

6.1 Prerequisites

The energy supply company operates several hundred heat generation systems. In order to analyze their monthly consumption and the efficiency or coefficient of performance (COP), it needs all the meter readings from the systems. These include electricity meters, water meters, gas meters, oil levels and much more. As these meter readings are read by various systems and therefore end up in different source systems, the system developed is intended to merge and plausibilize the data from the various source systems and display it to users in the business view. The data required for a detailed analysis is stored within the company in Excel files, in Microsoft SQL databases and behind a REST interface.

6.2 Data Integration

For the implementation of the data integration service, the integration of an open source ETL tool was chosen as already mentioned in Section 5.2.1. This tool must cover all the functions set out in the concept and, in view of the company's environment, offer the option of importing data in the format of the existing source systems.

6.2.1 Software Selection

For the selection of the open source tool for data integration, an extensive Internet search was first carried out in order to obtain as complete a list as possible of all the open source ETL tools that could be considered. In the end, a list of twelve ETL tools was drawn up which, at first glance, were suitable for the requirements set out.

- Airbyte
- HevoData
- CloverDX
- Pentaho
- Apache NiFi
- Mage
- Kafka
- Singer
- CloudQuery
- Apache Camel
- Logstash
- Talend

An initial closer look at the selected tools allows seven to be excluded before a more specific comparison is made. Although HevoData describes itself as an open source solution, the source code is not online, which is why this is not an option. CloverDX does not make its source code available either. Talend and Singer both do not provide APIs, which is necessary for a connection to the graphical user interface. Although CloudQuery itself is open source, all but two of the required connectors are not. Apache Camel is not an ETL tool, but a framework for implementing an ETL. Pentaho is only available in an outdated version as open source and without good documentation, as there is now a commercial version.

For a more specific comparison, this leaves the five systems Airbyte, Apache NiFi, Mage, Kafka and Logstash. All of these systems offer the required scope of connectors and would therefore be able to extract data from the energy supplier's existing source systems. In addition, they all offer an interface for the user to configure the desired ETL processes. In terms of transformations, all offer the option of implementing custom transformations. Airbyte also offers a build in normalization of the data, which is appropriate in the case of the current system, as the Core Database is an SQL database (an explanation of this decision can be found in Section 6.3). Ultimately, the desired scope can be implemented with all the other available tools. In this case, the decision was made in favor of Airbyte, as it offers built-in normalization and a better overall impression in terms of installation, use and documentation.

6.2.2 Airbyte

Before taking a closer look at the integration of Airbyte into the system, the structure of Airbyte will be first explained and what possibilities this tool offers to meet the requirements for data integration. Airbyte essentially consists of five components [Air24a]:

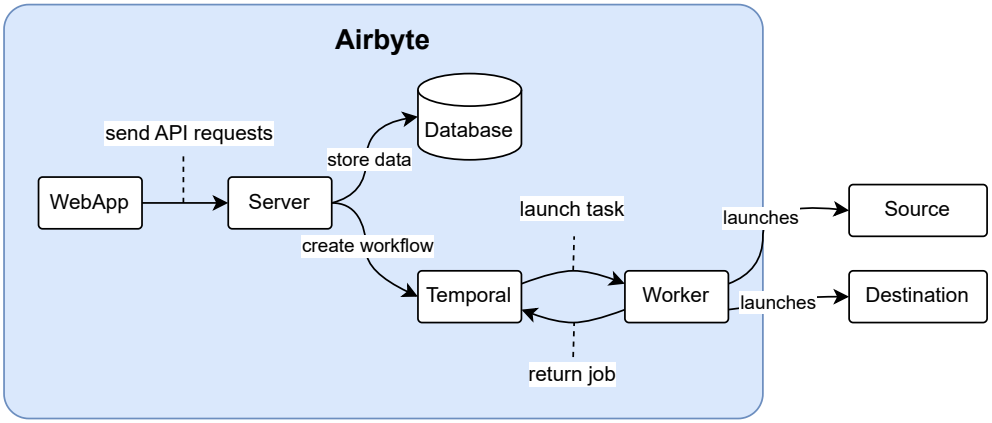

Figure 6.1: Architecture Overview of Airbyte [Air24a]

Web application. Airbyte already provides a graphical user interface that can be used to interact with the Airbyte API. This interface can be used to configure connections between source and target systems. Airbyte distinguishes between sources, destinations and connections. Sources and destinations each have a pool of connectors from which a selection can be made and for which the corresponding configurations, such as host or port, can be stored. Connections are the combination of defined sources and destinations as well as the selection of the data to be imported.

Server. The Airbyte server offers two REST APIs via which sources, destinations and connections can be created and tested. One of the REST APIs is the *Airbyte API*, which allows users to programmatically control Airbyte, and a *Configuration API*, which is designed for communications between different Airbyte components. The latter is far more powerful.

Database. The Airbyte database is responsible for storing all configurations of sources, destinations and connections, as well as for storing jobs to be executed.

Temporal Service. The Temporal Service manages the task queue and the workflows. It therefore represents the scheduler as presented in the concept (see Section 5.2.1).

Worker. The Worker connects to the source system and extracts the data from it. It then establishes the connection to the target system and imports the data. Depending on the number of tasks to be executed, there can be any number of workers to execute them.

A closer look at Airbyte shows that the architecture is quite similar to the structure of the Data Integration service developed in the concept and therefore fits very well as a service in the system. By configuring multiple targets, Airbyte can also transfer data to different target systems. As this exceeds the requirements of the system to be

implemented and makes the configuration more complex, the configuration process can be simplified using a separate user interface as described in Section 6.6.1.

6.2.3 Integration

The selected ETL tool needs to be integrated into the overall system, which is briefly discussed in this section.

As can be seen in Figure 6.1, Airbyte consists of several components. In order to run Airbyte on its own server, an instance must be started for each component. Airbyte relies on containerization (see Section 2.5). It thus provides a container for each component. In order to relieve the developer of configurations such as dividing the containers into individual networks and setting environment variables during integration, Airbyte uses *Docker Compose*. Docker Compose is a tool that makes it possible to run applications consisting of several containers [Doc24]. The various containers can be defined together with their configurations, such as networks or environment variables, in a YAML file, which is then used as the basis for creating and managing the containers.

To start an instance of Airbyte, it is simply necessary to copy the YAML file to the server and execute the *docker compose up -d* command. Airbyte can then be accessed via the port specified in the YAML file. As Airbyte's own web interface is also a separate container and is not required in this implementation due to its own graphical user interface, the container *webapp* can be removed from the YAML file. This means that only the required Airbyte services are started.

6.3 Core Database

6.3.1 Software Selection

In order to find a suitable selection of software for data storage, another extensive Internet search was conducted. The following eight software solutions were considered as potential technology for the Core Database:

- Apache Cassandra
- ArangoDB
- CouchDB
- MariaDB/MySQL
- PostgreSQL
- SQLite
- Apache Hadoop
- Greenplum

The database systems CouchDB, ArangoDB and Apache Cassandra are not suitable for the development of the required system, as these are special NoSQL databases that only allow very limited data analysis options. Their strength lies in the storage of very large volumes (TB, PB) of unstructured data. Since the energy supplier's data is available in Excel tables or SQL databases (which means they are already structured) and has a volume of MB to GB, relational databases offer the greater advantage here.

Some SQL databases can also be excluded from a more specific comparison. SQLite is a very lightweight database that is mainly used in embedded systems and mobile applications [Bho15]. SQLite is an embedded database that does not run as a standalone process but is deployed within an application [All10]. Due to the goal of being a lightweight database, SQLite does not offer the usual scope of other SQL databases such as stored procedures. For these reasons, SQLite is also not the best choice for a database for big data that is to be provided as a separate service. Greenplum is used much less than PostgreSQL and MariaDB/MySQL [DE24] and therefore has a correspondingly smaller community. This offers significant disadvantages in terms of security and further development of the software.

Compared to PostgreSQL and MariaDB/MySQL, the former database is more performant with large amounts of data and more complex SQL queries. PostgreSQL can also be a very good data warehouse for executing complex report queries for large amounts of data [EDB24].

Apache Hadoop, unlike PostgreSQL, is not a database system, but a framework that was developed for processing big data (TB, PB). The data is not stored in an SQL database, but in the distributed file system HDFS, which can also store unstructured data. PostgreSQL, on the other hand, is a database system that is used in many areas to store large amounts of data. It is also used, for example, for data storage (MB, GB) on platforms with several million users per day. It offers a very wide range of functions and is easily scalable. Since the installation and operation of PostgreSQL is much simpler and since the available data of the municipal utilities is well-structured and not so extensive, the use of PostgreSQL is best suited as a database management system for the Core Database.

6.4 Semantic Layer

6.4.1 Software Selection

The selection of suitable open source software for the semantic layer was based on the software list of the topic 'semantic-layer' on GitHub [Git24a], which is a developer platfrom to manage code.

The size of the community of the individual software is important for the software selection, as a larger community is an indicator that the software is better maintained. The community size can be recognized in GitHub via the stars, as a user gives a star to a software if (s)he follows it. Since the third software in this list already falls below 500 stars when sorting by the most stars, only the first three were shortlisted.

The first software is *cubejs* with approx. 17,500 stars. cubejs is a semantic layer that was developed for connecting to SQL-capable data sources such as Postgres, Google BigQuery and many more [Cub24]. cubejs offers the possibility to create and change data models and views both statically and dynamically and thus to create any view of the connected data source.

The second software is *Metricflow* from the company dbt. Metricflow is a Python library for the implementation of a semantic layer and is not a complete open source software. Dbt also offers a complete software as a semantic layer, but this is not open source and is therefore not considered further in this selection [Dbt24].

The last software in the list is *synmetrix*. With 491 stars, this is already well behind cubejs in terms of the size of the community. Since Synmetrix itself uses cubejs for flexible data modeling, using this software instead of cubejs offers no advantages in terms of data modeling [Syn24]. And since a separate user interface is developed with the low-code platform, the use of this software can also be ruled out.

The open source software cubejs is used here to implement the semantic layer. The following section describes how the dynamic data modeling was implemented with the help of cubejs.

6.4.2 Dynamic Modeling

Data models

In cubejs, data models based on tables in an SQL database are referred to as *cubes*. Such cubes can be created statically or dynamically. For static creation, a YAML file

must be created for each required cube and stored in the correct file path within cubejs. From these YAML files, cubejs then creates and compiles the data models, which can then be queried via the API.

Before the implementation of dynamic data modeling with the help of cubejs is considered, the structure of a YAML file for the creation of static cubes is first considered in order to gain a better understanding of the cubes.

Listing 6.1: Creating a Cube

```
cubes:
  - name: customer
    sql_table: customer
```

To create a cube, it requires a name that can be freely chosen and the name of the referenced SQL table. These can be specified in the YAML file as shown in Listing 6.2.

However, simply creating a cube does not make it possible to query the data. To be able to access the data in the created table, a so-called *dimension* must be added to the cube for each column of the SQL table that is needed. Again, it is necessary to specify a name that can be freely chosen. A dimension can be of four different data types. These are: *time*, *string*, *number*, *boolean* and *geo*. Depending on the selected data type, the name of the SQL column can simply be specified in the *sql* field or, when using the string type, a concatenation of other text columns can also be specified. There are also other options, such as specifying whether the dimension is a primary key, etc.,

Listing 6.2: Adding Columns

```
cubes:
...
  dimensions:
    - name: street
      sql: street
      type: string
```

In addition to the dimensions, a cube can also contain one or more measures. These also consist of a name and a type. These types are either an aggregation function from SQL such as count, max etc. or a number or string. For the former, the column of the table on which the function is executed is specified in the *sql* field, whereas for the latter two formulas or entire SQL statements can be specified. The specified formulas or SQL statements must then result in either a string or a number, depending on the selected type. This makes it possible to add measures that calculate a new value from two numbers, or a switch case to display different strings under certain conditions.

Listing 6.3: Adding Measures

```
cubes:
...
  measures:
    - name: count
      sql: id
      type: count
```

Finally, cubes can have joins, which are relationships to other cubes. These relationships are specified directly in the cube for the reason that cubejs only implements the LEFT JOIN, so the cube containing the join is the left set and the cube specified in it is the right set. If a RIGHT JOIN is to be implemented, the JOIN can simply be specified in the second cube. Cube does not offer the implementation of INNER and OUTER JOINs directly when defining the join, but via small detours. For example, filters that filter out NULL values can be specified for an INNER JOIN. To achieve an OUTER JOIN, this must be specified directly as an SQL statement in a cube.

Listing 6.4: Adding Joins

```
cubes:
...
  joins:
    - name: line_items
      sql: "{CUBE}.id = {line_items.order_id}"
      relationship: many_to_one
```

A condition must also be specified in the *sql* field; this is similar to the condition from the on clause of a SQL join statement. The cardinality of the relation is also specified. There are the relationships one-to-one, one-to-many or many-to-one.

Views

Alongside the definition of cubes with their dimensions, measures and joins, several cubes can also be combined into a new view in the YAML file. To do this, a view and all other components can be given a name under which the content of the view can be queried. In addition, several cubes can be specified. These are not specified by entering the name of the cubes but by means of a join path. The view is thus based on a cube whose join path is then simply the name of the cube. All further cubes are then specified by the path from the base cube to the desired one.

The example in Listing 6.5 shows how a view of the two cubes *base_cube* and *second_cube* is created. Since the *base_cube* has a join to *inter_cube* and the *inter_cube* has a join

to *second_cube*, the path must be specified accordingly in the order so that the view can be defined.

Listing 6.5: Adding Cubes to View

```
views:
  - name: new_view

  cubes:
    - join_path: base_cube

  cubes:
    - join_path: base_cube.inter_cube.second_cube
```

The example in Listing 6.5 shows how a view of the two cubes *base_cube* and *second_cube* is created. Since the *base_cube* has a join to *inter_cube* and the *inter_cube* has a join to *second_cube*, the path must be specified accordingly in the order so that the view can be defined.

When defining a view, the dimensions and measures of the individual cubes to be included in the view can also be specified in addition to the join path. These can be specified under the include field.

Listing 6.6: Adding Dimensions to View

```
views:
  ...
  cubes:
    - join_path: base_cube
      includes:
        - id
        - name
```

Dynamic data models

By creating the YAML files described above, the data models and views can be created statically. In order to fulfill the requirement **U7**, these must be created dynamically. Although cubejs does not provide the ability to create and model data models via an interface, it is possible to load data via a Python script and dynamically create these YAML files using the *Jinja* template engine. This offers the possibility to write python like code by using special placeholders to create documents. In this case, these are YAML files.

Cube provides a python package from which the class *TemplateContext* can be imported. This offers the option of registering a function using the function *add_function* or the

decorator *template.function* so that it can be called from a Jinja template. As can be seen in Listing 6.7, a function is implemented that loads the data from the database specified in Section 6.4.2 via a http client so that this data is available in the Jinja template.

Listing 6.7: Load Data for Dynamic Cube Creation

```
from cube import TemplateContext

template = TemplateContext()
client = ApiClient()

@template.function('load_data')
def load_data():
  return client.load_data()
```

Instead of creating a YAML file for each cube, a YAML file with the name *cubes* can now be created, which then renders the various cube files as in Listing 6.8. Thanks to the Jinja template engine. To do this, the data that is available by registering the *load_data* function in the template is first stored in the variable *data* in order to then create a cube for all cubes in data using a for loop.

Listing 6.8: Render YAML File via Jinja Template Engine

```
{%- set data = load_data()%}
cubes:
  {%- for cube in data["cubes"] %}
  - name: {{ cube.name }}
```

By loading the data, it is now possible to create the data models dynamically, but there is still a problem with the execution of the Python script. Cube always executes it when the entire data model is recompiled. To trigger this process, it is possible to update the schema version. To do this, Cube offers the option of implementing a function in a JavaScript file that returns the schema version. This function is executed with every request to Cube and if the returned schema version does not match the previous version, the data model is recompiled. In order to recompile the data model when changes are made via the API, the schema version must also be updated with each change. For this reason, the schema version is also saved in the service database.

Listing 6.9 shows an example of how the function is implemented in the file *cube.js*. The function *queryVersion* makes a query to the database containing the schema version. As this is updated with every change to the data models. The next time Cubejs is queried, the latest data model is available to the user.

Listing 6.9: Implement Schema Version for Cube

```
module.exports = {
  schemaVersion: () => {
    try {
      return await queryVersion();
    } catch (e) {
      throw Error(`Could not get schema version: ${e}`)
    }
  }
};
```

Database

Since cubejs itself does not offer the possibility to create or customize data models via an API, but only to create them dynamically by loading data, it requires an additional database in addition to the cubejs internal database, which provides the information on the data models and makes it queryable for cubejs.

When developing the schema of this database, the structure of the data models of cubejs could simply be adopted, but it requires an adapted structure to support graphical modeling via the user interface. For this reason, this section deals with the structure of the schema, which is referenced again in Section 6.6.2 when implementing the graphical data modeling.

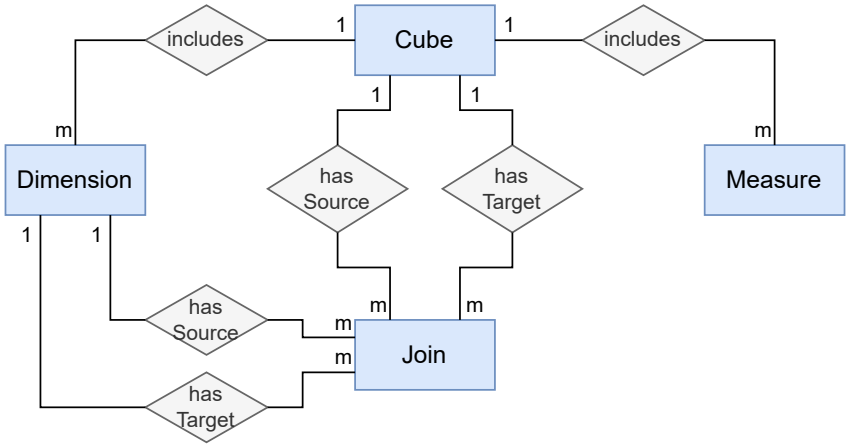

Figure 6.2: Database Schema for Storing Dynamic Data Models

As can be seen in Figure 6.2, a cube can have several dimensions and measures. A join, on the other hand, always refers to exactly one source cube and one target cube, as well as a dimension of the source cube, which is the first part of the join condition, and a dimension of the target cube, which is the second part of the join condition.

Besides the schema for the cubes, it also requires one for storing the views created by the user. This database schema is similar to the structure of the views in the YAML file, as a less complex interface for creating views was designed here as described in Section 5.3.2 and therefore no additional information and relationships are required.

In order for the Model Synchronizer and the user to be able to create data models, an interface to the database is required. This is not provided by the service itself but by the API gateway whose design was presented in Section 5.2.6.

6.5 Model Synchronizer

6.5.1 Database Changes

When integrating data from the ETL tool, new tables are created in the Core Database. A data model must now be created for these in cubejs as described in the previous section. To avoid the user having to do this via the interface, the Model Synchronizer was introduced in Chapter 5 as a concept that receives data changes via triggers implemented in the Core Database and transfoms them to data models of the database schema of the semantic layer. Since the transfer of changes from the Core Database to the Model Synchronizer depends on the choice of software used and is therefore not specified in more detail in the concept, the communication used in the implementation of the Model Synchronizer is examined in more detail here.

For the communication between the Core Database and the Model Synchronizer there are two approaches. The first is that functions written in PostgreSQL can request a REST-endpoint implemented in the Model Synchronizer via HTTP with the changed data. The second is to use a message broker through which the Core Database can publish changes that the Model Synchronizer then consumes. By using PostgreSQL as the database management system, the second option is easier to implement, as PostgreSQL provides built-in functions for communication between postgresql clients via channel using the NOTIFY/LISTEN mechanism. It would also be possible to write PostgreSQL functions using for example the PL/Python procedural language to request REST-endpoints, but this would require implementing an endpoint in the Model Synchronizer and the request in the Core Database, which would result in significantly more development effort than communication using the NOTIFY/LISTEN mechanism.

6.5.2 Realization

The three components of the Model Synchronizer designed in Section 5.2.5 were all developed in the Typescript programming language, as this was the preferred programming language of the developers. To connect the *DB Listener* to the PostgreSQL message channel, it requires the implementation of an SQL client, which then connects to the PostgreSQL instance. For Typescript there is the library *pg*, with the help of which an SQL client can be created, which is done in Listing 6.10 in method *getSQLClient*. Once this client has been created, an SQL statement can be executed on the connected database using the *query* function. In this case, LISTEN is used to listen to the channel *syncdatamodel*.

Listing 6.10: Consuming Messages from the Postgres Channel

```
const client = getSQLClient();

await client.connect();

client.query("LISTEN syncdatamodel")
```

To finally process the messages that are published on the specified channel, the *on* function can be called, which uses the *notification* key and a function that is always executed when a message is consumed. As can be seen in Listing 6.11, the message content is first parsed into the correct type script object. It is then transformed into the structure of the cubejs API and finally created using an API request.

Listing 6.11: Execution of the Data Model Transformation

```
client.on("notification", async (msg) => {
  const dataModel = parseDatamodel(msg.paylaod);
  const transformedDataModel = transformDataModel(dataModel);
  await createCubeWithDimension(transformedDataModel);
})
```

6.6 Graphical User Interface

To implement the graphical user interface, the decision was made to implement a web application using the *React* framework in the programming language *Typescript*. React is the second most widely used framework for the development of web applications [Ove23]. It follows the concept of developing individual reusable components for more efficient

development. There is also a large number of libraries that facilitate the development of components in React. Typescript is a programming language based on JavaScript. Typescript extends JavaScript with additional language features, such as a static type checker or interfaces, which are explained in more detail in the following Section [Har20]. The choice of technology was made with regard to the skills and preferences of the developers.

6.6.1 Data Integration

The challenge in implementing the Data Integration service is the dynamic loading of configuration forms that depend on the type of source system as well as an easy extensibility of new source systems. If, for example, the energy supplier wants to integrate a PostreSQL database, it should be easy for a developer to develop the code for the configuration via the frontend. Since Airbyte provides almost 100 source connectors [Air24c], it is too much effort and unnecessary to provide all of them via the user interface, but rather on demand. Although Airbyte also offers its own interface, the system is designed to be available as a standardized interface for all functions and not as a system consisting of several user interfaces. In addition, functions and settings offered by Airbyte can be abstracted away, such as the definition of multiple target systems. As there is only one target system in the developed system, these settings can be configured once during deployment and configuration is made one step easier for the user by eliminating this option.

To create an environment for developers that makes it easier to provide new connectors in the user interface, it is first necessary to see what options Airbyte provides to define connections to source systems. As mentioned in Section 6.2.2, Airbyte also offers a REST API with the *Airbyte API* for configuring new source systems, so-called *sources*. New sources are created using a POST request via the */v1/sources* endpoint [Air24b]. A user-defined name, the ID of the Airbyte connector to be used and a JSON object of the connector-specific configuration can then be specified in its request body.

As the user should not be dependent on the structure of the request body when designing the input form, the challenge when designing the code is to give the developer the possibility to freely develop the input form and add any possible airbyte source without re-implementing the actual request.

The basis for meeting these challenges in Typescript are generic types and interfaces. With interfaces, typescript offers the possibility of defining structures — a kind of blueprint — of objects. Thanks to the interfaces, the structure of an input field can be defined, which the developer then only has to implement. Because of the predefined

structure, however, the developer does not have to worry about rendering the fields, as this is done independently of the implementation by the predefined structure.

Listing 6.12: Example of Rendering an Input Field

```
interface InputForm {
  component:
  props: {
    title: string;
    defaultValue: string;
  }
}

export const ParentNode: React.FC = (props) => {
  ...

  <input.component title={props.title} value={props.defaultValue} />
}
```

Listing 6.12 shows an excerpt from the interface of the input field and how it is rendered independently of an actual object.

Apart from interfaces, generic types are used. Generic types are placeholders that are used to enforce a type restriction at certain points [Che19]. For example, generic types can be used in the definition of interfaces in order to have a placeholder for the type of attribute. The actual type is then only specified when an object is created by the interface.

Listing 6.13: Example for the Use of Generic Types

```
interface ActuelType {
  attribute: string
}

interface Generic<T> {
  uncertainType: T
}

const obj: Generic<ActuelType> = {
  attribute: "value"
}
```

For a better understanding, two interfaces are defined in Listing 6.13, the second of which has an attribute with a generic type. When creating an object with the second type, the first type is then passed in angle brackets, so that the typecasting knows which attributes the object must contain.

With this knowledge, an interface can now be created on the basis of which the rendering of the input fields and the request to Airbyte can be implemented once, so that the developer who wants to add another source only has to create an object of the interface and does not have to worry about any further implementation.

The interface, which can be seen in Listing 6.14, initially consists of the *sourceId* and the *name*, both must be included in the request body to Airbyte. It also contains an *icon*, which is displayed on the interface for better user experience, and a list of inputs, which the developer can use to create any input form, as shown in Listing 6.12. Finally, there are two functions that need to be implemented. Since the input form can be freely designed and therefore does not correspond to the structure of the request body of the configuration field of the Airbyte API, the values in the input form must be mapped to the structure of the request body. This is where the generic types come up. The function receives a list of all values of the input fields and returns the generic type *SourceData*. *i*s the interface that corresponds to the structure of the configuration field in the request body of the Airbyte API. Since this differs depending on the source, as already mentioned, the developer must first create this interface and can then pass it as a type when creating the object from the interface *CreateSource*. As configured connectors can also be edited via the user interface, the function must exist in reverse order so that the queried configuration from airbyte can be mapped into the structure of the input fields.

Listing 6.14: Interface for Creating New Sources

```
export interface CreateSource<SourceData> {
  sourceId: string;
  name: string;
  icon: React.FC<SvgIconProps>;
  inputs: InputForm[];
  mapping?: (obj: any[]) => SourceData;
  reverseMapping: (source: SourceData) => any[];
}
```

If a developer now wants to add PostgreSQL as a new possible source system, for example, (s)he must check in the Airbyte documentation for the structure of the request body and create an equivalent interface. Then the developer can create a list of objects of the interface *InputForm* to define the input form and finally, the developer creates an object of the interface *CreateSource* which is then used to render the input form and make the requests to Airbyte. The effort is therefore manageable, and the system can be easily expanded to include additional sources.

6.6.2 Data Modeling

Software Selection

As described in the concept, interactive data modeling is implemented using a flowchart. Following the approach of using open source software to fulfill the requirements **B1** and **B2**, an open source React library must first be selected to implement the flowchart.

To get a selection of possible libraries, a search was conducted on the developer platform GitHub. The search resulted in a selection of three flowchart libraries that offer the required scope and are written in Typescript for React. To make a selection, a table (see table 6.1) was created that lists the community size (stars), the last activity (last commit) and the number of contributors for all libraries.

Name	**Stars**	**Last Commit**	**Contributors**
React Flow Chart	1,500	28-06-2020	19
REAFLOW	2000	02-08-2024	26
React Flow	23,100	31-07-2024	91

Table 6.1: Overview of React Flowchart Libraries (Accessed: 04-08-2024)

React Flow Chart has the smallest community in comparison and the last activity was four years ago. As a result, this library should not be used, as the further development and security of the software cannot be guaranteed. *REAFLOW*, on the other hand, has an active community, but with 2,000 stars it is significantly smaller than *React Flow*. Also, while both libraries provide the desired functionality, React Flow has significantly more features. For this reason, the decision was made to use React Flow for the implementation.

Realization

The concept for connecting data models described in Section 5.3.2 shows a flowchart with nested nodes. The representation of the nodes is different, as the inner nodes, which represent the attributes, have a connector at both ends for connecting attributes. To display nodes, React Flow offers the option of creating custom nodes. For this reason, two types of nodes have been implemented: A *DataModelNode* and a *AttributeNode*.

To add a custom node, a component must first be created in React, which receives all the node's data as properties (see Listing 6.15). The node can then be designed within the HTML div container. If the node has a title, the title can be displayed in the node using the data received.

Listing 6.15: Creating a Custom Node

```
export const AttributeNode: React.FC<NodeProps> = ({ data }) => {
  return (
    <div>
      {data.title}
      ...
    </div>
  )
}
```

Unlike the *DataModelNode*, the *AttributeNode* has a connector at each end. To display these, React Flow provides the component *Handle*. This can then be implemented within a node by passing properties such as position, type and ID to the component. An example can be seen in Listing 6.16.

Listing 6.16: Add Connectors to Custom Node

```
export const AttributeNode: React.FC<NodeProps> = ({ data }) => {
  return (
    <div>
      <Handle
        type="target"
        position={Position.Left}
        id="handle-left"
      />
      ...
      <Handle
        type="source"
        position={Position.Right}
        id="handle-right"
      />
    </div>
  )
}
```

To render the nodes in the flowchart of React Flow, an object of type *Node* must be created for each node that is to be rendered and passed to the flowchart component as a list. Such an object has a unique ID, a type (name of the custom node specified), a position and the required data that is passed to the node for rendering. This list of nodes can then be passed to the component *ReactFlow* (see Figure 6.18), which then renders the flow diagram with the nodes at their specified position. To display the nodes in a nested manner, a *parentId* can also be specified for a node in addition to the properties mentioned, which then corresponds to the ID of the associated *DataModelNode*.

Similar to the creation of nodes, edges can also be created by creating a list of objects of the type *Edge*. An edge also contains a unique ID as well as the ID of the source

node under the attribute *source* and the ID of the target node under *target*. This list is also passed to the *ReactFlow* component (see Figure 6.18) and thus the edges between the nodes are rendered.

Since the data models are stored in the database of the semantic layer as shown in Figure 6.2, it is now possible to query the data via the API layer and create the lists of nodes and edges from it. Each entry in the table *Cube* becomes a *DataModelNode* and for each associated dimension a *AttributeNode* with the ID of the cube as *parentId*. The list of edges is created from the entries in the *Join* table. As a join always has exactly one source and one target dimension, the object of an edge can be created easily. Listing 6.17 shows how the required nodes for the flow diagram components are created from the loaded cubes from the semantic layer.

Listing 6.17: Create Node Objects of Cubes

```
const nodes = cubes.forEach((cube) => {
  id: cube.id,
  data: { label: cube.title },
  position: getPosition(cube.id),
  ...
})
```

To create new joins between data, a function can be passed to the *ReactFlow* component, which is always called as soon as a connection is created between two nodes. This function receives the ID of the source as well as that of the target as parameters, whereby a new entry can then be created in the *Join* table with the appropriate dimensions via the API layer in the database (see Listing 6.18).

Listing 6.18: Create New Join via Flow Chart

```
...
const onConnect = (conn: Connection) => {
    createJoin(conn.sourceId, conn.targetId);
);

return (
  <ReactFlow
    nodes={nodes}
    edges={edges}
    onConnect={onConnect}
    ...
  />
)
```

7 Evaluation

In order to fulfill the objective of building an individual, easy-to-use platform for versatile data integration for a company with challenging requirements, the concept developed in Chapter 5 is evaluated as proposed by Kazman et al. [Kaz96] based on scenarios.

For this purpose, the first step is to develop scenarios for both the user and the developer that describe the expected use of the designed system from the perspective of the developer and the end user. In the second step, the scenario evaluations are performed. For this purpose, the scenarios are classified as direct or indirect. Direct means that the system fulfills the described scenario, whereas indirect means that an adaptation of the concept is required to execute the desired scenario. If a scenario is classified as indirect, the required changes that need to be made to the components of the concept are also noted. The so-called *scenario interactions* are then identified in the third step. A scenario interaction exists when two scenarios classified as indirect require changes to the same component. Finally, the system is evaluated on the basis of the scenarios classified as indirect and the identified scenario interactions.

7.1 Develop Scenarios

The scenarios were developed by an employee of the energy supplier. The roles considered were that of a user who uses the graphical user interface and a developer of the system.

User:

1. *Add a new source system.* Add a new source system that is based on an existing connector. For example, a new PostgreSQL database is to be added.

2. *Add a new source system type.* Add a new source system from a source system type, which is not displayed in the selection of source system types.

3. *One-click data importing.* Triggering the ETL process by clicking on a button.

4. *Periodic import of data.* Configuring regular automated data imports from the source systems.

5. *Transformation of imported data.* Execution of all required transformations during the ETL process.

6. *Plausibility of data.* Execute all required plausibility checks of incorrect data.

7. *Selection of the data to be imported.* Selection of the data records to be imported and the individual fields contained in the data record.

8. *Adding and editing data.* Manually add data in addition to the imported data and edit this and the imported data.

9. *Creating a business view of the data.* Transferring the imported data from the technical view to the business view.

10. *Create relationships between different data records.* Create relationships between data from the same and different sources.

11. *Configuration of the extraction mechanism.* Specification of the extraction mechanism, whether all data should be overwritten on re-import or only appended.

12. *Creation of aggragations on data sets.* Possibility to apply aggregation functions to the imported data records.

Developer:

13. *Adding a new source connector.* Possibility to easily add new required connectors.

14. *Use different database technology.* Possibility to change the database technology if, for example, the data storage requirements change.

15. *Use different semantic layer.* If the semantic layer no longer meets all the desired requirements, it should be possible to replace it.

16. *Provide custom plausibility algorithm.* Extension of the system with all necessary plausibility algorithms for the plausibility check

17. *Provide custom transform algorithm.* Extension of the system with all necessary transformation for ETL-Prozess.

18. *Replacing the ETL tool.* The ETL tool used should be replaceable.

19. *Restrict user access to the services.* Ability to provide an endpoint that only allows access to the functionalities of the services that the user should access.

7.2 Perform Scenario Evaluations

When classifying the scenarios, 5 out of 19 were classified as indirect. In other words, 74% of the scenarios can be implemented directly by a user or developer. The remaining 26% require changes to the concept. The changes are explained in more detail in the table 7.1.

Scenario	Description	Changes
2	Add a new source system type	To add a new source type as a user, the **Graphical User Interface** must be modified, and the **Data Integration** service must also be modified.
5	Transformation of imported data	If the ETL tool does not provide the required transformation, both the **Graphical User Interface** and the **Data Integration** must be changed.
6	Plausibility of data	Similar to a user's restrictions on transformations, if the desired algorithm is not provided, changes must be made to the **plausibility** service and the **Graphical User Interface**.
14	Use different database technology	When changing the database technology, changes will have to be made to the **Model Synchronizer** services, since they were designed specifically to map the data structures between the database and the semantic layer.
15	Use different semantic layer	The changes to the **Model Synchronizer** must be made for the same reason when replacing the semantic layer as when changing the database technology.

Table 7.1: Scenario Evaluations

7.3 Reveal Scenario Interactions

The evaluation of the scenarios results in the number of interactions per component shown in table 7.2.

Component	number of changes
Graphical User Interface	2
Data Integration	1
Model Synchronizer	1

Table 7.2: Scenario Interactions per Component

Three of the seven components proposed in the concept have interactions. The higher the interactions, the worse the functionality of the component is isolated and the more likely the component will lead to problems in the end product [Kaz96]. By specifying the interactions, the components that require particular attention in the overall evaluation stand out.

7.4 Overall Evaluation

Through the evaluation of the scenarios performed in Section 7.2 and the following specified scenario interactions, the developed concept can be evaluated in the overall evaluation. The developed concepts can perform the majority of the set up scenarios without fundamental changes to the concept. However, when calculating the scenario interactions, three components of the concept were identified that require conceptual changes so that all scenarios can be fulfilled.

Taking a closer look at the scenario interactions, it can be seen that the graphical user interface has two interactions, whereas the remaining two components have only one interaction, which in turn indicates a lower risk of the component causing problems in the final system.

The graphical user interface shows restrictions regarding scenarios that are to be executed by a user. The first is that a user cannot use the interface to specify a new source of a source type that has not already been implemented in the user interface. As the ETL tool should be selected so that it offers connectors to all common data sources, all of these should also be available in the user interface. This means that there are only limitations in cases where very specific source systems are to be connected. The last two limitations are revealed in the transformation and plausibility check of imported data. Here, the user only has the option of selecting already implemented transformations and plausibility checks; if new algorithms are required, the user interface must be adapted so that these can also be configured graphically.

The limitations of the Data Integration service result from the adding of new sources with non-implemented source types and the need for non-implemented transformation algorithms. The former is a limitation that cannot be eliminated, as it is hardly possible to implement all types of source systems and their protocols. In addition, new protocols and source systems may be developed in the future, in which case the Data Integration inevitably has a limitation and must be extended. To counteract the second limitation, it should be possible to provide all possible transformations in the Data Integration service, but since the transformations are strongly dependent on the domain and the data, this is also a limitation that must be dealt with.

The last component that has limitations with regard to the developed scenarios is the model synchronizer. As this accepts data from the Core Database, transforms it and writes it to the semantic layer, it must be adapted both when changing the database technology and when changing the semantic layer, as the mappings between the data structures depend on these. Similar to the restriction of the source system types of an ETL tool, it is not possible to implement all possible mappings for current and future database technologies and semantic layers.

8 Conclusion

8.1 Discussion

In order to answer the research question posed at the beginning, requirements were first derived from related literature and from a specification sheet provided by an energy supplier. Based on these requirements, the approach chosen in the concept was to develop a system based on open source components in order to implement the concept in a cost-effective and time-saving manner. The microservice architecture style was chosen as the basis for the software architecture style, whereby the event-driven architecture style was used in some cases for communication within and between the services. The system was divided into six services with a view to the loosest possible coupling between the services and the possibility of implementing the services using open source software. To provide the easy-to-use platform required by the research question, a graphical user interface was designed as a low-code platform. Since the system was implemented for the energy supplier as part of this work, parts of the implementation and the selection of open source tools were presented after the conceptualization. Finally, the created concept was evaluated based on scenarios to show that the research question can be answered with this concept.

As the evaluation reveals, the concept fulfills the set scenarios with the exception of a few limitations (explained in more detail in the following Section). The system makes it possible to use the easy-to-use low-code platform to combine and model data from various sources as a user who is not a developer. In addition, the structure offers the possibility to implement parts using open source tools, which meets the business requirements **B1** and **B2**. Since the system was not only developed on the basis of the requirements of a company, but systems from other domains were also considered as well as the general requirements for a data integration platform, the research question can be answered with the concept not only for the energy sector, but also for other areas such as medicine or digital urban planning. By fulfilling the requirement **S2** and the approach of implementing shared services itself, the challenging requirements of each company can be addressed when implementing the concept, which means that the concept also answers the last part of the research question.

8.2 Limitations and Future Research

As already discussed in Section 7.4, the concept has limitations with regard to the three components *Graphical User Interface*, *Data Integration* and *Model Synchronizer*. The greatest limitations of the concept are due to the *Graphical User Interface*. This is limited in that it is not possible for users to add new source system types, define new transformations and new plausibilitiy checks via the low-code platform without development effort. In all three cases, the concept of the graphical user interface could be extended in such a way that it is possible to eliminate these current limitations via the interface in the low-code approach. With regard to the creation of new source system types, the system only needs to make it possible for the user to define the settings via the interface that are currently still defined by the implementation of the generic interface described in Section 6.6.1. Further research is required into the conception of a low-code interface for the creation of new transformations and plausibility checks, as this implementation is far more complicated. In future work, a solution can be found to define transformation and plausibilization algorithms as well as possible without developing actual code.

Bibliography

[Air24a] Airbyte: Architecture overview (2024), URL https://docs.airbyte.com/understanding-airbyte/high-level-view, accessed: 19-07-2024

[Air24b] Airbyte: Create a source (2024), URL https://reference.airbyte.com/reference/createsource, accessed: 10-08-2024

[Air24c] Airbyte: Hundreds of connectorsout-of-the-box (2024), URL https://airbyte.com/connectors?connector-type=Sources, accessed: 21-07-2024

[All10] Allen, Grant: *The Definitive Guide to SQLite*, Apress L. P., Berkeley, CA, 2nd edn. (2010), description based on publisher supplied metadata and other sources.

[Alo17] Alotaibi, Saad B.: ETDC: An Efficient Technique to Cleanse Data in the Data Warehouse, ETL, in: *Proceedings of the International Conference on Advances in Image Processing*, ICAIP 2017, ACM

[AtS24] AtScale: What is a Semantic Layer? (2024), URL https://www.atscale.com/glossary/semantic-layer/, accessed: 15-07-2024

[Bau13] Bauer, Andreas: *Data-Warehouse-Systeme*, dpunkt.verlag, Heidelberg, 4th edn. (2013)

[Bet22] BetterCloud: Average number of software as a service (SaaS) applications used by organizations worldwide from 2015 to 2022 (2022), URL https://www.statista.com/statistics/1233538/average-number-saas-apps-yearly/, accessed: 01-08-2024

[Bho15] Bhosale, Satish Tanaji; Patil, Tejaswini and Patil, Pooja Patil: SQLite: Light Database System. *International Journal of Computer Science and Mobile Computing* (2015), vol. 4(4):pp. 882 – 885

[Boc21] Bock, Alexander C. and Frank, Ulrich: Low-Code Platform. *Business Information Systems Engineering* (2021), vol. 63(6):pp. 733–740

[Bru10] Bruns, Ralf and Dunkel, Jürgen: *Event-Driven Architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse, Microservices*, Springer Berlin Heidelberg (2010)

[BSI24] BSI: Open Source Software und Vorabversionen von Betriebssystemen (2024), URL https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher, accessed: 10-08-2024

[Che19] Cherny, Boris: *Programming TypeScript*, O'Reilly, Beijing, 1st edn. (2019)

[Che22] Chen, Lili; Di, Yingqi and Zhang, Lele: Implementation of Cloud-based Urban Rail Big Data Platform, in: *The 6th International Conference on Computer Science and Application Engineering*, CSAE 2022, ACM

[Cub24] Cube Dev, Inc.: Cube Website (2024), URL https://cube.dev/, accessed: 09-07-2024

[Dat23] Datanyze: Leading containerization technologies market share worldwide in 2023. *Statista* (2023), URL https://www.statista.com/statistics/1256245/containerization-technologies-software-market-share/

[Dav16] Davidson, Louis and Moss, Jessica: *Pro SQL Server Relational Database Design and Implementation*, Apress (2016)

[Dbt24] Dbt: About MetricFlow (2024), URL https://docs.getdbt.com/docs/build/about-metricflow, accessed: 09-07-2024

[DE24] DB-Engines: Vergleich der Systemeigenschaften Greenplum vs. PostgreSQL (2024), URL https://db-engines.com/de/system/Greenplum;PostgreSQL, accessed: 19-07-2024

[Des24] Destatis, Statistisches Bundesamt: Statistischer Bericht - Daten zur Energiepreisentwicklung - Januar 2024 (2024), URL https://www.destatis.de/DE/Themen/Wirtschaft/Preise/Publikationen/Energiepreise/statistischer-bericht-energiepreisentwicklung-5619001241015.html, accessed: 01-08-2024

[Doc24] Docker Inc: Docker Compose Website (2024), URL https://docs.docker.com/compose/, accessed: 05-06-2024

[EDB24] EDB: PostgreSQL vs. MySQL: A 360-degree Comparison (2024), URL https://www.enterprisedb.com/blog/postgresql-vs-mysql-360-degree-comparison-syntax-performance-scalability-and-features, accessed: 07-08-2024

[Fie00] Fielding, Roy Thomas: *Architectural styles and the design of network-based software architectures*, Ph.D. thesis (2000)

[Git24a] Github: semantic-layer (2024), URL https://github.com/topics/semantic-layer, accessed: 21-07-2024

[Git24b] Github: Understanding GitHub Actions (2024), URL https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions, accessed: 15-07-2024

[Git24c] GitLab: CI/CD development guidelines (2024), URL https://docs.gitlab.com/ee/development/cicd/, accessed: 15-07-2024

[Har19] Harris, Austin and Sartipi, Mina: Data integration platform for smart and connected cities, in: *Proceedings of the Fourth Workshop on International Science of Smart City Operations and Platforms Engineering*, CPS-IoT Week '19, ACM

[Har20] Hartmann, Nils: *React*, dpunkt.verlag, Heidelberg, 2nd edn. (2020)

[Hel09] Helmis, Steven and Hollmann, Robert (Editors): *Webbasierte Datenintegration*, SpringerLink, Vieweg+Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden, Wiesbaden (2009), URL https://link.springer.com/content/pdf/10.1007/978-3-8348-9280-5.pdf

[Hlu21] Hlupic, Tomislav and Punis, Josip: An Overview of Current Trends in Data Ingestion and Integration, in: *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, IEEE

[HM20] Hossein Motlagh, Naser; Mohammadrezaei, Mahsa; Hunt, Julian and Zakeri, Behnam: Internet of Things (IoT) and the Energy Sector. *Energies* (2020), vol. 13(2):p. 494

[Hos24] Hoseini, Sayed; Theissen-Lipp, Johannes and Quix, Christoph: A survey on semantic data management as intersection of ontology-based data access, semantic modeling and data lakes. *Journal of Web Semantics* (2024), vol. 81

[Hua23] Huawei Technologies Co., Ltd.: *Cloud Computing Technology*, Springer, Singapore, 1st edn. (2023)

[Idc21] Idc and Statista: Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025 (in zettabytes) [Graph]. In Statista (2021), URL https://www.statista.com/statistics/871513/worldwide-data-created/

[Ihi20] Ihirwe, Felicien; Di Ruscio, Davide; Mazzini, Silvia; Pierini, Pierluigi and Pierantonio, Alfonso: Low-code engineering for internet of things: a state of research, in: *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '20, ACM

[Inm96] Inmon, William H.: *Building the data warehouse*, Wiley computer publishing, Wiley, New York, NY, 2nd edn. (1996)

[Jay19] Jayaratne, Madhura; Nallaperuma, Dinithi; De Silva, Daswin; Alahakoon, Damminda; Devitt, Brian; Webster, Kate E. and Chilamkurti, Naveen: A data integration platform for patient-centered e-healthcare and clinical decision support. *Future Generation Computer Systems* (2019), vol. 92:pp. 996–1008

[JRC19] JRC., CEU.: *Web Application Programming Interfaces (APIs): general purpose standards, terms and European Commission initiatives.*, Publications Office (2019)

[Kau23] Kaufmann, Michael and Meier, Andreas: *SQL and NoSQL Databases: Modeling, Languages, Security and Architectures for Big Data Management*, Springer Nature Switzerland (2023)

[Kaz96] Kazman, R.; Abowd, G.; Bass, L. and Clements, P.: Scenario-based analysis of software architecture. *IEEE Software* (1996), vol. 13(6):pp. 47–55

[Kof21] Kofler, Michael: *Docker*, Rheinwerk Computing, Rheinwerk Verlag, Bonn, 3rd edn. (2021)

[Kon18] Konopasek, Klemens: *SQL Server 2017*, Hanser eLibrary, Hanser, Carl, München (2018)

[Kö14] Köppen, Veit; Sattler, Kai-Uwe and Saake, Gunter: *Data Warehouse Technologien.*, Mitp Professional, MITP, 2nd edn. (2014), URL https://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=979062&site=ehost-live

[Las23] Laster, Brent: *Learning GitHub Actions*, O'Reilly, Beijing, 1st edn. (2023)

[Lyu15] Lyu, Dao-Ming; Tian, Yu; Wang, Yu; Tong, Dan-Yang; Yin, Wei-Wei and Li, Jing-Song: Design and Implementation of Clinical Data Integration and Management System Based on Hadoop Platform, in: *2015 7th International Conference on Information Technology in Medicine and Education (ITME)*, IEEE

[Mar82] Martin, James: *Application development without programmers*, Prentice-Hall, Englewood Cliffs, N.J., 5th edn. (1982)

[Mar93] Marsch, Jürgen and Fritze, Jörg: *SQL*, Vieweg+Teubner Verlag (1993)

[Mas24] Masmoudi, Maroua; Ben Abdallah Ben Lamine, Sana; Karray, Mohamed Hedi; Archimede, Bernard and Baazaoui Zghal, Hajer: Semantic Data Integration and Querying: A Survey and Challenges. *ACM Computing Surveys* (2024), vol. 56(8):pp. 1–35

[McD24] McDonald, Andy: Temporal Tables and how to use them in SQL Server (2024), URL https://sqlspreads.com/blog/temporal-tables-in-sql-server/, accessed: 09-07-2024

[Mic23] Microsoft: Temporal tables (2023), URL https://learn.microsoft.com/en-us/sql/relational-databases/tables/temporal-tables?view=sql-server-2017, accessed: 09-07-2024

[Mic24] Microsoft: Das API-Gatewaymuster im Vergleich zur direkten Kommunikation zwischen Client und Microservice (2024), URL https://learn.microsoft.com/de-de/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern, accessed: 16-07-2024

[New15] Newman, Sam: *Microservices*, mitp Verlags, [Germany], 1st edn. (2015), URL https://ebookcentral.proquest.com/lib/thm/detail.action?docID=2089872

[Nie21] Nie, Wenyi; Zhang, Quanjiang; Ouyang, Zhiqiang and Liu, Xingang: Design of big data integration platform based on hybrid hierarchy architecture, in: *2021 IEEE 15th International Conference on Big Data Science and Engineering (BigDataSE)*, IEEE

[Ope24] Open Source Initiative: The Open Source Definition (2024), URL https://opensource.org/osd, accessed: 01-06-2024

[Ove23] Overflow, Stack: Most used web frameworks among developers worldwide, as of 2023. *Statista* (2023)

[Pas14] Pasculescu, Adrian; Schoof, Erwin M.; Creixell, Pau; Zheng, Yong; Olhovsky, Marina; Tian, Ruijun; So, Jonathan; Vanderlaan, Rachel D.; Pawson, Tony; Linding, Rune and Colwill, Karen: CoreFlow: A computational platform for integration, analysis and modeling of complex biological data. *Journal of Proteomics* (2014), vol. 100:pp. 167–173

[Pau21] Paulus, Alexander; Burgdorf, Andreas; Pomp, Andre and Meisen, Tobias: Recent Advances and Future Challenges of Semantic Modeling, in: *2021 IEEE 15th International Conference on Semantic Computing (ICSC)*, IEEE

[Pin23] Pinho, Daniel; Aguiar, Ademar and Amaral, Vasco: What about the usability in low-code platforms? A systematic literature review. *Journal of Computer Languages* (2023), vol. 74:pp. 101–185

[Pos23] Postman, Inc.: Postman 2023 State of the API Report (2023), URL https://www.postman.com/state-of-api/, accessed: 18-07-2024

[Ric15] Richards, Mark: *Software Architecture Patterns*, O'Reilly, 1st edn. (2015)

[Ric20] Richards, Mark: *Fundamentals of software architecture*, O'Reilly, Beijing, 1st edn. (2020), unitary Architecture

[Ros13] Rossak, Ines: *Grundlagen der Datenintegration*, Carl Hanser Verlag GmbH & Co. KG (2013), pp. 16–60

[Sar17] Sarnovsky, M.; Bednar, P. and Smatana, M.: Data integration in scalable data analytics platform for process industries, in: *2017 IEEE 21st International Conference on Intelligent Engineering Systems (INES)*, IEEE

[Sch16] Schnider, Dani: *Data Warehouse Blueprints*, Hanser eLibrary, Carl Hanser Verlag GmbH & Co. KG, München (2016)

[Sch17] Schicker, Edwin: *Datenbanken und SQL*, Springer Fachmedien Wiesbaden (2017)

[Ste24] Stephan, Max: *Building a modular and scalable data-driven analytics platform*, Master's thesis, Technische Hochschule Mittelhessen (University of Applied Science) (2024)

[Sub19] Subramanian, Harihara: *Hands-on RESTful web API design patterns and best practices*, Packt Publishing, Birmingham, UK (2019)

[Syn24] Synmetrix: Synmetrix Website (2024), URL https://synmetrix.org/, accessed: 09-07-2024

[Sá24] Sá, Daniel; Guimarães, Tiago; Abelha, Antonio and Santos, Manuel Filipe: Low Code Approach for Business Analytics. *Procedia Computer Science* (2024), vol. 231:pp. 421–426

[Wey17] Weynand, Christopher: *Improving Hosted Continuous Integration Services*, no. 108 in Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam, Universitätsverlag, Potsdam (2017)

[Wir19] Wirdemann, Ralf: *REST*, Carl Hanser Verlag GmbH & Co. KG (2019), pp. 27–37

[WM15] Winters-Miner, Linda A.; Bolding, Pat; Hill, Thomas; Nisbet, Bob; Goldstein, Mitchell; Hilbe, Joseph M.; Walton, Nephi; Miner, Gary; Jakubowski, Jacek; Kulach, Leslaw and Murawski, Piotr: *Platform for Data Integration and Analysis, and Publishing Medical Knowledge as Done in a Large Hospital*, Elsevier (2015), pp. 1019–1029

[Wol18] Wolff, Eberhard: *Microservices*, dpunkt.verlag, Heidelberg, 2nd edn. (2018)

[Zha18] Zhao, J T; Jing, S Y and Jiang, L Z: Management of API Gateway Based on Micro-service Architecture. *Journal of Physics: Conference Series* (2018), vol. 1087