THM | CAMPUS GIESSEN | MNI
TECHNISCHE HOCHSCHULE MITTELHESSEN
Mathematik, Naturwissenschaften und Informatik

Master Thesis

Building a Modular and Scalable Data-Driven Analytics Platform

Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science (M.Sc.)

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

by

Max Stephan

August 11, 2024

Referee: Prof. Dr. Frank Kammer

Co-Referee: 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 11, 2024 Max Stephan

The increasing amount of data available and collected is leading to an increase in data-driven decision-making. This helps companies to make better business decisions and reduce production and service costs. In order to analyze data, this thesis proposes a concept for an analytics platform based on complex use cases, since existing solutions cannot be used for complex use cases, or can only be used at high cost. The concept was originally developed from an analytics platform for a local energy supplier, but always with a view to creating a generalized concept based on existing systems and findings from the literature. The concept proposes a service-based architecture with ten services to create the analytics platform. This includes an application to allow non-technical users to analyze data and a calculation subsystem to calculate metrics. This thesis also includes descriptions and ideas on how to implement the concept and which existing open source software could be used to realize parts of the concept. Finally, a scenario-based evaluation was carried out, showing the effectiveness of the concept and its limitations. This results in the open questions for future work to create a builder UI or plugin system that allows easy creation of new diagram types for visualization and the ability to monitor values and send notifications for errors etc.

Die zunehmende Menge an verfügbaren und gesammelten Daten führt zu einem Anstieg der datengesteuerten Entscheidungsfindung. Dies hilft Unternehmen, bessere Geschäftsentscheidungen zu treffen und Produktions- und Servicekosten zu senken. Um Daten zu analysieren, schlägt diese Arbeit ein Konzept für eine Analyseplattform vor, die auf komplexen Anwendungsfällen basiert, da bestehende Lösungen für komplexe Anwendungsfälle nicht oder nur zu hohen Kosten eingesetzt werden können. Das Konzept wurde ursprünglich aus einer Analyseplattform für einen lokalen Energieversorger entwickelt, jedoch immer mit dem Ziel, ein verallgemeinertes Konzept auf Basis bestehender Systeme und Erkenntnisse aus der Literatur zu erstellen. Das Konzept schlägt eine serviceorientierte Architektur mit zehn Diensten vor, um die Analyseplattform zu realisieren. Dazu gehören eine Anwendung, die es auch nicht technischen Nutzern ermöglicht, Daten zu analysieren, und ein Berechnungs-Subsystem zur Berechnung von Metriken. Diese Arbeit enthält auch Beschreibungen und Ideen, wie das Konzept umgesetzt werden kann und welche vorhandene Open-Source-Software zur Realisierung von Teilen des Konzepts verwendet werden kann. Schließlich wurde eine szenariobasierte Evaluation durchgeführt, die die Effektivität des Konzepts und seine Grenzen aufzeigt. Daraus ergeben sich offene Fragen für zukünftige Arbeiten zur Erstellung einer Builder-UI oder eines Plug-in-Systems, das die einfache Erstellung neuer Diagrammtypen zur Visualisierung und die Möglichkeit zur Überwachung von Werten und zum Senden von Benachrichtigungen bei Fehlern usw. ermöglicht.

Contents

List of Figures

List of Tables

Listings

1 Introduction

1.1 Motivation

The use of data-driven decision-making has increased in recent years, largely due to the increased availability and collection of data [Rid19]. As studies show, data creation and storage will grow from 64.2 zettabytes in 2020 to an estimated 181 zettabytes in 2025 [Idc21]. By analyzing this data, companies can make better decisions than their competitors and stay ahead of them, since the analytics allow them to gain insight to guide, optimize and automate their business decisions. This results in the achievement of organizational goals, such as reducing production or service costs. [Bos09] As a result, analyzing data is nowadays a critical part of all types of organizations and businesses [Cho18]. There are numerous articles in the popular media and numerous books highlighting the exciting prospects of data collection and analysis, which shows the relevance of this topic. But even with an abundance of data available, extracting value from it is still a challenging task. As a result, a data analysis task will typically involve the use of a combination of tools for the organization and analysis of the collected data. [Aci14]

In light of the previously outlined benefits of data analytics, a regional energy provider sought to use an analytics platform capable of displaying data in dashboards and reports, calculating efficiency metrics, and more for its power plants within its district heating network. Since the structure of the plants can vary from plant to plant, the company wanted a system that would define consistent analytics across these plants with as little effort as possible, as the heating network currently consists of more than 150 power plants. The required analytics platform must therefore be able to handle very complex use cases, which none of the existing platforms can do, or only at a very high cost (see more in Section 4.1), leading to the development of a custom analytics platform.

Other utilities have similar problems, and many other industries need complex data analysis platforms. For example, in smart cities, where vast amounts of data must be collected and analyzed to improve living standards [AN15]. Or in healthcare, where data analytics can help make informed decisions to improve patient health outcomes [Rag14]. In other critical sectors, such as agriculture, analytics platforms are needed to increase food safety and improve efficiency [NKK22]. So the concept was developed and

described in a more generalized way and derived as a concept for an analytics platform that can fulfill complex requirements.

1.2 Objectives and Approach

The objective of this thesis is to design a software for an analytics platform with complex requirements. This includes deriving concept ideas from existing concepts and implementations while showing their limitations. Another objective is to evaluate the designed concept by incorporating scenarios to demonstrate its effectiveness.

The initial step in archiving the objectives involves identifying the system's requirements and subsequently making technical decisions (Chapter 3). Afterwards, existing analytics platforms and concepts are presented and compared to see if they can meet the requirements (Chapter 4). Then, a concept is proposed that fully satisfies the requirements, and some ideas are given on how to easily extend the concept if needed (Chapter 5). After that, the difficult parts of the reference implementation are shown (Chapter 6). Chapter 7 evaluates the previously described concept based on the scenarios identified by the stakeholders during the development of the original system. Finally, the concept is discussed in a conclusion and future extensions are debated.

1.3 Limitation

In order to perform analysis, a lot of data needs to be stored and collected. This is a difficult process because the data typically does not come from a single source, but from many systems in different formats. In order to perform analysis, a system is needed that can integrate all this data into a single system, unify the formats, and build relationships between the data. Since analysis requires accurate data, this system must also validate and clean up the data if necessary. This thesis does not address any of these issues and assumes that such a system already exists. The system that was developed and used for the analysis platform on which this concept is based is described in a co-worker's master's thesis [Pel24].

2 Fundamentals

2.1 Analytics Platform

A data analytics platform is software used to collect, process, analyze and visualize large amounts of data. It provides tools for exploring data, including visualizing the data and running machine learning algorithms or other statistical functions on the data to discover patterns, correlations and trends. In this thesis, the latter is referred to as *metric calculation*. Analysis platforms often use dashboards to visualize and explore data dynamically and interactively. [Amp24]

As stated in the limitations section, this thesis is not about collocating or integrating data, but about analyzing the data, which includes metrics calculation and visualizing the data for human analysis. In a business context, this process is known as *Business Intelligence* (*BI*), so tools that provide this functionality are also referred to as *Business Intelligence platforms*. To simplify the data analysis process, some analytics platforms offer the ability to use a low-code approach to visualize data and calculate metrics. [Ave23]

2.2 Low-Code Platform

Low-code platforms (*LCP*), or also called *low-code application platform* (*LCAP*) and *low-code development platform* (*LCDP*), is a class of software development. Its primary goal is to increase productivity and reduce the cost of developing and maintaining software systems, while also allowing for easy adoption in a rapidly changing environment. This is done by reducing the need for traditional coding and giving the user tools to build software at the problem level without needing to know many details about the underlying software implementation. For example, an LCP gives the user tools to access external data, authenticate the user, and build a GUI with a graphical interface. [Boc21]

Low-code platforms have become increasingly popular in recent years, with research showing that the market revenue for low-code development will increase from $7.87 billion in 2018 to an estimated $32 billion in 2024 [Gar22]. They are particularly popular

for database applications, mobile applications, process applications and request handling applications, and all the major cloud players include them in their general-purpose solutions [Tis19]. Part of this popularity, says Waszkowski [Was19], stems from the problem that while the demand for information systems is growing, the number of people employed in IT departments is not correspondingly growing. This calls for systems that can be operated and customized by non-IT staff, which can be achieved using low-code platforms. [Was19] However, low-code platforms today are mostly used to build small applications rather than applications that are modular and scalable [Tis19].

2.3 Modular and Scalable Software

In software architecture, a modular architecture describes software that is built from multiple components that are interconnected. This makes it easy to replace one part (module) of the software with another without changing the other parts of the software. [Men22]

A scalable software is a software that has the ability to scale itself to adapt to an increased demand. There are two types of scaling, horizontal and vertical scaling. Horizontal scaling distributes the workload among multiple independent machines to improve the processing capacity. In contrast, vertical scaling increases the memory, upgrades the hardware and runs more processes on the same machine. In general, vertical scaling can be applied to almost any type of software, while horizontal scaling requires changes to be made to the software that needs to be scaled. [Sin14] Modular software typically makes it easier to scale software horizontally, because the parts (modules) of the software work independently and can run on multiple machines [Men22]. For easy deployment of modular software, a *container runtime* such as *Docker* can be used.

2.4 Container

Containers can be used to deploy applications in a scalable, secure and easily migrateable way. Instead of running the application on the host system or a virtual operating system, only parts of the host system are used (see figure 2.1). This has the advantage that each container runs in an isolated environment with its own file system, users and environment variables, while still being lightweight and resource efficient. A container is defined using an image, which defines the file system of the container at startup. This image contains all the configurations, libraries and system tools that the application needs to run. To run containers, a container runtime is needed to take the image

definition and run the container. According to a statistic from Datanyze [Dat23], *Docker* [Doc24] is the most popular container runtime. [Hua23]

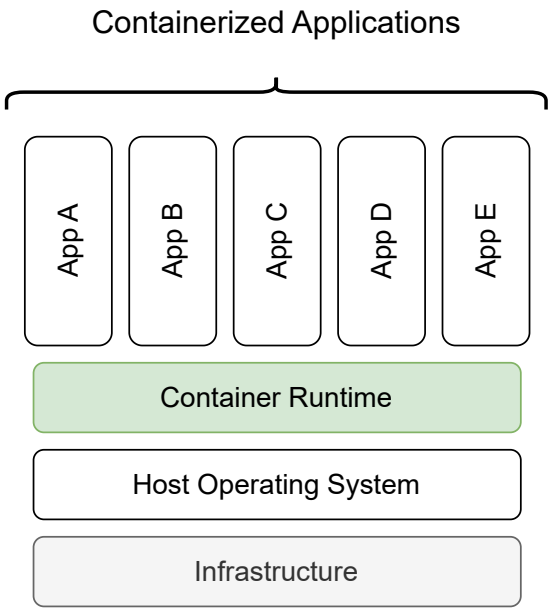

Figure 2.1: Applications in Containers (based on [Doc24])

Since it is sometimes necessary to share files between the host operating system and the container, or between containers, it is possible to use mounting. The container runtime therefore provides the ability to mount files or folders into a container, which are then shared between them. This is also needed to persist data from a container, as when the container is stopped and rebuilt, the container's file system will only contain the files configured in the image. So files or folders that should persist between restarts, such as database data, should be mounted. [Hua23] For the communication between applications running in a container, an application programming interface could be used.

2.5 Application Programming Interface (API)

An *application programming interface (API)* is an interface that allows applications to exchange information with an abstraction of the underlying implementation. It can be used to allow third parties to extend an application. APIs can be divided into two general categories: in-process and out-of-process APIs (see Figure 2.2). In-process APIs are APIs where the API call is handled by the same process for which the call was made.

For example, a *Java* method call from one class to another is an in-process API. An out-of-process API is an API where the call and execution are not handled by the same process. These API calls often involve data that traverses a network, such as a REST API call over the HTTP protocol. [Gou22]

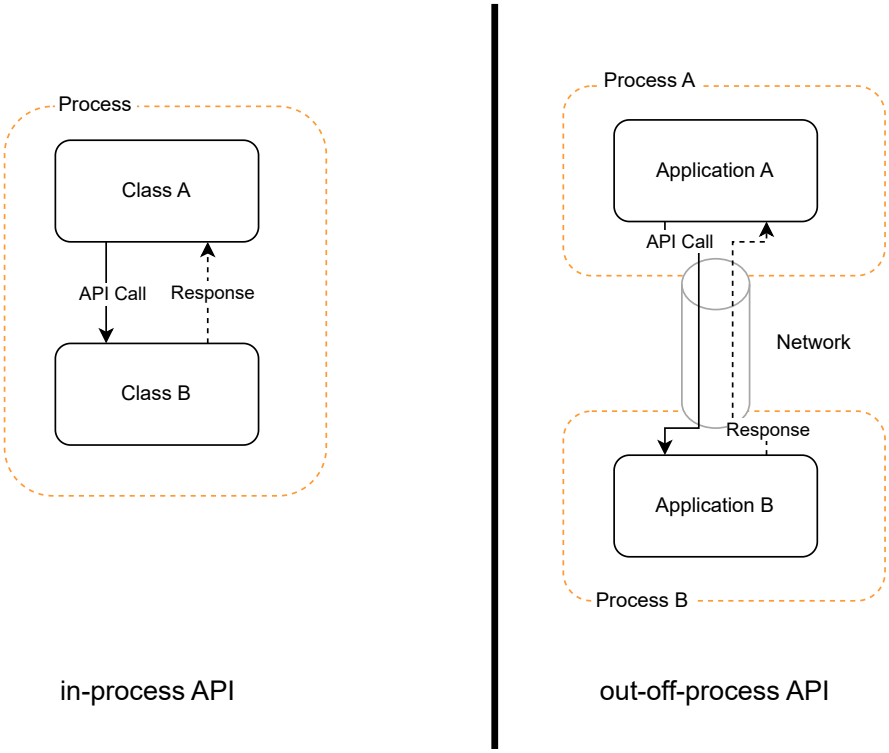

Figure 2.2: Application Programming Interface Categories

2.6 Hypertext Transfer Protocol (HTTP)

The *hypertext transfer protocol (HTTP)* is a protocol for exchanging data. It is used everywhere on the Internet to transfer web pages, images, movies, etc. It uses reliable data transfer and guarantees that the data will not be damaged or scrambled in transit, even if it comes for the other side of the globe. Internet communication via HTTP normally takes place between a web client (such as a browser) and a web server. [Gou02]

HTTP sends messages in a specific format for communication (see Figure 2.3). These messages contain a start line that indicates what to do for a request and what happened for a response. Header fields for additional configuration, and the body, which carries the data. [Gou02]

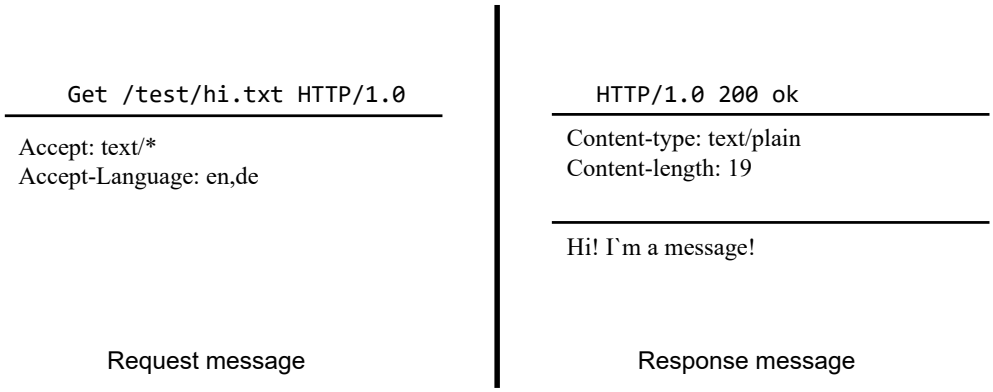

Figure 2.3: HTTP Messages (based on [Gou02])

2.7 Representational State Transfer (REST)

REST, which stands for *Representational State Transfer*, is by far the most widely used API architecture [Pos23]. It was first proposed by Fielding [Fie00] for communication between distributed hypermedia systems. The standard defines principles for accessing and manipulating Internet resources via an API. It defines six constraints that require a client-server architecture, that the API is stateless, that it is cacheable, that it contains a unified interface, that it uses a layered system, and that it allows program code to be sent on demand. In REST, each resource has a unique identifier called a *Unified Resource Identifier (URI)* to access it. A resource could be anything like a file, a dashboard, a user profile etc. So the URI */dashboard/1* could return the dashboard with the ID one. REST uses HTTP as the protocol for client-server communication. The HTTP verbs *GET*, *PUT*, *POST* and *DELETE* are used to specify what should happen to a resource on a REST request. GET simply returns the item, PUT creates a new resource, POST edits a resource and DELETE deletes a resource. In REST, each resource has one or more representations that determine the response to the request. For example, a resource can be returned as a web page in the *HTML* format or in the *JavaScript object notation (JSON)* data format. [Fie00, Gou02]

REST is not the only widely used API architecture, there is for example GraphQL, which was the third most used API technology in 2023, used by 29% of respondents [Pos23].

2.8 GraphQL

GraphQL [The24e] is a query language that can be used to query data for building client applications. It describes data requirements and interactions using an intuitive and flexible syntax. In GraphQL, a request source is called a document, which is similar to a resource in REST. Each document can contain operations: *queries*, *mutations*, and *subscriptions* to retrieve data, modify data, or be notified of data changes. To define the capabilities of a document, GraphQL has a type system that defines the request and response types. The definition of all documents of a service is referred to as a schema (see Listing 2.1 for an example). [The24e]

Listing 2.1: GraphQL Schema

```
type Plant {
  title: String
  variant: String
}
```

To retrieve data, a GraphQL query can be sent to the server via an *HTTP POST* request, which contains a self-defined name, the documents it wants to receive, and the response fields it is interested in (see Listing 2.2 for an example). This query then results in a JSON response that contains the requested information or an error (see Listing 2.3 for an example). [The24e] There are many open source programs and libraries available to use GraphQL, such as Apollo Server [Apo24b] and Apollo Client [Apo24a].

Listing 2.2: GraphQL Request

```
query PlantQuery {
  plant(variant: "heat") {
    title
  }
}
```

Listing 2.3: GraphQL Resonse

```
{
  "data": {
    "plant": {
      "title": "Heat Plant"
    }
  }
}
```

2.9 Open Source Software (OSS)

Software that is called *open source software* is software that has a license that guarantees everyone to read, redistribute, modify and use the software without cost. These criteria are defined by the Open Source Initiative [Ope24]. Normal OSS is developed publicly by internet-based communities of volunteers, but there are also some companies or

non-profit organizations that provide their software as OSS. These companies usually earn money with services and support around the software or non-open source extensions. There are many open source software systems available today that are used by companies around the world. [Ben10] In order to specify and document the architecture of an open source system, the C4 model can be used for this purpose.

2.10 C4 Model

The *C4 Model* [Bro24b] is an approach to documenting software architecture using diagrams. It defines four levels of diagrams to create an abstraction layer that makes it easier for viewers to see only relevant parts of the architecture. The model defines four basic architectural parts: *Person*, *Software System*, *Container* and *Component*. The *Person* represents a human user of the software. A *Software System* describes something that provides value to its users, and is the highest level of abstraction. It includes the software system described by the C4 Model and the software systems required for the current software system. Other definitions often refer to it as an application, product, or service. A *Container* represents an application or data store that must be run for the software system to function. It has nothing to do with the Docker container described above except for the fact that the names match. The last part is a *Component*, which is a collection of related functionality encapsulated behind a well-defined interface. For example, in the Java programming language, this is a collection of implementation classes that are encapsulated behind an interface. [Bro14, Bro24b]

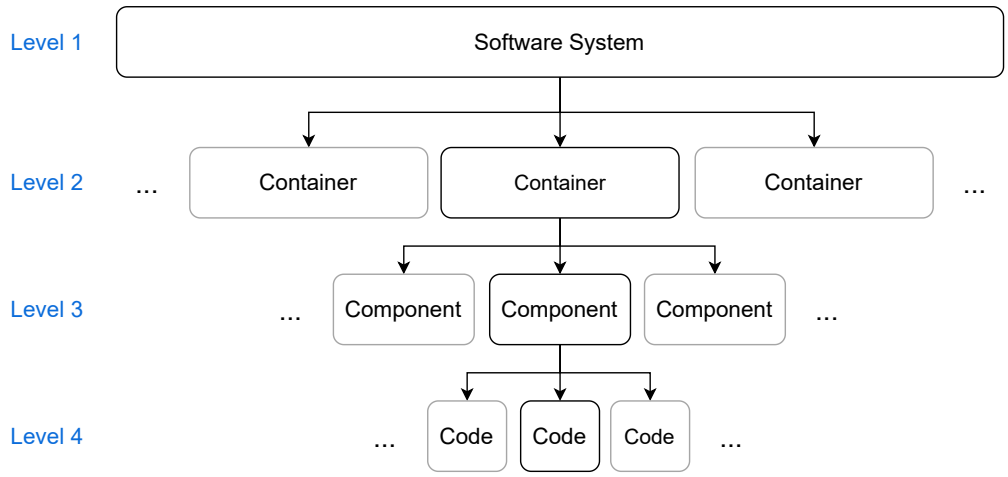

Figure 2.4: C4 Model (based on [Bro24b])

The four levels are defined in a hierarchical order so that the user can zoom in to see more and more details of the architecture. This concept can be compared to something like

Google Maps, where it is possible to zoom in to see more details of the map, but instead for an architectural description. The first level is a *context diagram*, which is a high-level diagram that shows the scope of the software, including system dependencies and actors. The second level is a *container diagram*, which shows high-level technology choices (*Containers*) with their dependencies and the communication between them. The third level is a *component diagram*, which shows the components and their relationships. The last level is the *code diagram*, which shows the implementation details of each component using existing code diagrams such as UML class diagrams, etc. The last two levels are optional and can be omitted if they do not add value, or they can be auto-generated from the actual implementation. [Bro14, Bro24b]

3 Requirements

To design the analytics platform, requirements are needed to lay the foundation. The requirements are derived from the specific features of the company and from existing analytics platforms. Bot are outlined next and afterwards the requirements are presented. The requirements are divided into two categories. Functional requirements, define what functions the system must provide and quality requirements, define the quality that the previously described requirements must satisfy. A key of the form *<first category letter><number>* (e.g., *F1 - Functional Requirement one*) is assigned to each requirement for later reference.

3.1 Foundations

In order to define the specific requirements for the energy supplier analytics platform, a product requirements document was created. This was used as a basis for the concept requirements. Table 3.1 shows an excerpt with the relevant requirements.

Keyword	Requirements
Reports	It should be possible to view all stored data in dashboards that contain visual elements. Users should be able to create and edit these dashboards.
Metrics	Efficiency metrics should be calculated for each plant. These metrics should be flexibly created and edited by the user. Some metrics may require machine learning algorithms.
Easy use	No programming should be required to create dashboards. And it should be possible to create templates or reuse parts.
Time	It should be possible to analyze and display data in specific time periods and granularity, such as monthly and yearly.
Exports	Exporting data and dashboards in common formats should be possible.
Grouping	There should be the ability to create individual or general dashboards.
Security	A user administration with different authorization levels should be provided.

Table 3.1: Excerpt from the Product Requirements Document

Existing analytics platforms fall into three categories. First, there are analytics platforms that focus on creating and combining algorithms to generate metrics and insights for the data. This includes the *KNIME* [KNI24b] platform, which allows blocks of algorithms to be graphically linked together to analyze data. It is also possible to visualize the data and there results, but it's not possible to create interactive dashboards. The second categories are analytics platforms where the main focus is to visualize the data on dashboards. The tools *Metabase* [Met24b] and *Apache Superset* [The24c] are the most popular in this category and allow to use a low-code approach to visualize and combine data by creating interactive dashboards. The third category consists of systems that can do both. These include systems from the major cloud or software vendors such as *Google Cloud* [Goo24b], *Microsoft Azure* [Mic24b] and *Amazon AWS* [Ama24], *SAP* [SAP24] and *Snowflake* [Sno24b]. All provide a platform consisting of multiple tools to analyze and visualize data. There are also open source tools such as LinceBI [Lin24] and Knowage [Eng24]. These also provide a platform for visualizing and analyzing data. The concept for which the requirements are built must be of the third category to fulfill the needs from the product requirements document.

3.2 Functional Requirements

F1 - Use by non-technical persons. Because the people who analyze data for an organization are typically not programmers or system administrators, but data scientists, they have a limited technical background. Therefore, the system must be easy for these people to use without the need for technical support. So the system needs an intuitive graphical interface for all features. This is similar to all the analytics platforms listed above, which all provide this functionality in some way.

F2 - Easily addable and flexible dashboards and metrics. Because new dashboards or metrics are often needed or change over time, it should be easy to add new dashboards and metrics without programming, such as in Metabase, Apache Superset for dashboards and KNIME for metrics. It is also difficult to predict which dashboards and metrics will be needed in the future, so they should be as flexible as possible.

F3 - Individual and general dashboards. To account for the fact that some dashboards are used by only a few users and others by almost everyone, it should be possible to create 'individual' dashboards that only the creator sees, and 'general' dashboards that are shared with all users who need them.

F4 - Dashboard reusability. Since some part of a dashboard and its display may be similar in different dashboards, it should be easy to reuse dashboard or parts of a dashboard.

F5 - Data abstraction. As an analytics platform can contain a lot of data with complex structures and relationships, it should be possible to define abstractions over the structures. This would make it easier for users to use this data without a deeper understanding of the underlying data structure. This would also reduce the initial overload and make it easier for administrators to change these structures without affecting the analytics. Just like Metabase, Apache Superset or Snowflake, for example.

F6 - Export to standard formats. Since some dashboards and analysis results may be shared with non-system users or other companies, it should be possible to export the data to standard formats such as PDF, Image (PNG, JPG), JSON, and CSV. This can be done by any of the analytics platforms listed above.

F7 - Time range analysis [use case specific]. Many analyses need to be performed over a given period of time with a certain granularity in the use case of the reference system, an analytics platform for a local energy supplier. For example, a user wants to monitor the efficiency of a heat generation plant for this year grouped (granularity) on a monthly basis. But the given values may not be at the given granularity (e.g., monthly), so the system must provide a way to determine the values for that granularity. For example, if the meter readings of a given heat generation plant are generated daily and the granularity is monthly, the system needs to find the value for the last day of the month and use it for analyzes.

F8 - Same outcome with different queries [use case specific]. Since the structure of a heat generation plant can be very complex, the system needs a way to define analyzes in a general way, but with variables that can be changed for each plant. So that the definitions of the analyses do not have to be done several times similarly, but with the flexibility to realize the differences in the plants. For example, if a heat generating plant was a heat pump, an efficiency indicator is the coefficient of performance (COP), which can be calculated with ($\frac{|Q|}{W}$), where Q is the heat output and W is the electricity used for its production. The formula itself is always the same, but where Q and W come from differs from heat generation plant to heat generation plant. So it should be possible to define the formula for the COP globally, but the mapping for each heat generation plant must be stored separately and inserted when the calculation is actually done.

F9 - Machine learning predictions [use case specific]. To calculate metrics or predicted future values, the analytics platform needs a system that can perform machine learning predictions, such as AI analysis, on the stored data. These calculations should be able to generate new data that can also be viewed and analyzed by the user or used for future calculations. To make the process as easy as possible, these calculations should be performed automatically when the data changes.

3.3 Quality Requirements

Q1 - Scale system easily. Because usage and data can grow rapidly, it should be easy to scale the system as needed without much effort.

Q2 - Efficient data Excess. To avoid waiting time for users, the system should access data efficiently to reduce system load times. This can also help reduce system operating costs, as efficient data access reduces the system resources needed.

Q3 - Secure data access. Since an analytics platform has access to most of the critical data in a company, security is a critical issue. Therefore, it should be possible to define restrictions on which users can access which data.

Q4 - Extendable and replaceable components. Since the software was developed for special use cases, it should be easy to add more of these special use cases by extending the software with new parts or replacing existing parts with a better or possibly already existing implementation. This also includes the system's API, which should allow all the system's features to be used by an external program for easy integration into other programs.

Q5 - Keep development and maintenance costs low. In order to keep the cost of the system close to that of a standard solution, the design and development should balance the trade-off between fully satisfying the requirement and the cost of development and maintenance. To do this, and to focus on the specifics of the application, open source software should be used to implement parts of the software whenever possible.

Q6 - Host platform on-premises [use case specific]. Since the analytics platform holds sensitive data, it should be hosted on-premises, so all services used must provide this option.

4 Related Work

4.1 Differentiation from Existing Systems

This section contains a distinction from existing analytics platforms, which are first described in section 3.1 and were initially selected based on extensive internet research. This is accomplished by describing how well the systems can meet the requirements described in Chapter 3.

The *KNIME* analytics platform fulfills many functional requirements: analyses can be performed easily and flexibly for non-technical users [F1], analyses (called workflows in KNIME) can be shared with others [F3, F4], data can be exported to standard formats [F6], and time-series or machine learning predictions are possible [F7, F9]. Its disadvantages are that KNIME does not allow configuring data abstraction [F5] and unify very complex metrics [F8]. It also does not have an easy way to create visual dashboards that update in real time for users who just want to see the data [F2], and some analyses still require some programming knowledge. Another disadvantage is that in order to share workflows or parts of them with own infrastructure [F3, F4, Q6] an expensive license is needed, starting at €35,000 per year. [KNI24b, KNI24a]

Since *Metabase* and *Apache Superset* have very similar features, they are grouped together in this comparison. Both allow non-technical people to create interactive dashboards [F1, F2], share them with others [F3], reuse parts of them [F4], export data [F6], and perform time range analysis [F7]. In terms of data abstraction, both allow this, while in Apache Superset it requires programming, as Metabase allows this with a graphical editor [F5]. Both do not allow the definition of metrics or machine learning predictions [F8, F9]. [Met24b, The24c]

The tools from the major vendors all provide solutions for most of the requirements, except for the requirement [F8]. But they have the disadvantage that they are very costly, for example to use Snowflake a minimum cost of €6,000 per month [Sno24a] is required, which does not include the cost of setup, integration, etc. Adding custom features can also be very expensive. Some solutions, such as Snowflake, cannot be hosted on-premises [Q6]. [Goo24b, Mic24b, Ama24, SAP24, Sno24b]

LinceBI can handle most of the requirements expect [F8]. However, it is difficult to set up and use because the documentation is not very informative, and some parts are only available in Spanish. It is also not very transparent which features are included in which versions, so the open source version may not be able to meet all requirements. Customizations are also very difficult due to the poor documentation. [Lin24]

Knowage can handle most of the requirements except [F8] and [F1] only partially. While some parts of the analytics platform can be used and configured by a non-technical person, some parts cannot. For example, the configuration of the data abstraction [F5] can only be done by a technical person. The documentation on how to deploy the program in a production environment is also poor. [Eng24]

As this differentiation shows, there are programs that could solve the requirements, but at a high cost or effort. And even then, they may not be as customizable as desired. So building a custom solution that still uses some available open source tools seems to be the best compromise for the requirements to get a solution that fits them well by not being too expensive. For comparison with existing concepts in the literature, some of them are listed in the next section.

4.2 Literature

Related literature was searched in the *ACM*, *AIS*, *IEEXplore*, and *Sciencedirect* databases using the keyword 'analytics platform' and 'Business intelligence'. The publications found were limited by their focus on either describing a concept for an analytics platform, deriving features needed for a good analytics platform, or comparing different analytics platforms. In addition, many of the publications found were not selected because they were strongly focused on data integration or solving a problem with a single, already existing analytics platform. At the end there were six publications left, which will be described in the following.

Three of the selected publications describe the concept and implementation of an analytics platform. Pankaj et al. [Pan06] describes important factors about dashboard implementations and challenges in doing so. Some of their important factors include that the dashboard should be built out of widgets and enable the analysis of dynamic data that allow the user to take proactive action in the operational time frame. The publications do not contain much detailed information about the implementation itself. It is also older (from 2006) which results in technical limitations and smaller amount of available data to be analyzed/displayed.

Wang et al. [Wan23] describes a concept for a low-code analysis platform in the energy sector. The concept proposes to build dashboards from multiple slices to make their

appearance flexible. To decouple the visualization from the business logic, they allow the user to build the slices without a strict binding to specific data. To display the data, the user then uses a splice arrangement that maps the data to the visual elements. This arrangement can also include filters to restrict the data used. The limitations of the concept for Wang et al. [Wan23] are that they lack the ability to perform complex computations [F8, F9] or time analysis [F7]. They also do not describe how the platform could be extended, e.g. to add new types of diagrams (called graphs in the paper) without much effort [Q4].

The third concept of Sá et al. [Sá24], describes a low-code approach to business analytics. Their concept allows users to configure and customize a dashboard (called panel in the paper), built from widgets, for data analysis without any code. The limitations are that the user has to upload the data himself and there are no complex calculations [F8, F9] or time analysis [F7] possible. The concept also does not address how future extensions could be handled without much effort [Q4].

Gunklach et al. [Gun23] conducted a study on what are the current problems with dashboards and how data stories can solve these problems. They identified seven key issues for dashboards. Based on this, they proposed five requirements for data stories to solve these problems. They then developed and evaluated a data story based on the requirements. The evaluation showed that three requirements can help improve data stories: Data stories should include a narrative structure by presenting information sequentially, they should include guidance in the form of text-based explanations and annotations, and they should include data perspectives that the user can choose to easily access the data that is relevant to them.

The last two publications are studies that compare current analysis platforms. While a comparison of current tools is already available in Section 4.1, these studies still provide important insight into what a good analytics platform needs.

Khatuwai and Puri [Kha22] compare the tools Tableau, Power BI, and Spago BI. They present six key characteristics of analytics platforms (referred to in the paper as BI tools). According to them, an analytics platform should provide executive dashboards, location intelligence, what-if analysis, interactive reports, a data abstraction layer, and the ability to rank the available data in dashboards.

The publication by Aveiro et al. [Ave23] compared several open source analytics platforms (they call them business intelligence platforms) for integration into a low-code platform. First, they analyze which open source analytics platforms exist and then select four platforms for further comparison: LinceBI, Knowage, Metabase, and Superset. Five criteria were used for the comparison: Visualization and Dashboards, the ability to query, aggregate and analyze data without coding, the ability to import data from

multiple systems, the ability to handle large and complex data sets, and the ability to perform data mining techniques and algorithms. The paper identified Knowage as the best fit for their use case.

In conclusion, none of the concepts proposed in the relevant literature can meet the requirements. However, there are still helpful insights about which problems can occur, how they could be solved, and which features are important for an analytics platform. Therefore, literature learning were derived from the literature review, which were taken into account later in the concept and are presented below.

L1 - Data Visualization. Aveiro et al. [Ave23] suggest that visualization features are crucial to enhance user experience, simplify tasks and make data analysis easier to understand. Other literature such as Sá et al. [Sá24], Wang et al. [Wan23], and Khatuwai and Puri [Kha22] also use data visualization as part of their concepts.

L2 - Explore/Visualize Data without Coding (Low-Code Approach). Aveiro et al. [Ave23], Sá et al. [Sá24], Wang et al. [Wan23], and Khatuwai and Puri [Kha22] all suggest allowing the user to explore and visualize the data without coding.

L3 - Dashboards should be build from multiple Build Blocks. According to Sá et al. [Sá24] and Pankaj et al. [Pan06] dashboards should be built from multiple widgets to show all relevant information. Wang et al. [Wan23] call these widgets building blocks, and Gunklach et al. [Gun23] also use building blocks to define their proposed dashboards.

L4 - Easy Adoption of Dashboards for different data Perspectives. According to Pankaj et al. [Pan06], different personalities in an organization need different data, so different or customizable dashboards are needed. Gunklach et al. [Gun23] also suggests providing different data perspectives that can be selected by the user so that the user can see the information the user is interested in.

L5 - Ability to Aggregate or Combine Data into Meaningful Metrics. To make better business decisions, data should be aggregated or combined into meaningful metrics according to Pankaj et al. [Pan06]. The system proposed by Wang et al. [Wan23] also allows the user to combine multiple data sets into a single diagram.

L6 - Interactive and Real-Time Analyses. Since nowadays, many data are collected in real time, Aveiro et al. [Ave23], Khatuwai and Puri [Kha22], Wang et al. [Wan23] and Pankaj et al. [Pan06] propose the use and support for real time analysis. To better analyze these real-time data, Aveiro et al. [Ave23], Khatuwai and Puri [Kha22], and Wang et al. [Wan23] also propose or use interactive analysis.

5 Concept

The literature learnings from Chapter 4 are used to propose a concept for an analytics platform that fulfills the requirements described in Chapter 3. The proposed concept is described using the C4 Model. The next section contains a selection of the overall system architecture. Then the system context diagram is shown. After that, all the services are shown in a container diagram. Details about what the services do, why they are needed, and how they should be implemented are then explained in the following sections. The context diagram and container diagrams in this Chapter include the shapes shown in the legend of Figure 5.1. The third and fourth level diagrams are not included in this concept as they would contain excessive implementation detail.

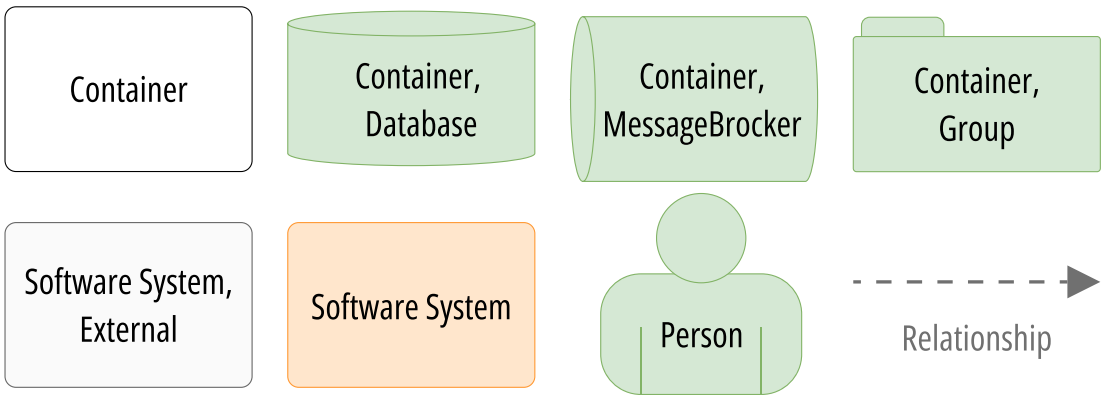

Figure 5.1: Diagram Legend

5.1 System Architecture

This section describes the overall architecture style of the system. This is done by first giving an overview of the most common architectures and then selecting one. Note that this architecture style is the base architecture style for the described concept, but this does not mean that some parts of the system may follow a different architecture style. This concept is known as architecture partitioning, and is a common approach because everything in software is a trade-off [Ric20].

Overall, architecture styles can be divided into two main categories: monolithic and distributed. Monolithic architecture styles have a single deployment unit for all the program code, while distributed styles have a number of deployment units that are connected to each other through a network. For this concept, the monolithic architectures *layered*, *pipeline*, and *microkernel* and the distributed architectures *service-based*, *event-driven*, *space-based*, *orchestration-driven service-oriented*, and *microservices* are considered and compared in the following section. The architectures described by Richards [Ric20] form the basis of this selection. [Ric20]

In order to make the comparison easier to read, a preselection is made first, without explaining the architectural styles in detail. The remaining styles are then explained and compared. The preselection is based on Richards' [Ric20] architecture ratings and the system requirements (Chapter 3). The architecture ratings consist of ratings for a set of characteristics of the architecture. Ratings for evolution [Q4], modularity [Q4, Q5], performance [Q2], and scalability [Q1] are the most important factors in the preselection process as these can be derived from the requirements. The characteristics used and an explanation for them are shown in Table 5.1. [Ric20]

Characteristics	Explanation
Evolutionary	How well handles the architecture implementation and criteria changes.
Modularity	Does the architecture work well with independent and interchangeable modules?
Overall cost	Is it possible to build a system using this architecture without incurring significant costs?
Performance	How well can a performant system be implemented with this architecture?
Scalability	Can a system built with this architecture scale easily?
Simplicity	Is this architecture easy to implement?
Testability	How well can a system with this architecture be tested?

Table 5.1: Explanation of Architectural Characteristics (based on [Ric20])

The *layered architecture* has a poor rating on each of these characteristics (1–2/5) and is therefore not considered any further. Both the *pipeline architecture* and the *microkernel* architecture have bad ratings for scalability (1/5), so they are not discussed either. The *orchestration-driven service-oriented architecture* has a bad rating at Evolutionary (1/5) and a not so good rating at Performance (2/5) with also excludes it. Similarly, the *microservices'* architecture also has a not so good rating for performance (2/5), but very good ratings for all other characteristics (5/5), which is why it is still considered for the comparison. All other architecture styles have good ratings for the relevant characteristics. Therefore, they are used for the comparison (see Table 5.2 for an overview). [Ric20]

Service-based architecture

The service-based architecture style typically consists of a discrete UI, discrete remote coarse-grained services, and a monolithic database (see the upper left diagram in Figure 5.2). However, there are also topologies with multiple discrete UIs or multiple databases. These services are typically independent and separately deployed parts of the application. Often these services are partitioned by domain (for example, item assessment) and are therefore referred to as domain services. A user interface accesses them through a remote access protocol, such as REST, which usually accesses the services directly, but can also use an API layer between them. This style is a very popular choice for many business-related applications because it is a flexible distributed architecture without the complexity and cost of other distributed architectures. [Ric20]

Event-driven architecture

This style consists of decoupled event processing components that receive and process events asynchronously (see the top-left diagram in Figure 5.2). It can be used to create highly scalable and high-performance applications. This may be used as a standalone architectural style or embedded in other architectural styles, such as event-driven microservices. Its unique asynchronous capabilities can be a powerful technique for increasing the overall responsiveness of a system. [Ric20]

Space-based architecture

The space-based architecture is designed to solve bottleneck issues with applications that have high user load or variable and unpredictable concurrent user volumes. It consists of multiple processing units, a virtualized middleware, and a data reader/writer (see the lower left diagram in Figure 5.2). The Processing Units contain the application logic or parts of it. Instead of reading/writing data directly into the database, it is kept in the memory shared among the Processing Units, to then persist the data, it is read/written asynchronously to the database throwing queries and the Data Reader/Writer. The virtualized middleware is there to handle the incoming requests and coordinate the processing between the processing units. The benefits of this architecture are elasticity, scalability and performance, but it comes at the cost of a very complex application that is also difficult to test. [Ric20]

Microservices architecture

The Microservices Architecture is an architecture in which the application is divided into microservices. Microservices are small, independent services that work together to create a large software system (see the lower left diagram in Figure 5.2). Each microservice is an independent, deployed process that can be written in different programming languages and updated independently. To collaborate, microservices communicate

over the network, for example using the REST protocol. The main advantage of this architecture is its great decoupling, since each microservice runs independently, it can use the best components for its use case, for example, it is not dependent on the shared database and could use a different database style if it fits better. This also makes it easy to scale the application because each microservice can be scaled individually. All of this comes at the cost of duplication of some parts, high development complexity, and performance drawbacks. [Ric20, New15, Wol18]

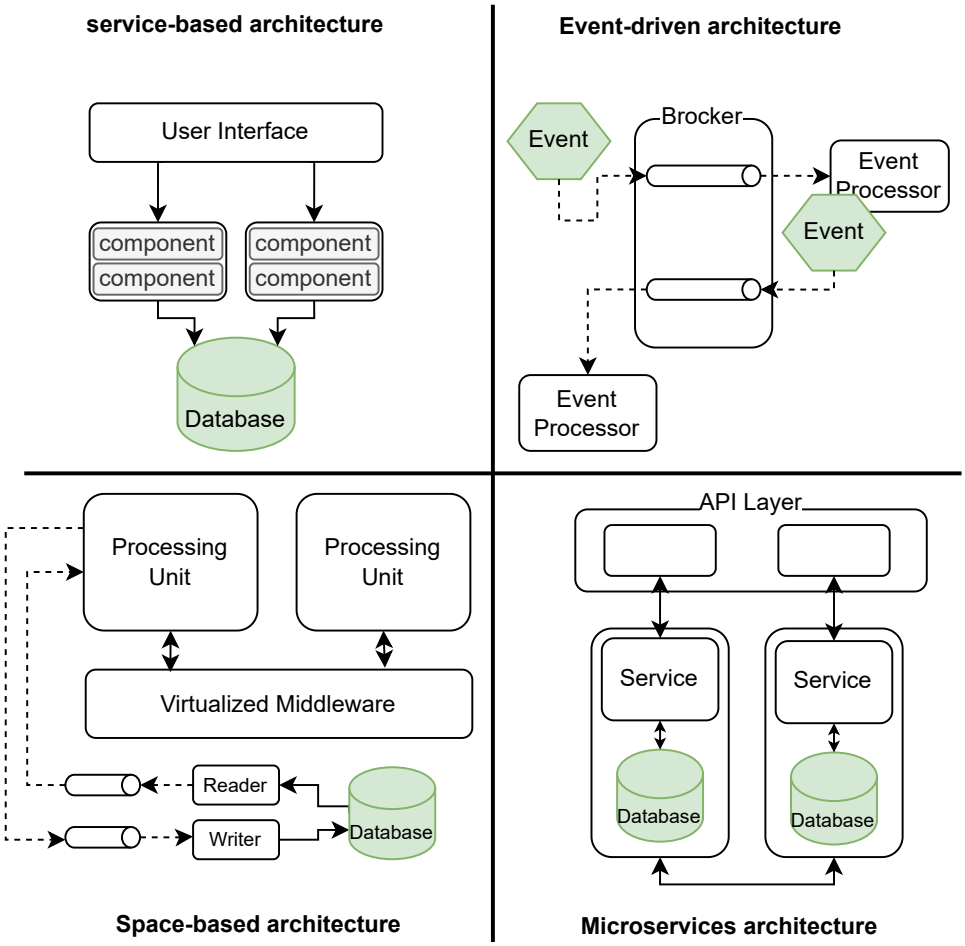

Figure 5.2: Architecture Styles (based on [Ric20])

Table 5.2 gives an overview of the strengths and weaknesses of the selected architectures, with a rating from one to five for each characteristic. The characteristics Evolution [Q4], Modularity [Q4, Q5], Performance [Q2], and Scalability [Q1], Total Cost [Q5], Simplicity [Q5], and Testability [Q5] from Richards' [Ric20] are used for the selection, which includes all the characteristics from the preselection. The meaning of each characteristic is explained in Figure 5.1. As the table shows, *microservices architectures* are very

good at evolution, modularity, scalability, and testability, but very bad at performance, total cost, and simplicity. *Event-based architectures* have similar strengths, except for poor testability, but therefore have a better overall cost rating. The main advantages of a *space-based architecture* are performance and scalability, at the cost of being very complex to implement (simplicity, testability, total cost). *Service-based architectures* are not as good as *microservices* and *event-based architectures* in their strong characteristics, but still very decent, with the advantage that they are easier to implement. So they give a good *trade off* of all the chosen characteristics, which this architecture is chosen for this concept.

Characteristics	Service-based	Event-Driven	Space-Based	Microservices
Evolutionary	●●●○○	●●●●●	●●●○○	●●●●●
Modularity	●●●●○	●●●●○	●●●○○	●●●●●
Overall cost	●●●●○	●●●○○	●●○○○	●○○○○
Performance	●●●○○	●●●●●	●●●●●	●●○○○
Scalability	●●●○○	●●●●●	●●●●●	●●●●●
Simplicity	●●●○○	●○○○○	●○○○○	●○○○○
Testability	●●●●○	●●○○○	●○○○○	●●●●○

Table 5.2: Rating of Architectural Characteristics (based on [Ric20])

5.2 System Context

The analytics platform is designed to be accessed by the data analytics user directly or through a third-party system. As described in the limitations section, the analytics platform requires a pre-defined data store with all necessary data stored. To run complex computations, the system also needs a container runtime to execute them. More information about this in section 5.8.1.

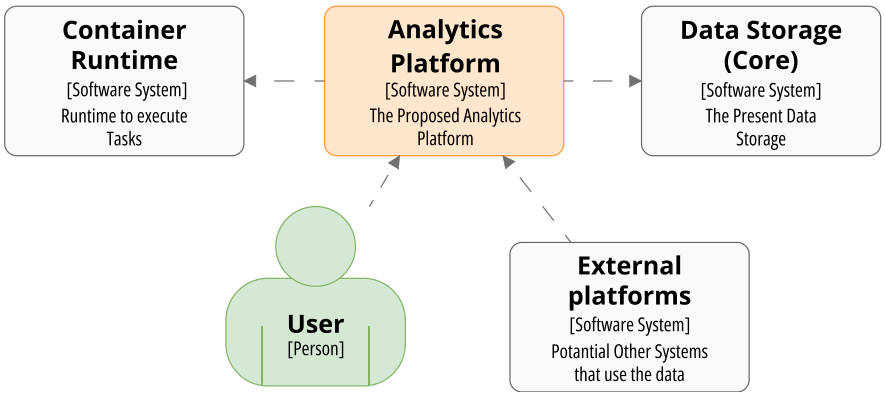

Figure 5.3: [System Context] Analytics Platform

5.3 System Overview

Figure 5.4 shows an overview of all services of the application and their data flow. To meet the requirements and to make the concept easier to understand, the Analytic Platform has been divided into five main features: 5.4 Interface, 5.5 API Security, 5.6 Data Storage, 5.7 Data Access and 5.8 Analysis, which are implemented with one or more services. A detailed explanation of why these services are needed and how they are implemented can be found in the following sections. To improve clarity and reduce the number of containers shown in the diagram, all containers related to the Calculations feature are hidden in Figure 5.4, a figure containing these containers and all their dependencies can be found in the Section 5.8.1 (Figure 5.15).

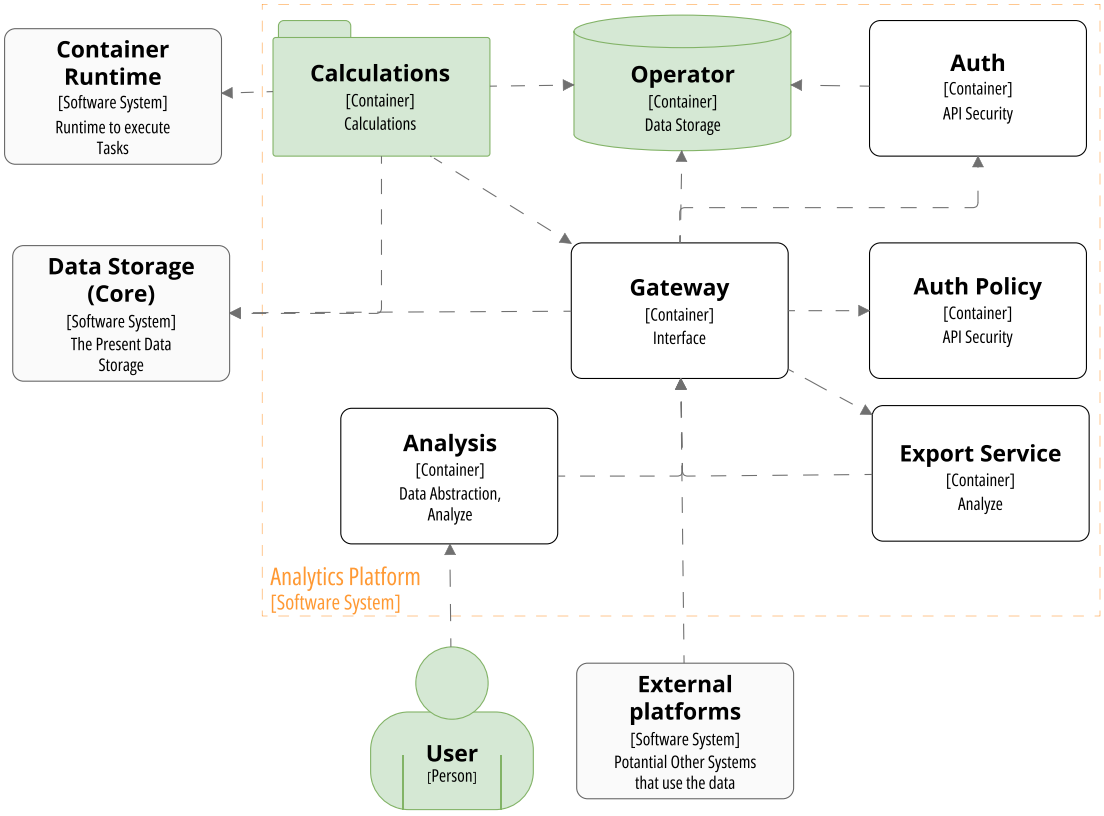

Figure 5.4: [Container] Analytics Platform

5.4 Interface

To implement an interface for accessing the services, two concepts could be used. One concept is to directly access the services APIs in the web front-end. The other concept uses the API gateway pattern for communication between the web front-end and the services. The API Gateway pattern describes an architecture in which there is an API

Gateway service that is the single point of entry to all services. All communication to the services goes through this endpoint. (see Figure 5.5). [Zha18, Dud20]

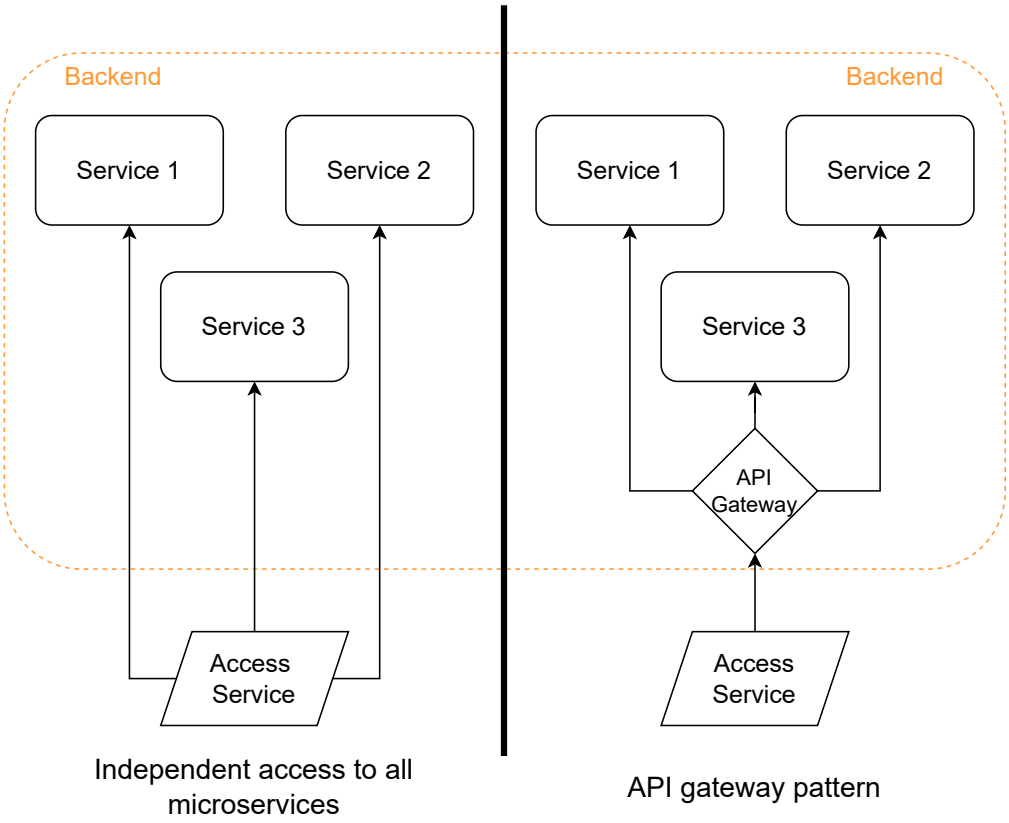

Figure 5.5: API Gateway Pattern

This concept uses the API Gateway pattern as it has several advantages, which is implemented by the *Gateway* service. Because it allows easier standardization of the API [Q5], has higher abstraction which makes it easier to swap out services [Q4, Q5]. It also makes requests more efficient [Q2], since only a single request is needed even when multiple services are involved. Furthermore, it increases the security of the system [Q3] because a single system checks the authentication, the architecture and services are not publicly accessible, and the architecture of the system is harder to figure out. [Zha18, Dud20]

Another consideration for the API gateway is how the data is requested by the client. It could be either a REST API or a GraphQL API (see Section 2.7 for an explanation of both). This concept suggests using GraphQL as it has a better resource usage [Q2] according to Lawi et al. [Law21]. And a study by Brito and Valente [Bri20] also shows that it is easier to implement API requests via GraphQL, even for experienced developers.

5.5 API Security

To secure access [Q3] to data, analysis configuration, user accounts, etc., two essential elements are authentication and authorization. Authentication is the process of verifying that the user is who they say they are, and authorization is the process of verifying that the current user is allowed to do what they are trying to do. [Jin18]

5.5.1 Authentication

There are two main methods of API authentication: *Basic authentication* and *token-based authentication*. [Mad21]

Basic authentication is the simplest technique, which sends the username and password encoded in *base64* in the HTTP request header (see Listing 5.1). This technique has the disadvantage that the username and password are needed in every request and therefore must be stored by the application (*Analysis*) for an acceptable user experience [F1]. Since this can lead to serious security problems [Q3], this technique is not adopted for this concept. [Mad21, Jin18]

Listing 5.1: HTTP Basic Authentication

```
Authorization: Basic dXNlcm5hbWVAcGFzc3dvcmQK
```

With token-based authentication, instead of sending the username/password combination to the server each time, there is a special login endpoint that handles a temporary token for the client. This token can then be used for further authentication (see Figure 5.6). This makes it more secure [Q3] because the password is not stored in the *Analysis* service and the tokens expire over time, meaning that a stolen token is less damaging than a password. That's why the concept uses token-based authentication, where the special login endpoint is realized by the *Auth* service, which takes the username and password and generates a token. [Mad21]

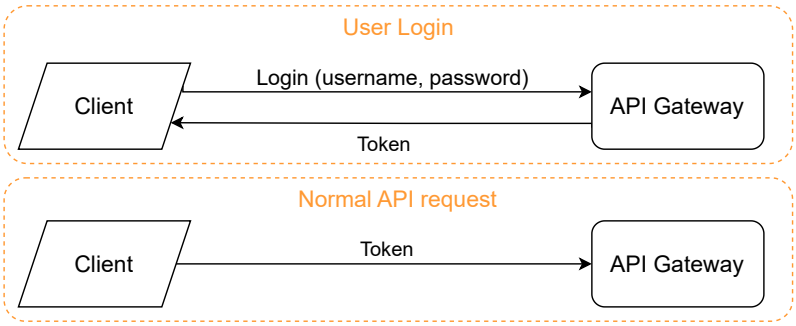

Figure 5.6: Token-based Authentication

There are two ways to implement token-based authentication: *database-backed (DB) tokens* and *self-contained (SC) tokens* (see Figure 5.7). With database-backed tokens, the access tokens are stored in the database to verify the token provided by the user and to look up user information. In contrast, with self-contained tokens, all information is stored in the token itself. To verify that the information stored in the token is valid, the SC token contains a signature that can be validated by the server to verify the authenticity of the token. [Mad21]

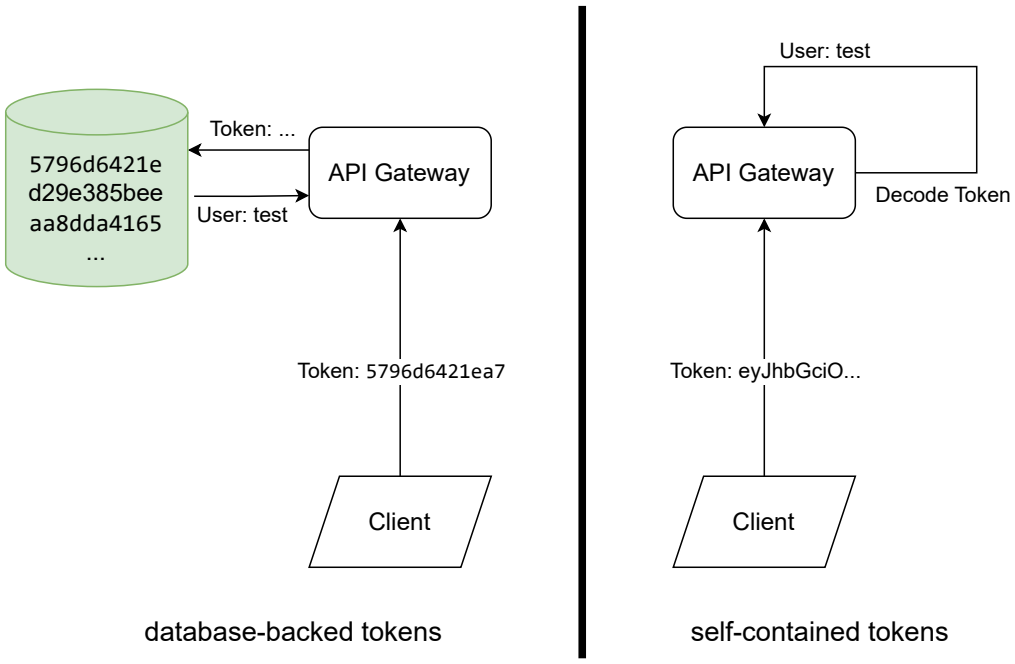

Figure 5.7: Token-based Authentication Variants

Both methods have advantages and disadvantages. While DB tokens can be easily revoked if a token is leaked, they do not scale very well, are difficult to develop, and create a single point of failure since a database is needed to store the tokens. In contrast, SC tokens are difficult to revoke because once a token is issued, it remains valid until its expiration date. However, SC tokens do not have the disadvantages of DB tokens and also make validation in multiple services easier. [Mad21]

Since both have advantages, this concept uses a hybrid approach. The authentication tokens are SC tokens with a short lifetime. However, a second SC token, a refresh token, is created so that the user does not have to repeatedly log in [F1]. This token can then be used to obtain a new authentication token. The refresh token's ID is stored in the database to check on a refresh whether the token has been used more than once; if this is the case, all tokens generated from the same login chain, called a refresh token family, are invalidated (see Figure 5.8). This concept is known as refresh token rotation.

[Lod24] This results in high security and the ability to revoke tokens with minimal delay (maximum the expiration time of the access token). At the same time, the disadvantages of DB tokens are minimized, since only a database call is needed to update the refresh tokens, and the access tokens can still be used without it. The refresh token rotation can also be implemented by the *Auth* service.

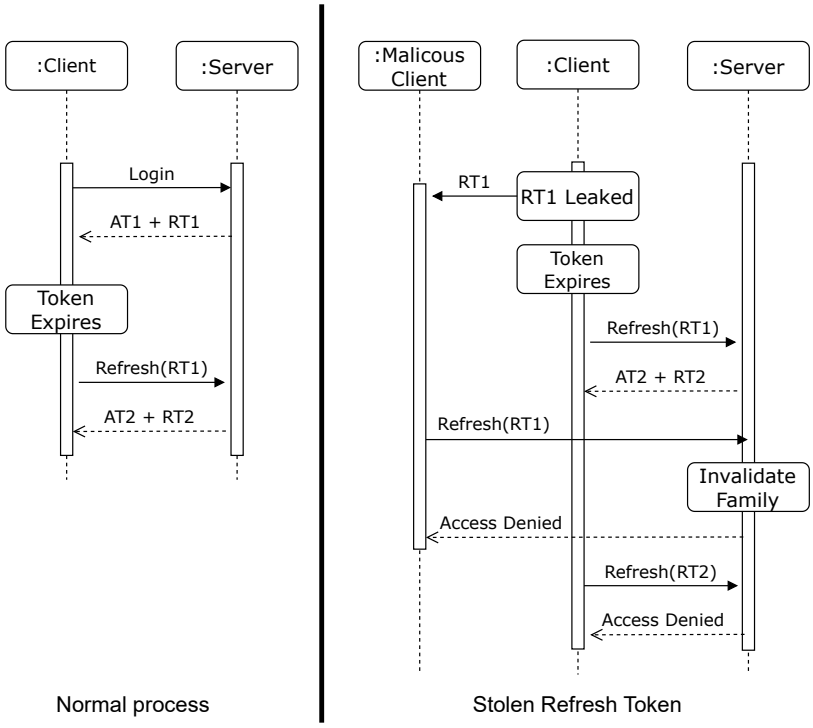

Figure 5.8: Refresh Token Rotation

Another issue is how the authorization and refresh tokens are passed to the client/server. There are two main possibilities: Cookies and the HTTP header [Mad21]. Since both have their advantages, and it is not difficult to support both, the concept is to use both and allow the client to decide which method is best for its purposes.

Single Sign-On (SSO)

To integrate the analytics platform into the existing infrastructure, it may be necessary to integrate it with an existing *single sign-on (SSO)* system. A single sign-on system is a central system that stores user information and allows a single sign-on for multiple applications, so the application simply passes the user authentication process to the SSO system [Shr24]. To integrate this into the Analytics Platform, the *Auth* service can, instead of providing the username/password combination, just handle the communication

with the single sign-on system and generate the token when the SSO has validated the user's credentials (see Figure 5.9).

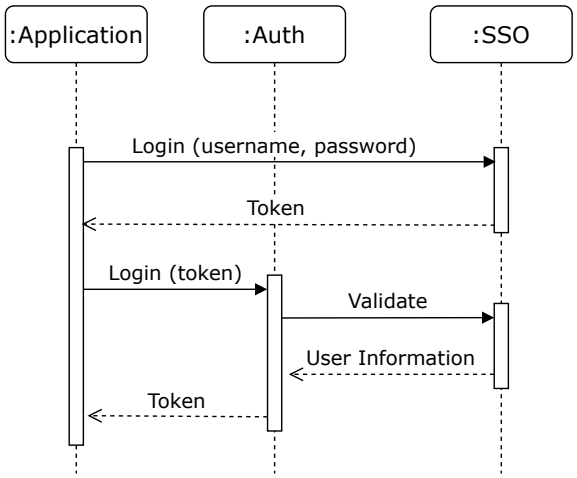

Figure 5.9: *Auth* Service with SSO

External System Authentication

To allow automated systems to access the Analytics Platform API [Q4], users should be able to create system accounts. It would be bad for security [Q3] if users had to give their username and password to these automated systems. As this would make it difficult to identify if the user made the changes/requests or the system, it is not possible to lock only some systems if they are compromised and the user has many more accesses writes (see section 5.5.2) than the automatic system needs.

Since an automated system has no problem 'remembering' many symbols, a system account does not need to have a username and password combination, but is authenticated with a token called an API key. This has several advantages; the token is more secure because the user cannot choose the symbols and length by himself, no username has to be invented, etc. When the system user wants to make an API call, he must first provide the API key to generate an access token, which can then be used to query the API (see Figure 5.10). This is like logging in for a 'normal' user. The API key is a self-contained (SC) token that contains the system user's ID. This makes it possible to revoke a system user's authentication by removing the user from the database or setting a flag that prevents the user from logging in. For this to work, the *Auth* service must check on token login to see if the system user still exists and is not locked out.

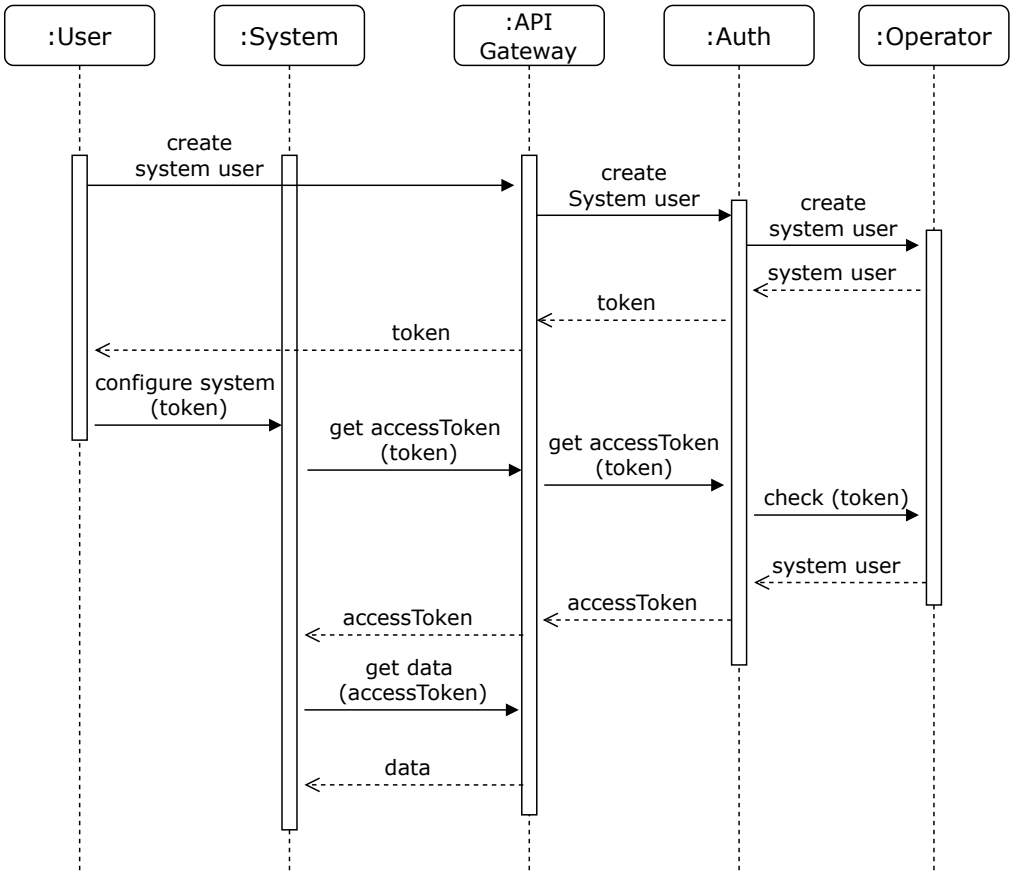

Figure 5.10: Token Login

5.5.2 Authorization

There are four main paradigms for performing authorization, often referred to as access control: *Discretionary access control (DAC)*, *mandatory access control (MAC)*, *role-based access control (RBAC)*, and *attribute based access control (ABAC)*. [Ben06, Hu15]

Discretionary access control allows the owner or administrator of a resource, such as a file, to control who has what rights. This can be done, for example, with an *Access Control List (ACL)* that defines resources with users and their permissions (see Listing 5.2). [Ben06, Pfl24]

Listing 5.2: Access Control List Example

```
ACL(File1) = ((Alice, {read, write}),(Frank, {read}))
ACL(File2) = ((Bob, {execute}), (Janet, {read}))
```

Unlike DAC, mandatory access control does not define controls directly on resources and users, but defines a hierarchical system that defines who can access which resources, where each resource and each user is assigned to a hierarchical level. For example, there are public, confidential, and secret levels: Then a user with the level confidential can access resources with the levels public and confidential, but not with the level secret (see Figure 5.11). [Cha21]

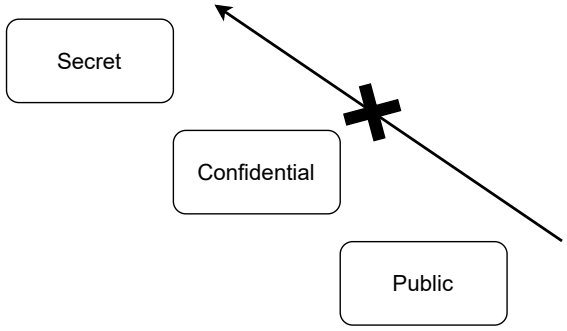

Figure 5.11: Mandatory Access Control Example

Role-based access control grants access to resources based on a user's role. A role is an abstraction that embeds a set of privileges along with the associated allowed actions. Thus, unlike DAC and MAC, privileges are not assigned to users, but to roles. [Ben06]

Listing 5.3: Role-based Access Control Examlple

```
Roles:
  - admin: can access everything
  - editor: can edit files
  - moderator: can write comments to files
  - user: can read files

User1: admin
User2: editor, moderator
User3: user
```

ABAC facilitates the authorization of access to objects by applying a rule-based evaluation process that takes into account the attributes of entities (including users and resources), operations and the environmental context relevant to a particular request. [Pfl24]

Listing 5.4: Attribute Based Access Control Examlple

```
User attributes: Department
File attributes: Department, Creator, Archived

Policies:
```

```
- Can view a file if the user is in the same department as the document
- Can edit a file if the user is the owner and the file is not archived
- Deny access before 6 o'clock
```

This concept suggests using role-based access control or attribute-based access control. As MAC cannot fulfill the requirement F3, since its hierarchical structure does not give enough flexibility. DAC could be used to fulfill F3, but it would be very cumbersome to maintain access rights to the stored data [Q3]. Because every user must be given access to all the data he wants to access. Both RBAC and ABAC can handle all requirements, which one should be used depends on the usage of the analytics platform. The proposed implementation (see Chapter 6.2) uses RBAC because it is sufficient for the use case.

The *Auth Policy* service was designed to implement this. Because it must be possible to determine whether an operation can be performed by users based on their roles or their attributes. So the *Auth Policy* service takes a request from one of the other services with all the necessary information like the user, the resources he wants to access, etc. and with this information and the stored information about the roles or policies it evaluates if this operation can be performed. It passes the result to the service, which performs the operation based on the decision or returns an error (see Figure 5.12). To obtain the user and their roles/attributes, the user must provide their authorization token, which should then be passed to the *Auth Policy* service, since this token contains this information. Therefore, the *Auth* service must include this information, which it receives from the *Operator* service, when it creates the authentication token.

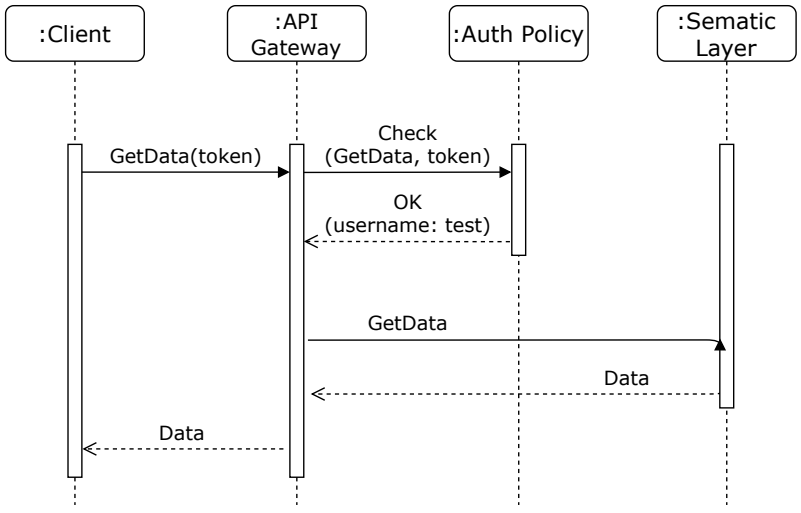

Figure 5.12: *Auth Policy* Service Usage

When implementing authorization for system users, they can be treated just like normal users, allowing them to be assigned roles or attributes to ensure correct write access.

5.6 Data Storage

To store metadata, such as the structure of a dashboard, user accounts, etc., the analytics platform needs a database to store this data. This data could be stored in the *Data Storage (Core)* service, but to separate analytical and operational data, which increases security [Q3] and makes implementation and scaling easier [Q1], a separate database is used. This database is implemented by the *Operator* service.

To access and store data through the *Gateway* service into the database, it must be accessible through an API. There are two ways to achieve this: Write a service that provides an API to access and modify the stored data, or use a tool that automatically generates an API based on the data structure (see Figure 5.13). The first approach has the advantage that it may be possible to combine multiple requests into one, which can improve performance and load times. On the other hand, it is much more difficult to develop and has a higher probability of containing bugs or security vulnerabilities. Also, many API generation tools include the ability to write custom functions at the database level, for instance PostGraphile [Pos24], which largely negates the first advantage of a custom solution. So this concept proposes to use a tool that automatically generates the API. There are also some databases that include such a tool out of the box, such as 'Dgraph' [Dgr24], which can be used if the selected database ships with it.

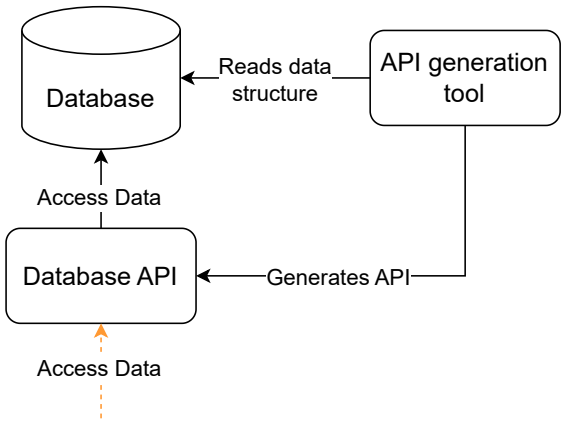

Figure 5.13: Automatic Database API Generation

5.7 Data Access

Since the concept does not specify how the *Data Storage (Core)* should be implemented or which application should be used to provide this functionality, many things related

to data storage are outside the scope of this concept. However, this section provides some features that the *Data Storage (Core)* should fulfill to be used with this concept.

The service should provide an option to retrieve the data through an API endpoint that can then be accessed through the *Gateway* service (see Section 5.4) to make the data available through the central API endpoint. In addition, the service should provide access to a standardized query language (such as SQL) to facilitate the use of an existing platform [F6]. The concept requires that the data be accessible in groups of data (called tables) with fields (called columns) that contain the values. The columns should have a data type to give the *Analysis* service more information about how to render them. In the later sections, these tables are also referred to as data sources. To abstract the analysis for the actual data source [F5] and to make the application easier for non-technical users [F1], since they no longer need to know anything about the underlying data structure, it should be possible to define 'virtual' data sources. For these virtual data sources, it should be possible to select columns from different data sources that should be included in this data source to create a new data source with these columns.

5.8 Analyze

To query and analyze the stored data as a non-technical person [F1], an application is required. This is implemented by the *Analysis* service described in Section (5.8.2). In order to fulfill the requirements [F7], [F8] and [F9], a separate system is required to perform complex calculations of metrics that are later used for analysis. The concept of this is described in Section 5.8.1.

5.8.1 Calculations

To perform complex calculations [F7, F8, F9] of metrics, the analytics platform needs a system that can calculate them on demand. Due to the complexity of implementing this, multiple services are required. So the feature is implemented as a subsystem with its own software architecture. This will be discussed in the following section, after which the concept will be proposed based on the selected architecture.

Subsystem Architecture

The subsystem should use the event-driven architecture style (see Section 5.1 for an explanation) because its unique asynchronous capabilities are perfect for a computation

system that has many independent tasks where the system that initiates those tasks does not need a response from them.

An event-driven architecture can be implemented with two primary topologies: the mediator topology and the broker topology (see Figure 5.14). Both topologies have an initial event that starts the process that can lead to a chain of subsequent events. The difference between them is that the broker topology has no central event mediator, but works with a chain-like transmission mechanism via a lightweight message broker (such as RabbitMQ [Bro24a], etc.). Thus, in the broker topology, the initial event is published to the broker, which is then consumed by an event process that publishes a processing event upon task completion. If another event processor is interested in this event, it uses it for its task and publishes a processing event of itself upon completion. This chain continues until no one is interested in the 'last' processing event. In contrast, the mediator topology uses a central event mediator to coordinate the workflow for events that require the coordination of multiple event processes. The initial event is sent to the event mediator, which generates processing events in different event channels, which can then be processed by different event processors. [Ric20]

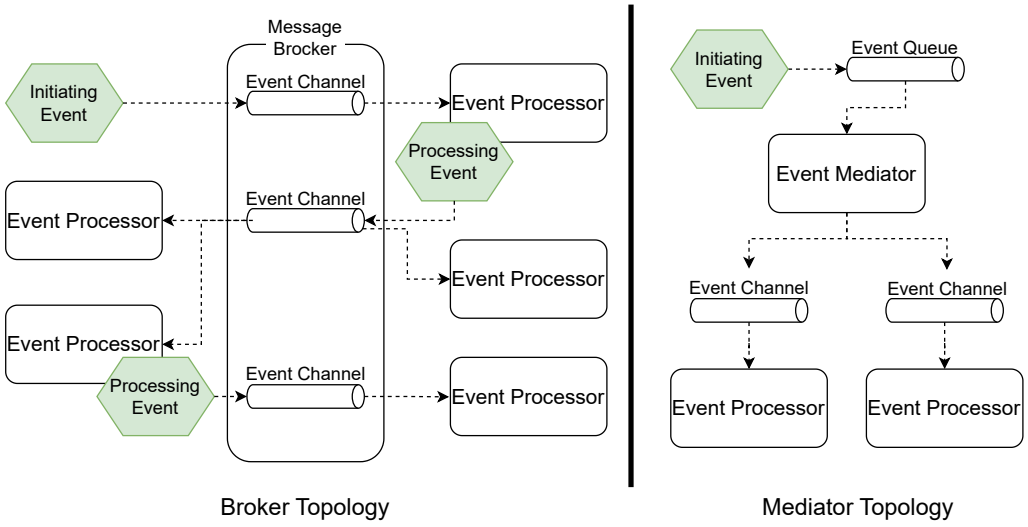

Figure 5.14: Event-driven Architecture Topologies (based on [Ric20])

Both topologies have advantages over each other, while the broker topology brings highly decoupled event processor, high scalability, etc. It has the disadvantages that the workflow control, error handling, etc. are difficult. The mediator topology on the other hand tries to solve the problems of the broker topology at the cost of shrinking the advantages of the broker topology. Thus, the choice between these two concepts boils down to a trade-off between workflow control and error handling capabilities on the one hand and high performance and scalability on the other. This concept uses the broker topology because high performance and scalability [Q1, Q2] are much more

important for a computation system than workflow control and error handling. Also, since the calculation service does not require many event processors, these issues are not as significant. [Ric20]

Subsystem Concept

Figure 5.15 shows an overview of the services needed for the subsystem, with all other services from this concept that have any relation to this system. The concept of services and their communication is explained below.

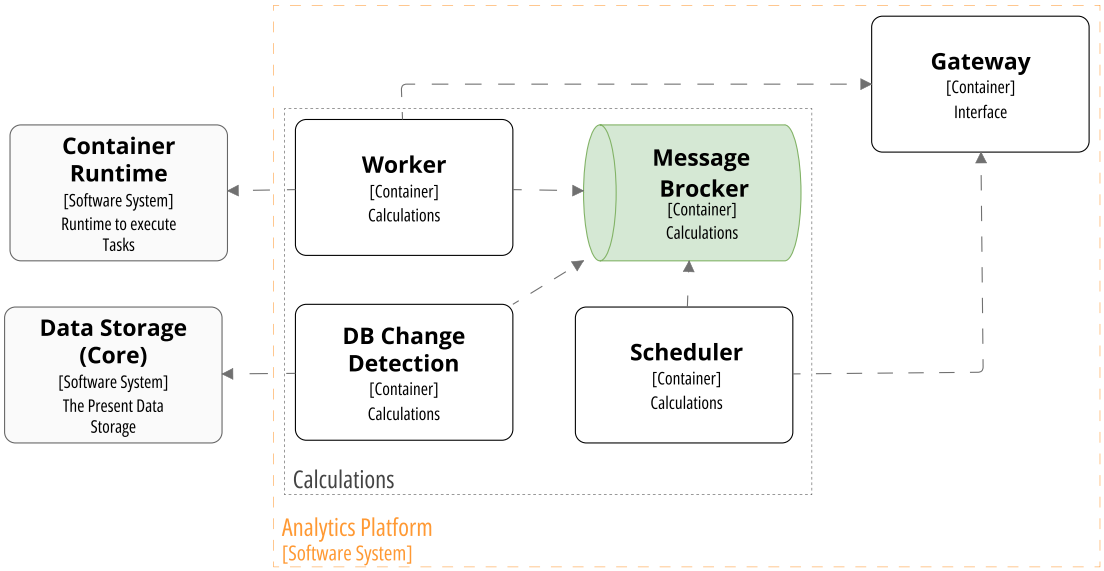

Figure 5.15: [Container] *Calculations* with all Dependencies

Since the [F9] requirement demands that calculations be reevaluated when the required data changes, the system needs a way to find out when and what data has changed. This should be implemented by the *DB Change Detection* service. How the service checks and retrieves the changes is beyond the scope of this concept, as it depends heavily on the chosen *Data Storage (Core)* service implementation. When the service detects a change, it should publish that change as an event to the *Message Broker* service, this event is the initial event.

The *Scheduler* service listens to these events and uses them to generate job events, which are its processing events (see Figure 5.18 for a sequence diagram). To generate these jobs, the service should lock up which jobs are configured in the *Operator* service, which could be configured with the application described above. These jobs should then include information about which data sources they depend on, so that the *Scheduler* can calculate which jobs need to run on the current changes (see Figure 5.16). The *Scheduler* will then create an event for each job that depends on the changed data

sources. These jobs should also contain the information for an actual computation task to be performed, which can be a simple computation task or a machine learning task. The service also includes the changed data in the job event so that it can be used to execute the task. To improve performance and reduce system load, the *Scheduler* service should combine multiple changes to the same data source into a single job if they occur in close succession.

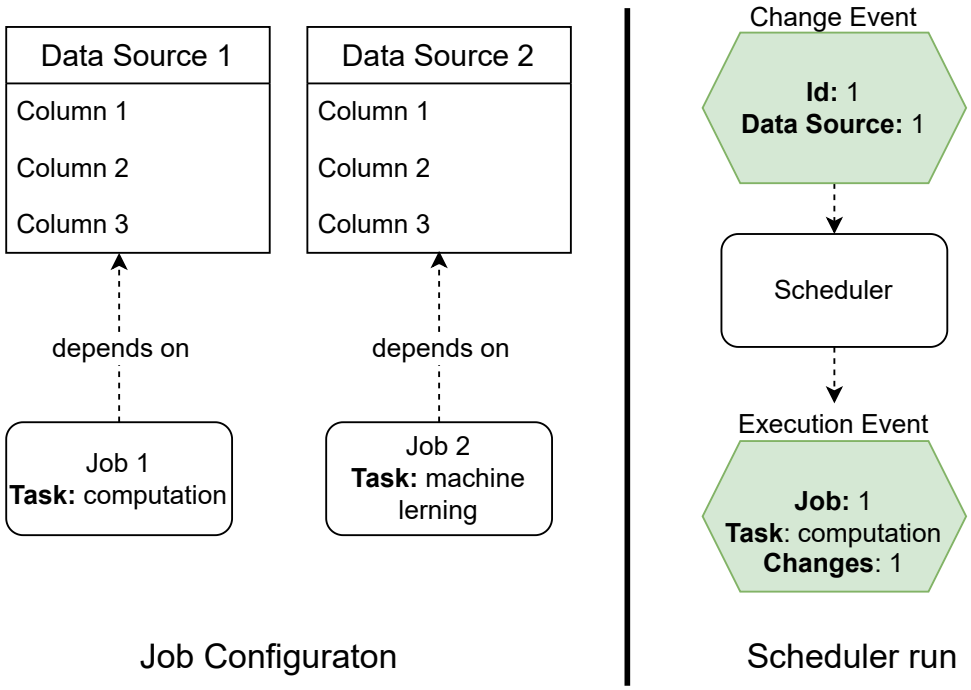

Figure 5.16: *Scheduler* Job Generation Example

The *Worker* service exists to consume these job events and perform the actual calculations (see Figure 5.18 for a sequence diagram). Since the jobs are independent of each other, there could be more than one *Worker* service running at the same time if necessary, which would then increase the number of jobs that could be executed in the same amount of time [Q1].

To accurately calculate a job, the *Worker* may need more data than just the changes. For example, the *Worker* calculates a difference between two values; if only one value changes, the system needs to fetch additional data, in this example the second value for the difference calculation. To do this, the job configuration should provide a way to fetch data from the *Data Storage (Core)* service, which can use the changes as a filter to dynamically resize the fetched data.

In order to run the job and accurately calculate the intended data, code is required to perform this calculation. Since the system should not be limited in what calculations it can perform [F9, Q4], it must be possible to run custom code that can be selected via the job configuration. The concept fulfills this requirement with the ability to specify a container image in the job configuration. This image is then used to start a container that performs the actual computation. To do this, a container runtime is required that is able to run these containers, for example a Linux server with Docker installed. In this concept, these container images are referred to as runners.

The before described solution to start a container for the calculation brings a future problem with it, how the container gets the data (changes and additional data), the configuration and how the worker gets the result back. To solve this problem there are three possible solutions. Write this information into environment variables, read/write this information into files that are then mounted in the container, or allow the container to access/write this information via an API, for example via the *Gateway* service. The first option has the limitation that the size of an environment variable is limited (e.g., *128 KB* on a normal Linux system [Mat22]), since the size of the data can be much larger, this option is not suitable. The third option is much more complex to implement than the second, and also has security issues [Q3]. Since the container would need access to the same network as the gateway service, and the gateway service would have to check that the job can only read/write the data it needs, etc. So the second option is used in this concept (see Figure 5.17). The *Worker* service generates two files, one containing the data and another containing additional configurations. These configurations are read from the job configuration and are there to allow users to make customizations to the job execution without rewriting the code. These files are then mounted into the container, which can then use them. To get the results back, the Container should write them to another file that can be read by the *Worker* service.

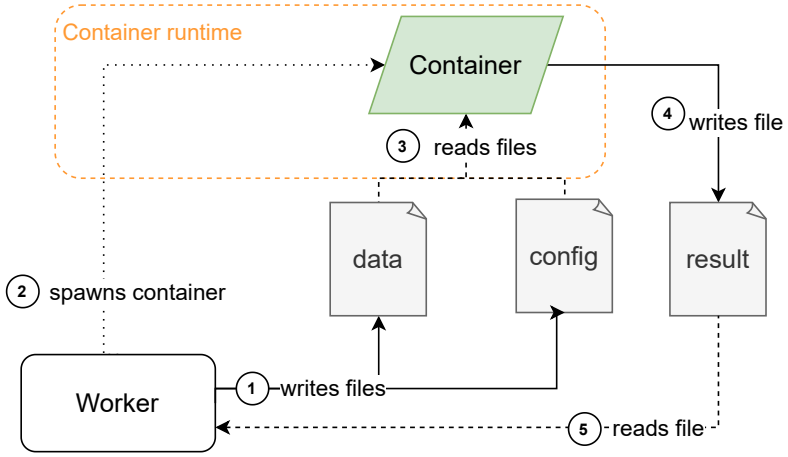

Figure 5.17: *Worker* Data Exchange

These results then need to be written back to the *Data Storage (core)* service to be used in future analyses and displayed to the user. As the data structure is job dependent, it should be possible to define the output data source with its columns and data types in the job configuration. The *Worker* service should handle the creation of this data source and also the write-back of the result data.

The entire process of performing complex computations is shown in Figure 5.18 as a sequence diagram to give a complete overview of the calculation concept.

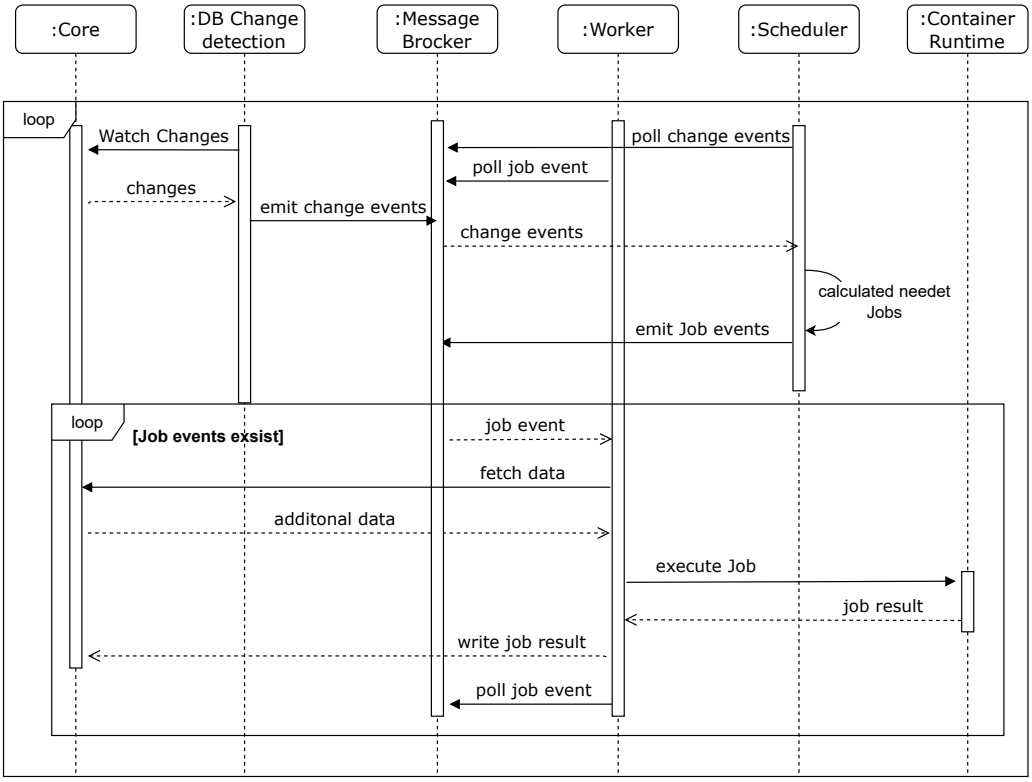

Figure 5.18: *Calculations* Process

5.8.2 Visualization

To meet the requirement that the application be usable by a non-technical audience [F1], the application in many cases uses the low-code approach to allow the user to configure/extend the application without the need for coding knowledge. For example, it is possible to create new dashboards using a low-code editor.

Besides analyzing, the application should also perform other tasks and allow users to change settings, etc. This gives the user a single point to use and manage the analytics platform [F1]. To accomplish this, the application should have a navigation bar on the left side that allows the user to access the different features of the application (see the

wireframe in Figure 5.19). To increase security [Q3] and make the application easier to use [F1], the application should also display the features that the current user is allowed to access. The header of the application can display useful information to the user, such as which account they are currently logged into [F1], etc.

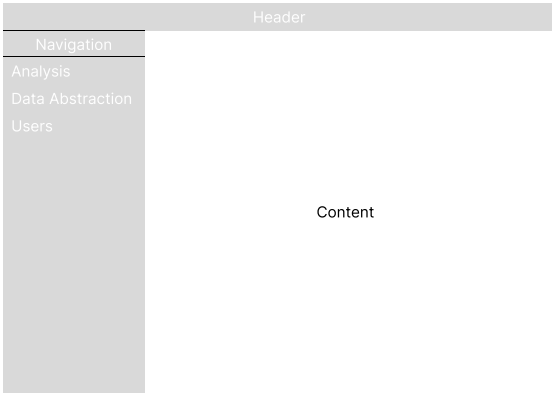

Figure 5.19: Application Overview (Wireframe)

Analysis

In order to make analyzes and save/present them [F2, F3, L1], the application should provide so-called reports. A report can be used privately by a single user or shared with others [F3, L4]. The application should present all reports that the user has access to (see left wireframe in Figure 5.20). A report itself consists of filters and multiple views (see the right wireframe in Figure 5.20). Views allow the user to build a report that consists of several individual blocks [L3], such as charts, text sections, etc., and makes it possible to present multiple information on a single page [L5]. This also makes it easier to reuse parts of the report [F2, F4, L4], as single views can be used as templates for views in other reports. To make the look of a report as customizable as possible [F1], it should be possible to drag and drop views and change their size. Views are also referred to as widgets in similar systems such as Pankaj et al. [Pan06] and Sá et al. [Sá24].

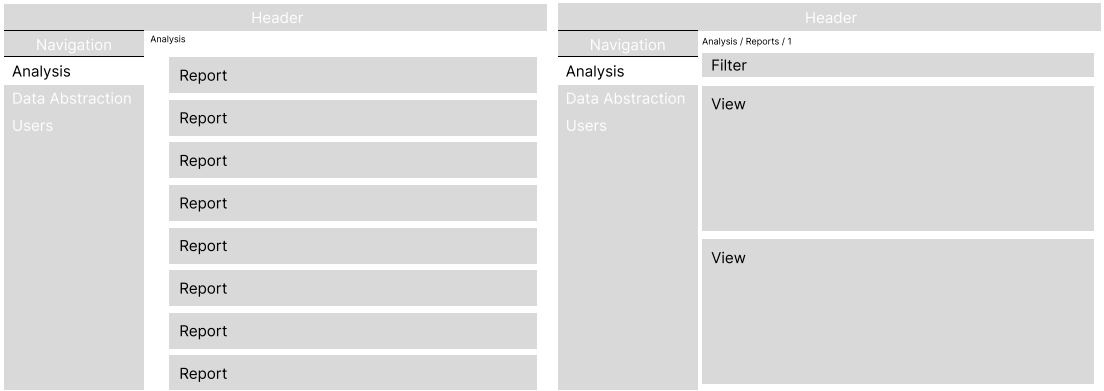

Figure 5.20: Application Analysis (Wireframe)

Views can be opened individually to explore them in more detail and to create/edit them (see Figure 5.21). The view itself also displays and uses the filters from the report that are relevant to the current view (all filters with an assignment to the displayed data are relevant to a view, see below for a detailed explanation). This allows future data exploration at the view level [L6], and keeps the view consistent with the report, as when a user opens a view from a report, it should still display the same data.

A view consists of one or more diagrams. A diagram is there to display the data [F2, L1]. The type of diagram determines how the data is displayed. The application should provide several diagram types like text, table, bar chart, line chart, pie chart etc. To customize the appearance of the diagram and the selected data displayed, each diagram type comes with its own type-specific settings. To allow combining different diagrams, it is possible to create multiple diagrams on a single view, which allows better comparison of the data or better presentation of the data [F2, L1, L5]. For example, it might be possible to display a line chart on top of a bar chart, or a line chart in a table. Since not all combinations of charts are valid, the chart type implementation should be able to specify which combinations are valid and prevent the user from creating unusable combinations [F1].

When a users edit a view a settings page should be displayed on the left side of the application (see Figure 5.21). There it should be possible to select the data source of the diagram (see Section 5.7), the diagram to edit, the current type of the diagram and other type specific settings.

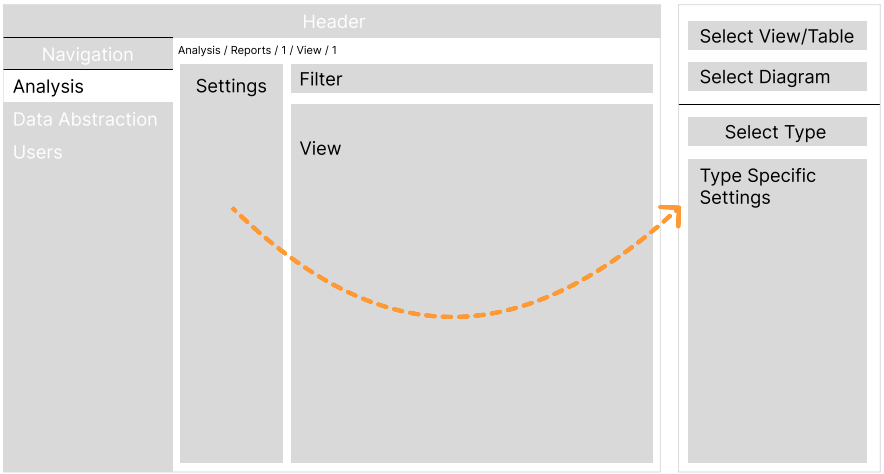

Figure 5.21: Report View (Wireframe)

To allow the user to export the data [F6], the application should display a button at the report and view level to select the desired format and download it. There are two options to implement the transformation of the data and the download: Transform the values into the desired output format locally or giving the user a download link where

the application can download the data from and transform the values in the backend (see Figure 5.22). The first one has the advantage that its structure is easy to implement. But the disadvantages are that the size of the exported data is limited by the user's memory, which varies from user to user, resulting in inconsistent behavior [F1]. So the advantage for the second solution is that the size limit can be defined globally by the application and changed without user notice if needed, the other advantage is that format errors could be fixed without user notice since the application does not need an update for this. Based on this, the second option is used for this concept and realized via the *Export* Service.

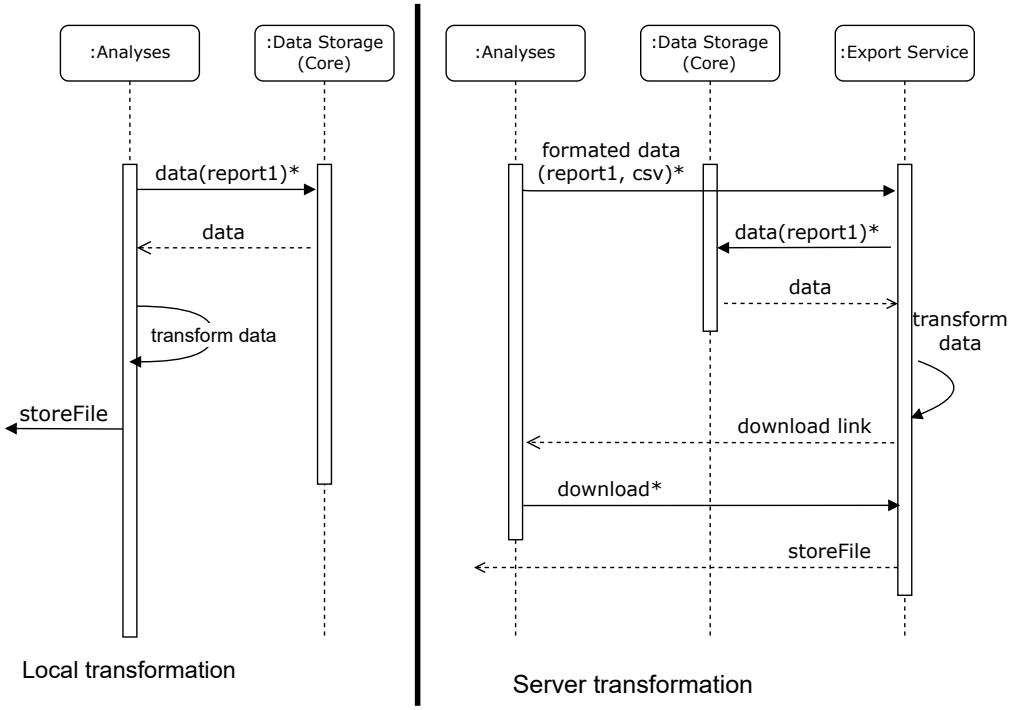

Figure 5.22: Data Export Variants

Since the reports/views should be interactive and easy to customize [F2, L4, L6], it should be possible to dynamically change the data displayed in a report. In order to do this, it should be possible to create filters per report that allow filtering of the data. When a user creates a new filter (on the report page), the user should be directed to a new page where the user can change the settings of the filter (see the left wireframe in Figure 5.23).

On this page, the user should be able to select the filter type, a display name, type-specific settings, and the filter match. The filter type specifies which value(s) the filter can handle and how the filter should be rendered. For example, a text filter will render

a text field, a multiple select text filter will render a drop-down menu, etc. This allows the user to create reports with all possible filter variants [F1, F2, L4]. In addition, the user should be able to select the (default) match filter. The match filter defines how the filter value should be compared to the actual data. For example, for a text filter, the user could select that the actual value should start with the filter text, etc. This is needed to make the filter(s) more flexible and data independent [F5]. For example, if a data column contains the street and house number, it would not be possible to select all entries for a street if the filter match setting were not available.

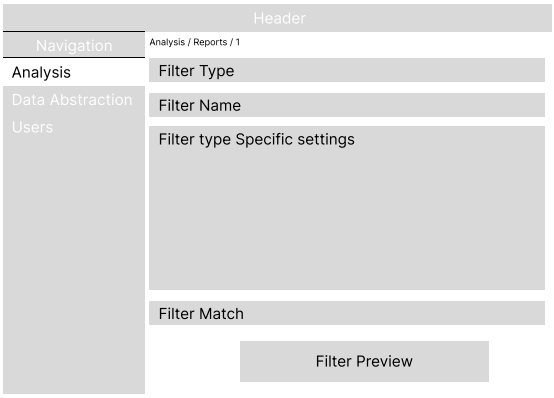

Figure 5.23: Application Analysis Filters (Wireframe)

In order to apply the filters to the used data sources [F5], it is necessary to select a column that should be checked by the filter, in this application this feature is called *Filter Mapping*. The application should provide a new page, which should be displayed when the filter is created, allowing the user to configure the mapping for each data source (see wireframe in Figure 5.24). This configuration should include the column to match and the filter match configuration, which will default to the previous setting. To simplify the UI [F1], the application should hide the data sources that are not currently used by default. This allows the user to change the mapping for unused data sources if they plan to use them in the future, while still making the configuration easier.

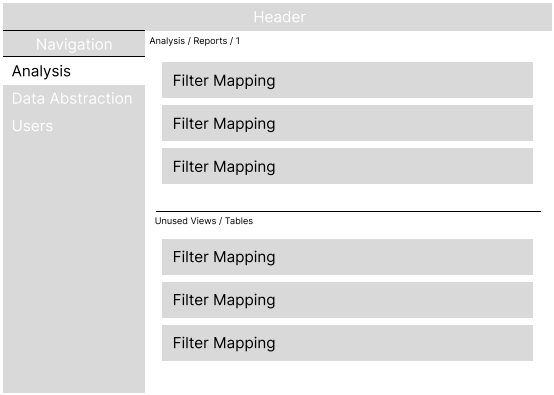

Figure 5.24: Application Analysis Filter Mapping (Wireframe)

When the data sources are then queried while the report is being rendered, the filter is evaluated based on the configuration. All existing filters for a data source configuration are concatenated with an *and* and then appended to the query (see Figure 5.25). If needed, there could also be an option to select how these filters would be concatenated, such as a combination of *and* and *or's* etc. Since this was not needed by the original application, it is outside the scope of this concept, but could easily be added.

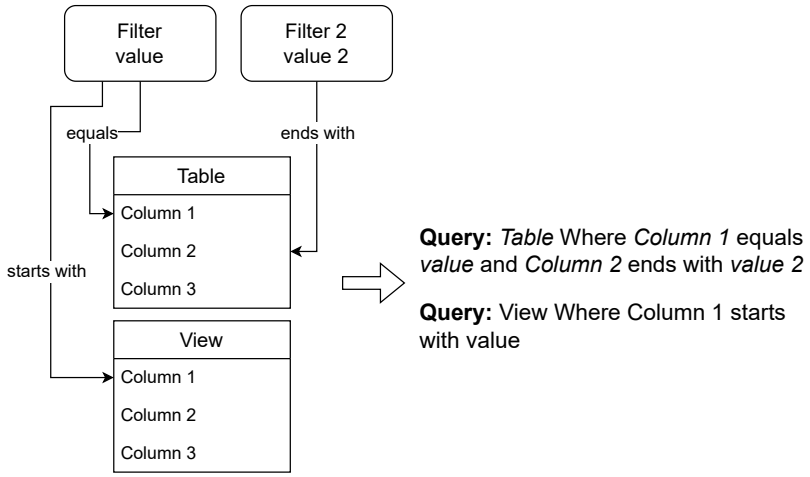

Figure 5.25: Application Analysis Filter Mapping example

To fulfill the requirement for time range queries [F7], the application should provide the special filter type *timeRange*. This filter type should allow the user to select a start and end date. It should also provide the ability to select the granularity as a filter match option. This granularity then determines the blocks into which the query data will be grouped, for example, if the data is available for every hour and the granularity is day, the query will return aggregated day values.

Data Abstraction

To configure the data abstraction [F5] described in Section 5.7 with a low-code approach [F1], the application should contain two pages. A page that shows an overview of all views and allows editing or delete them (left wireframe in Figure 5.26). On this page, there should also be a button to create new views, which leads to a new page (right wireframe in Figure 5.26). On this page, it is possible to select all the tables with their columns that should be included in the view. The application should restrict the selectable tables according to the restrictions described in Section 5.7. On the right side an overview should be displayed showing all selected tables with their columns [F1]. This page could also be reused to edit a view.

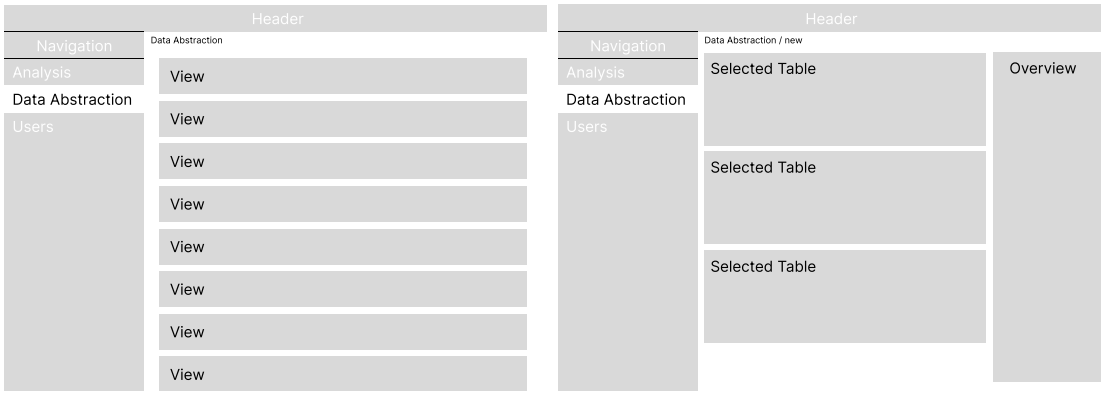

Figure 5.26: Application Data Abstraction (Wireframe)

Users

To manage users and system users with their roles/attributes (see Section 5.5), the application should provide a page for this purpose (see Figure 5.27) [F1]. On this page it should be possible to edit user information, create new users and see their last login, etc.

Figure 5.27: Application Users (Wireframe)

Calculations

To configure and monitor the calculations described above [F1, F8, F9], the application should have a calculations' page showing all calculations and their dependencies (see figure 5.28). It should also show when these calculations were last executed and provide the ability to view execution logs. This page should also provide the ability to create a new calculation, which will open in a new page.

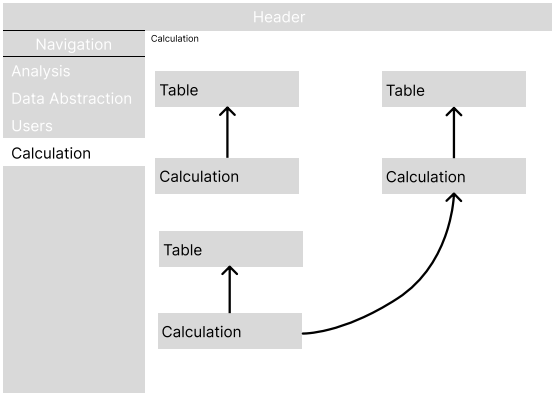

Figure 5.28: Application Calculations (Wireframe)

In order to specify all the settings required for the calculations (see Section 5.8.1), the creation page contains a stepper that guides the user through them (see Figure 5.29). The stepper makes it easier for the user to create a calculation [F1], as the page is not overloaded with all the information. The first step should allow the user to select the tables that will trigger the calculation when they change. The second step should allow the user to select additional data needed by the runner. After that, it should be possible to select which runner should perform the calculation and set up runner configurations. Finally, the user should be able to specify what and how to store the data that the runner outputs.

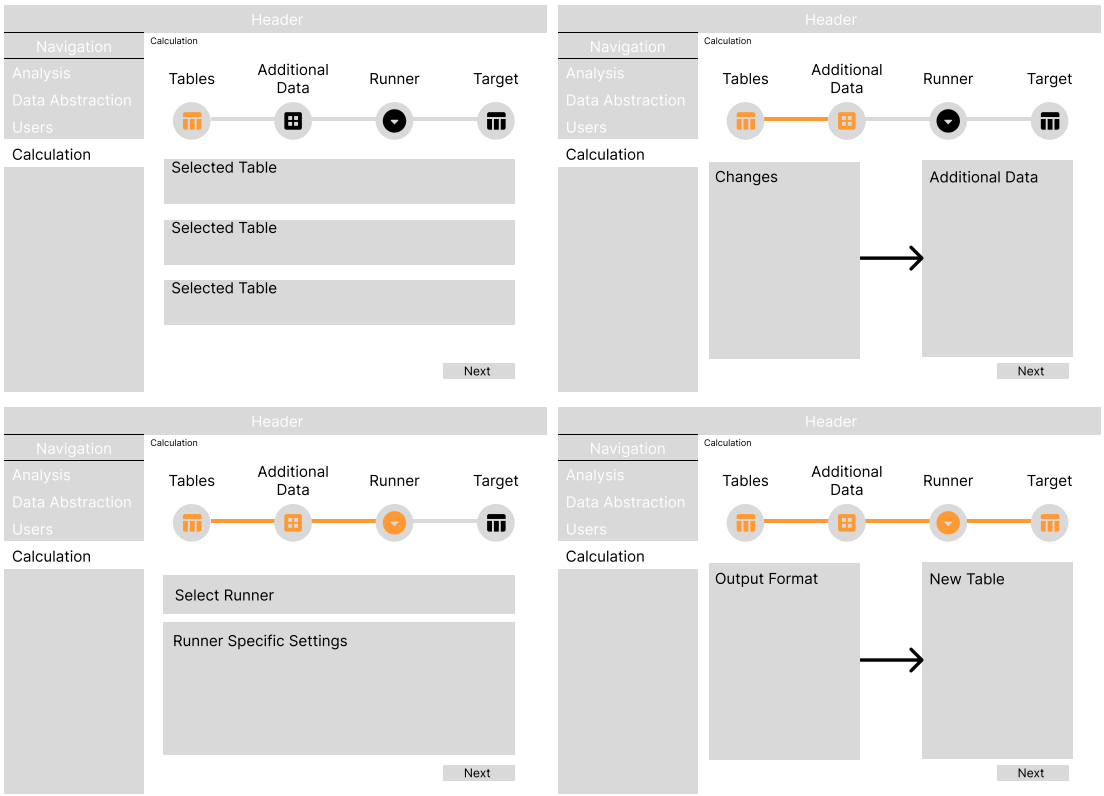

Figure 5.29: Application Calculation Creation (Wireframe)

To specify which runner can be selected in the previously described stepper [F1, F8, F9], the application should provide a page to view all runners and create new ones (see Figure 5.30). When creating a new runner, it should be possible to specify the image, the schema of the runner configuration and the schema of the output data.

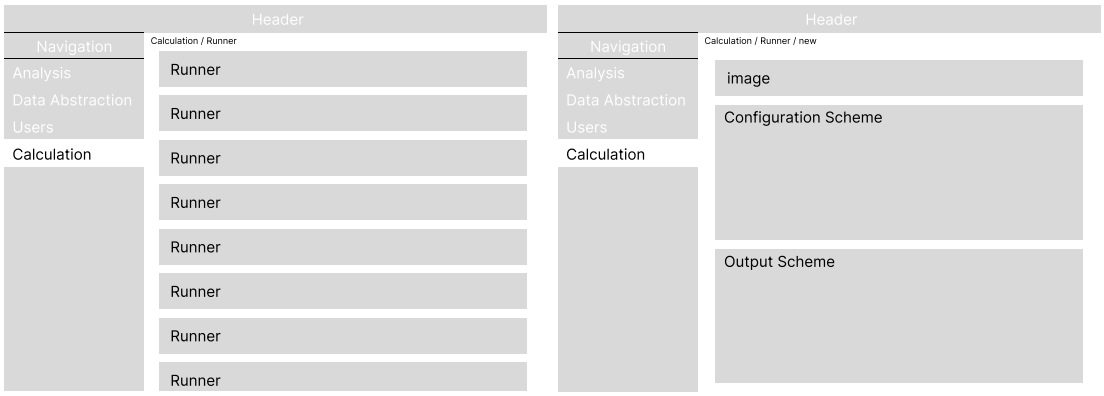

Figure 5.30: Application Runner (Wireframe)

5.8.3 Manual Data input

To meet the requirements of [F8] and [F9], it may be necessary to store some additional information for specific data in the database. For example, it may be necessary to store a formula for calculating the efficiency of a power plant, the mapping of the meters to the formula, etc. In such cases, it may be necessary to enter this data manually, as it is not already available in any other system. If this is required, the application should provide additional pages to insert this data by a non-technical person [F1].

5.9 Summary

This concept aims to construct an analytics platform that meets the specifications outlined in Section 3. It proposes a service-based architecture consisting of ten different services. For the interface, the gateway pattern is used, which should be implemented by the *Gateway* service. For authentication, the system should use self-contained tokens with refresh token rotation handled by the *Auth* service. In order to permit machine logins, this service should also provide the ability to login using a token. For authorization, the concept suggests using role-based access control or attribute-based access control enforced by the *Auth Policy*. To store metadata, the *Operator* service should provide a database with an automatically generated API. The external data storage should provide an API that provides data in groups of data (tables) with data types. It should also provide the ability to create virtual tables, for better data abstraction. In order to provide the ability to make complex and individual calculations,

the concept proposes a subsystem that handles these calculations. The subsystem should be built with an event-based architecture and consist of four services. It should include a service to detect data changes and generate change events. A service to listen to these events and generate events based on the configured jobs for these changes. To actually perform the computations, it should also contain a service that listens to the job events and computes the jobs with the configured container. For analyzing the data, the concept proposes to create an application that allows to configure the analytics platform and dynamically generate reports for analysis using a low-code approach. These reports consist of multiple views that are then built from multiple diagrams to achieve full customizability and reusability.

6 Implementation

This chapter describes the details of the implementation of the concept proposed in Chapter 5. The description does not include all the implementation details, only the difficult parts. This includes open source software choices that can be used to implement parts of the concept. It describes the implementation of an analytics platform based on the above concept for a local energy supplier, which uses this platform to monitor power plants. Thus, some parts and assumptions of the implementation may not be generalizable to other use cases in which the above concept could be used as well.

6.1 Interface

An existing open source solution is used to implement the *Gateway* service. Based on an Internet search, the following API gateways were considered for implementation: *KrakenD* [KRA24], *Kong* [Kon24], *gravitee.io* [Gra24], *Apache APISIX* [The24b], *GraphQL Mesh* [The24f], *Tyk* [Tyk24] and *Wundergraph* [Wun24].

The API gateways KrakenD, Kong, gravitee.io and Apache APISIX do not have the ability to provide a GraphQL API directly, but only to proxy GraphQL APIs and/or generate REST APIs from GraphQL APIs. These solutions are not suitable for this implementation and will not be considered in the future, as this would require all services to provide a GraphQL API in order to provide a general GraphQL API for the analytics platform, as described in the concept. Wundergraph has a similar problem as it can only combine multiple GraphQL APIs into one. [KRA24, Kon24, Gra24, The24b, Wun24]

Tyk and GraphQL Mesh both provide the functionality needed to generate a GraphQL API from multiple sources. GraphQL Mesh allows the use of 13 different source protocols while Tyk only allows 5 different ones, where all source protocols supported by Tyk are also supported by GraphQL Mesh. Tyk has the advantage that it also provides authentication, while in GraphQL Mesh this has to be implemented by the user. In terms of configuration, Tyk provides a graphical interface where the user can configure the endpoints and their mapping. This interface allows many configurations, but is very manual, as for example a mapping from REST to GraphQL cannot be generated automatically. GraphQL Mesh, on the other hand, generates the GraphQL API

fully automatically and allows to customize the generation via a YAML configuration. Therefore, the implementation uses GraphQL Mesh as the auto-generation of the API and the higher number of data sources are important factors to reduce the development time [Q5]. [Tyk24, The24f]

GraphQL Mesh is a framework that makes it easy to create GraphQL gateways without a lot of coding. It allows the combination of multiple GraphQL APIs and the automatic generation of GraphQL APIs for non-GraphQL sources. Listing 6.1 shows a sample configuration that generates a GraphQL endpoint based on a REST endpoint. Similar configurations are used to generate a unified GraphQL API for all services.

Listing 6.1: GraphQL Mesh Configuration Example

```
sources:
  - name: RestAPI
    handler:
      openapi:
        source: ./api/rest-api.json
```

6.2 API Security

6.2.1 Authentication

The *Auth* service is implemented in the *go* [Goo24a] programming language, due to the experience of the developers and its ability to build scalable distributed systems [Ana20]. This go service implements an API that takes the username and password or tokens on login and validates them. It handles the creation of authentication and refresh tokens, etc. To implement refresh token rotation, it gives each refresh token a family ID and a token ID and stores the family with ID in the database. On each refresh, it checks that the token's ID matches the ID stored in the db. If so, the ID in the database is incremented and a new token is generated. If not, the token has been used twice and the entry in the db is deleted to invalidate all tokens.

For the implementation of the self-contained tokens, this concept uses *JSON web tokens (JWTs)* as they are a standard format for self-contained security tokens. These tokens consist of a set of claims about a user, represented as a JSON object, together with a header describing meta-information and a signature to cryptographically protect the token against tampering. [Mad21] The *jwt-go* [gol24] library is used to generate (sign) and validate (parse) the tokens in go. To generate the token, there is a signing function that takes the secret used to sign the token and the claims that should be included in the token (see Listing 6.2). These claims include the current user with their roles and,

if the token is a refresh token, the token's ID and family. This function also adds an expiration time to the token based on the application configuration.

Listing 6.2: Sign JWTs in Go

```
1  func Sign(jwtSecret []byte, claims JWTClaims) (string, int64, error) {
2    claims, expirationTime := getClaims(claims)
3    t := jwt.NewWithClaims(jwt.SigningMethodHs256, claims)
4    s, err := t.SignedString(jwtSecret)
5
6    return s, expirationTime, err
7  }
```

To validate the tokens and obtain the claims, there is a parse function (see Listing 6.3). This function takes the secret to validate the token and the token itself and returns whether the token was valid and, if so, the claims it contains.

Listing 6.3: Parse JWTs in Go

```
1  func Parse(jwtSecret []byte, tokenString string) ( *JWTClaims, error) {
2    token, err := jwt.ParseWithClaims(tokenString, &JWTClaims{}, ...)
3
4    //...(handle error)
5
6    return token.Claims, nil
7  }
```

6.2.2 Authorization

To implement the *Auth Policy* service, an open source tool is used to evaluate whether the user should have access to certain GraphQL endpoints and fields. There are several existing open source tools for access control, based on an Internet search the following were selected for further investigation: *Open Policy Agent (OPA)* [Clo24d], *Casbin* [Cas24b], *Keto* [Ory24a], and *Oso* [Oso24b].

Since the open source version of Oso has been deprecated (no new features will be added and the software may be discontinued in the near future) in December 2023 [Oso24a], it is not a suitable option to use for a new software. There is still a cloud version available, but since the software should be hosted on-premises [Q6], this is not a viable option.

Keto is part of an ecosystem called Ory [Ory24c], which provides solutions for user management, authorization, access control, etc. In order to work securely, Keto needs a comparable API gateway, such as Ory Oathkeeper, since this application already uses

GraphQl Mesh as an API gateway, which is not a comparable API gateway, this tool cannot be used for this application. [Ory24b]

The main differences between Open Policy Agent (OPA) and Casbian are that Casbian is limited to supported access control models, while OPA only provides a policy language that allows the user to create any access control model, or even policies that have nothing to do with access control. This makes Casbian easier to implement and learn. Another difference is that OPA provides two ways to use it, in library mode or as an external service that can be called via REST, while Casbian only provides a library mode. Because of the higher flexibility and the option to run it as an external service, OPA was chosen for the implementation of the *Auth Policy* service. [Clo24c, Cas24a]

Open Policy Agent (OPA) is a policy engine that provides a declarative language for specifying policies as code and an API for querying them. In this way, OPA decouples policy decision-making from policy enforcement. To make a policy decision, a service queries OPA with some input data, while OPA then evaluates that input against the policies and stored data. The result of the evaluation is then returned to the service, which can then act on it (see Figure 6.1). [Clo24c]

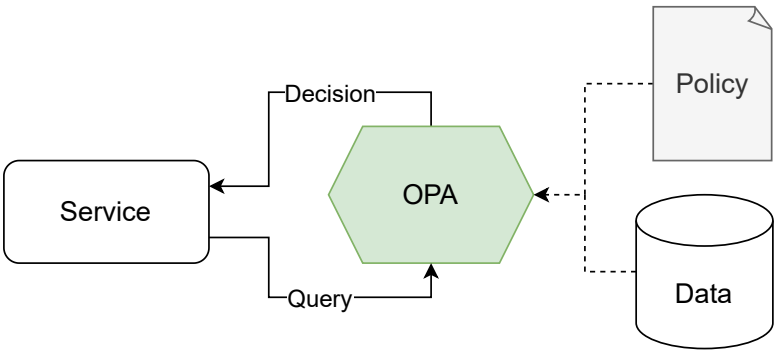

Figure 6.1: OPA Usage (based on [Clo24c])

OPA uses the *rego* language to define policies. Rego is a query language that allows policies to be defined that enumerate instances of data that are inconsistent with an expected system state. An example rego file implementing role-based authentication is shown in Listing 6.4. In this example, OPA has data that provides information about which user has which roles and which roles can access which object. So the service that wants to check authentication simply sends a query with the current user, the action the user wants to perform and the object on which the action is to be performed, and can check the *allow* variable in the OPA response. The rego definition simply loops through the stored roles and permissions and checks that the current permissions for the object match the existing ones. [Clo24c]

Listing 6.4: Role-based Access Control Example [Clo24c]

```
package rbac.authz

import rego.v1

# ...

# logic that implements RBAC.
default allow := false
allow if {
    # lookup the list of roles for the user
    roles := user_roles[input.user]
    # for each role in that list
    r := roles[_]
    # lookup the permissions list for role r
    permissions := role_permissions[r]
    # for each permission
    p := permissions[_]
    # check if the permission granted to r matches the users request
    p == {"action": input.action, "object": input.object}
}
```

6.3 Data Storage

To implement the *Operator* service, an existing database solution is used. Based on an Internet search, the following database solutions are considered: *Apache Cassandra* [The24a], *ArangoDB* [Ara24], *CouchDB* [The24d], *MariaDB/MySQL* [Mar24, Ora24], *PostgreSQL* [The24g] and *SQLite* [SQL24].

The database systems Apache Cassandra, ArangoDB and CouchDB are NoSQL databases, as the data that needs to be stored is structured and does not produce a very large amount of data, this database type does not offer any major advantage, but several disadvantages, as no relationships exist between the data. So these databases are not considered for the future. [The24a, Ara24, The24d]

SQLite is a lightweight database that is often used to store data for a single application, but not for large multiservice applications that use this database [SQL24]. It is therefore not suitable for this use case. In comparison, between PostgreSQL and MariaDB/MySQL, the former database performs better with large amounts of data and more complex SQL queries [EDB24]. PostgreSQL was therefore selected as the database for implementing the *Operator* service.

To generate an API for the database, the GraphQL Mesh API gateway described above is used, as it already provides the ability to generate a GraphQL API from a PostgreSQL database using Postgraphile. To improve the performance of complex operations, it allows the definition of custom functions at the database level, which also generate API endpoints. For example, if the user wanted to insert multiple records into the same table, the function shown in Listing 6.5 could be used.

Listing 6.5: Custom API Function Example [Pos24]

```
1 CREATE FUNCTION app_public.create_documents(num integer, type text,
      location text)
2 RETURNS SETOF app_public.document
3 AS $$
4   INSERT INTO app_public.document (type, location)
5     SELECT create_documents.type, create_documents.location
6     FROM generate_series(1, num) i
7     RETURNING *;
8 $$ LANGUAGE sql STRICT VOLATILE;
```

6.4 Analyze

6.4.1 Calculations

Message Broker

There are many message broker implementations such as *Redis* [Red24], *NATS* [Clo24b], *Apache Kafka* [Apa24], and *RabbitMQ* [Bro24a], all of which have their advantages and disadvantages. For the use case described in the concept, all of them could be used. Based on the developers' experience, NATS was used as the message broker.

NATS is an open source data layer, which provides publish and subscribe functionality for event based architectures. For persistent event queues, NATS provides something called *JetStream*. It allows the creation of multiple streams with multiple subjects where clients can publish or subscribe to events, which are called messages in NATS. Streams in NATS provide a way to organize events into groups. Each stream can contain multiple subjects, which allows future grouping of events. This makes it possible for a client to listen to all events from a stream or just events from some subjects in the stream. When configured this way, all events are persistent and are only deleted when they have been processed. A client can acknowledge a message, which tells NATS that the message has been processed and can be deleted. [Clo24b]

NATS provides specialized libraries for many programming languages, including a go library [Clo24a] used by the calculations services. To publish or subscribe to events, a connection to the NATS server must be established (see Listing 6.6). This connection can then be used to create streams and publish or subscribe to events. To use the JetStream described above, a new JetStream object has to be created from the NATS connection.

Listing 6.6: Create a NATS Connection

```
1  func Connect(url string) (jetstream.JetStream, error) {
2    nc, err := nats.Connect(url)
3    //...(handle error)
4    js, err := jetstream.New(nc)
5    //...(handle error)
6    return js, nil
7  }
```

Scheduler

The *Scheduler* service is, similar to the *Auth* service implemented in the go programming language. Once started, it always listens for data change events in NATS. When it receives changes, it uses the changes and the job information from the database to calculate which jobs should be executed. These jobs are then published as an event in a different subject (see Listing 6.7).

Listing 6.7: Process Changes

```
1  func ProcessAllMessages(js ..., sourceSubject ..., targetSubject ...){
2    consumer := createConsumer(sourceSubject)
3
4    for {
5      changes := FetchAllChanges(consumer)
6      jobs := TransformChanges(changes)
7      PublishJobs(js, targetSubject, jobs)
8    }
9  }
```

Since the concept requires the scheduler to group multiple changes that occur close together for performance reasons, a custom function is implemented to do this (see Listing 6.8). This function retrieves the maximum number of events that should be grouped together. Since this number is not always reached, it has a timeout that allows it to fetch only the number of events that are present in the subject.

Listing 6.8: Fetch all Changes

```
func FetchAllChanges(consumer ...){
  messages, err := consumer.fetch(
    MAX_BATCH_SIZE,
    jetstream.FetchMaxWait(MAX_WAITING_TIME)
  )
  //...(handle error)

  return MessagesToChanges(messages), nil
}
```

Since the NATS client does not provide a function to publish multiple events, Listing 6.9 shows a custom function to do this. It publishes all events asynchronously and waits for them all.

Listing 6.9: Publish Jobs

```
func PublishJobs(js ..., subject string, jobs Jobs) {
  for _, job := range jobs {
    js.PublishAsync(subject, JobToMsg(job))
  }
  <-js.PublishAsyncComplete()
}
```

Worker

The *Worker* service is also implemented as a go service, waiting for job events published by the *Scheduler* service. When it receives a job, it looks up the job configuration from the database, and if the configuration requires additional data, it also fetches that data from the *Data Storage (Core)* service. It then executes the job and writes the result back to the *Data Storage (Core)* service (see Listing 6.10).

Listing 6.10: Consume Jobs

```
func ConsumeJobs(subject string) {
  for {
    jobEvent := FetchJob(subject)
    job := GetJob(jobEvent.jobId)
    additionalData := AdditionalData{}

    if job.fetch {
      additionalData = FetchAdditonalData(job)
    }

    WriteData(job.Configuration, jobEvent.Changes, additionalData)
```

```
12        result, err := ExecuteJob(job.image)
13        //...(handle error)
14        WriteResult(result)
15      }
16    }
```

To run the job, the Docker container runtime is used. To interact with Docker, it provides a go library [Mob24] that is used by this service. Listing 6.11 shows the function that uses this library to spawn a container and wait until it is finished. In order to spawn the container, it must first pull the image of the job. It then cleans up the container and retrieves the results from the predefined file.

Listing 6.11: Execute Jobs

```
1   func ExecuteJob(image string) (Result, error){
2     client, err := client.NewClientWithOpts(...)
3     //...(handle error)
4     _, err := client.ImagePull(..., image)
5     //...(handle error)
6     container, err := client.ContainerCreate(...)
7     //...(handle error)
8
9     err = client.ContainerStart(..., container.ID, ...)
10    //...(handle error)
11    status, errChannel := client.ContainerWait(..., container.ID, ...)
12    select {
13      case err = <-errChannel:
14        //...(handle error)
15      case <-status:
16    }
17    client.ContainerRemove(..., container.ID, ...)
18
19    return GetResult(container.ID), nil
20  }
```

Runner

As the concept describes, the runner could be implemented in any programming language, it just needs to be executable via Docker. Since, as described in the concept, the configuration of the task and the data and result are passed via files, the program needs to know where to find these files and where to write the output file. To do this, the *Worker* service passes the location of the configuration file to the container as a command line argument. This configuration file then contains the location of the data file and the location where the result file should be written. Listing 6.12 shows an

example implementation of a runner written in go that reads the data, divides the values by two, and writes them back as the result.

Listing 6.12: Runner Example

```
func main() {
  configPath := os.Args[1]
  //...(handle error)
  config := LoadConfig(configPath)
  data := LoadData(config)
  for idx, d := range data {
    data[idx] := d / 2
  }
  WriteData(config, data)
}
```

In order to use this application as a runner, a docker image is required, and a *Dockerfile* can be used to create this image. Listing 6.13 shows this Dockerfile, which copies the source code, builds the runner, and configures docker to run it. Docker can then be used to create the image, which can then be used in the job configuration.

Listing 6.13: Runner Dockerfile

```
FROM golang:1.22.5-alpine3.20

WORKDIR /app

COPY . .
RUN go build -o ./example-runner

CMD [ "./example-runner" ]
```

6.4.2 Visualization

The application for viewing and configuring analytics is built as a web application. This makes it easier for all users in the company to access, since it does not need to be installed. It is easy to update and works on all devices and operating systems. [Tai08] Due to the experience of the developers, the web application is developed using the *TypeScript* [Mic24a] programming language and the *React* [Met24a] framework.

TypeScript is a programming language that extends the *JavaScript* programming language. The biggest advantage over JavaScript is that TypeScript brings a type system with it. Since web browsers cannot execute TypeScript code, it is compiled back to JavaScript before execution in order to use it for web applications. [Gol22]

React is a framework that aims to simplify web development. It allows the developer to create small reusable parts of the web application, called components. These components are defined using a special syntax called *JSX* (or *TSX* for TypeScript), which allows the combination of HTML and JavaScript/TypeScript code to specify the user interface (see Listing 6.14). In this syntax, regular HTML elements or other React components (starting with an uppercase letter) can be used in combination with JavaScript/TypeScript code. To pass information between components, props can be passed to a component. [Rip23]

Listing 6.14: React Component Example

```
interface TestComponentProps {
  text: string;
}

const TestComponent: React.FC<TestComponentProps> = ({ text }) => {
  return (
    <div>
      {text.toUpperCase()}
      <OtherComponent />
    </div>
  );
};
```

The next sections describe how to implement reports, views, and diagrams, as these are the most interesting parts of the web application. The next section describes a component that will be used in several examples, followed by an explanation of how data retrieval for reports/views works. Finally, the implementation for defining diagram and filter types is described.

Dynamic Inputs

Since the application often needs to define input masks with similar inputs, a dynamic input component was developed. This component allows easy definition of inputs, input validation, etc. without the need for much coding. The component takes a list of objects defining the required inputs and, based on these inputs, generates a corresponding output object containing all values entered by the user (see Listing 6.15 for an example).

Listing 6.15: Dynamic Input Definition Example

```
// Input definition
const inputs = [
{
  id: "name",
  type: "textField",
```

```
6       props: {
7         label: "Name",
8       },
9       required: true,
10      hasError: (currentValues) => ...
11      showIf: (currentValeus) => ...
12    },
13    //...
14  ];
15
16    // Example Output
17    {
18      name: "Max"
19    }
```

The input definition contains an ID, which defines the field where the output is written, and a type, defining which input element is rendered, e.g. *textField* for a simple text input field, or *select* for a selection box. To customize the rendering of the input element etc. it is also possible to define type specific props. To validate the input, two properties can be specified. If the value is required, an error will occur if the user leaves the input blank. The other properties allow the developer to specify a custom function that takes all current values and returns nothing or an error message to be shown to the user, allowing custom validation of the value. To define when an input field should be displayed, the developer can specify a custom *showIf* function that takes the current values and returns whether the input should be displayed. This can be used to show inputs only when other inputs have certain values. The implementation of the component ensures that these functions are re-validated when an input changes, so that the inputs are displayed as intended.

To render the input definitions, there is a React component (see Listing 6.16). In order to store the input values and detect changes, React needs a state to store them. This state is a special variable that allows React to detect changes. Since React can only pass properties to child components and not the other way around, this state must be defined in the component using the *DynamicInput*. Because this component needs the output of the values for further use.

Listing 6.16: *DynamicInput* Implementation

```
1   const inputTypes: ... = {
2     textField: DynamicInputTextField,
3     //...
4   };
5
6   const DynamicInput: ... = ({inputs, state, onChange}) => {
7     return (
8       <>
```

```
        {inputs.map((input) => {
          const InputElement = inputTypes[input.type];
          //...

          return (
            <InputElement ... />
          );
        })}
      </>
    );
  };
```

Listing 6.17 shows an example of using the component. A special function, called a *hook* in React, is used to get the state etc. for the component. This function takes the input definition and returns all the properties that the *DynamicInput* component needs. It also handles input changes, input validation, etc. It also returns the output values and if they are valid.

Listing 6.17: *DynamicInput* Usage

```
const DynamicInputUsage: React.FC<DynamicInputUsageProps> = () => {
  const { props, output, isValid } = useDynamicInput(inputs);

  return (
    <div>
      <DynamicInput {...props} />
      {/*Show Output: */ JSON.stringify(output)}
      {/*Show if Input is valid: */ isValid}
    </div>
  );
};
```

Data Access

Reports and views are implemented with their own React components. To reduce complexity and improve performance, the report (or view if it is rendered separately) should handle the data fetching for the backend to render its diagram(s). To do this, they should check which columns the diagrams need and combine them all, as multiple diagrams may need the same data. More on how the diagrams define which columns they need in the following section.

Diagrams

A generic approach is used to implement the diagrams types that can be displayed in a view, making it easier to add new diagram types [Q4]. To achieve this, there is an abstract base *DiagramType* class that defines the variables/functions that a diagram must implement in order to work. All diagram types must then implement this abstract class (see Figure 6.2). To explain the parts of the *DiagramType* class, the following sections show parts of this class with an example implementation.

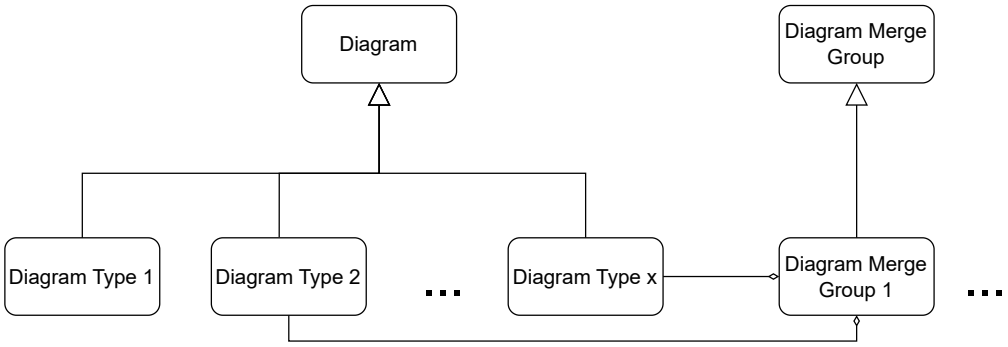

Figure 6.2: Diagram Classes

For implementation examples, a *TextDiagram* type is used. This *TextDiagram* type should render the given data into a prepared Markdown text (see Figure 6.3). To insert the data into the text, it should be possible to select the data keys with template strings.

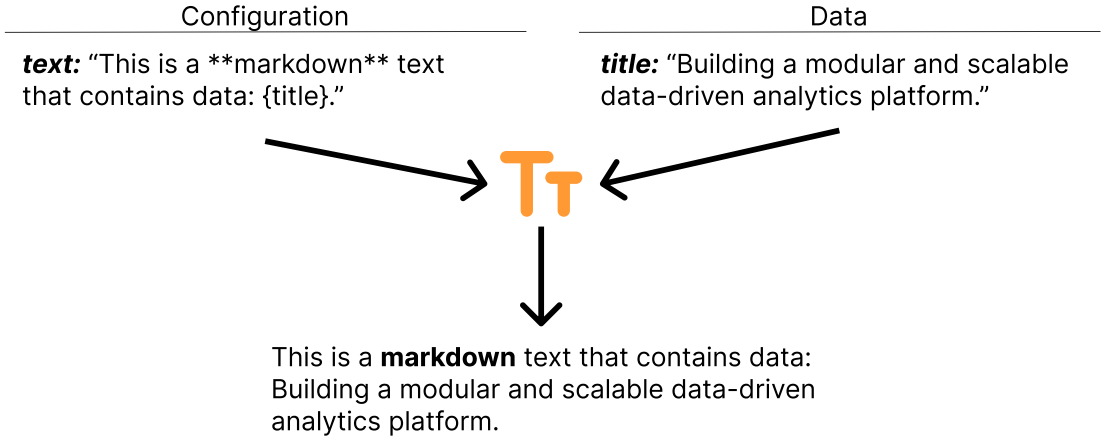

Figure 6.3: *TextDiagram* Example

Diagram Type Basic Settings

The *DiagramType* class (see Listing 6.18) has a *name*, a *description*, and an *icon* that is used to render the diagram type selector where the user can choose the type of the current diagram.

Listing 6.18: Abstract *DiagramType* Class

```
1  abstract class DiagramType<Configuration> {
2    abstract name: string
3    abstract description: string
4    abstract icon: Icon
5    //...
```

Diagram Type Configuration

To allow the user to change diagram-specific configurations, the *DiagramType* contains input definitions that are rendered when the diagram is edited. These input definitions are defined using the dynamic input definition described above (section 6.4.2). The class also contains a generic type parameter for the configuration, which must be provided by the implementation to define which configuration options are available.

Listing 6.19: abstract *DiagramType* Class

```
1  abstract class DiagramType<Configuration> {
2    //...
3    abstract configurationInputs: DiagramConfigurationInput[]
4    //...
```

The Listing 6.20 shows a sample implementation for configuring the *TextDiagram* class. It first defines the interface for the possible configurations and then an array with the input definitions to change these configurations.

Listing 6.20: *TextDiagram* Configuration

```
1  interface TextDiagramConfiguration {
2    text: string
3  }
4
5  class TextDiagram extends DiagramType<TextDiagramConfiguration> {
6    //...
7    configuration = [{
8      id: "text",
9      type: "textField",
10     props: {
11       label: "Text to Display"
```

```
12        }
13      }]
14      //...
```

Diagram Type Component

To render the diagram type, it must specify a React component which is rendered when the diagram is displayed. This component should follow a standardized interface (*DiagramTypeComponent*) that defines the properties of this component (see Listing 6.21). The properties include the data to be rendered and the current configuration. So the component itself should not care about how to fetch the data etc., this is all abstracted by the implementation that renders the diagrams.

Listing 6.21: Abstract *DiagramType* class

```
1  // DiagramType.ts
2  abstract class DiagramType<Configuration> {
3    //...
4    abstract component: React.FC<DiagramTypeComponent<Configuration>>
5    //...
6
7  // DiagramTypeComponent.ts
8  interface DiagramTypeComponent<Configuration> {
9    data: Record<string, unknown>[]
10   configuration: Configuration
11 }
```

Listing 6.22 shows the sample implementation of the *TextDiagram* component, which should be placed in a separate file for better readability. The sample component first retrieves the data and text to render from the props. It then inserts the data into the text by replacing predefined template strings. Finally, it passes the text to a Markdown renderer, which renders the text.

Listing 6.22: Example *TextDiagram* Component

```
1  // TextDiagram.ts
2  class TextDiagram extends DiagramType<TextDiagramConfiguration> {
3    //...
4    component: TextDiagramComponent
5    //...
6
7  // TextDiagramComponent.tsx
8  export const TextDiagramComponent:
9  React.FC<DiagramTypeComponent<Configuration>> = ({
10   configuration,
```

```
    data,
  }) => {
    const text = replaceTemplates(configuration.text, data)

    return <MarkdownRenderer text={text} />;
  };
```

Diagram Merge Group

As the concept describes, it should be possible to merge multiple diagrams into one. To implement this, an optional *MergeGroup* can be specified to determine if the diagram can be merged and to render the merged diagram (see Listing 6.23). The application prevents the user from creating multiple diagrams in a single view without a merge group or with different merge groups. Thus, if two diagrams have the same merge group, that group is used to render the diagrams, rather than the component of the diagram.

Listing 6.23: Abstract *DiagramType* Class

```
abstract class DiagramType<Configuration> {
  //...
  mergeGroup: string | undefined = undefined
  //...
```

The *MergeGroup* class itself defines a component that is used to render this *MergeGroup* (see Listing 6.24). This component then gets all the diagrams with their configuration and the fetched data. This can be done by using the original diagram classes to render them and display them on top of each other, for example, or by completely defining the component itself.

Listing 6.24: *MergeGroup* Class

```
// MergeGroup.ts
abstract class MergeGroup {
  abstract component: React.FC<MergeGroupComponentProps>
}

// MergeGroupComponentProps.ts
interface MergeGroupDiagram {
  type: string,
  renderComponent: (data: Record<string, unknown>[]) => JSX.Element;
  configuration: Configuration
}

interface MergeGroupComponentProps {
  data: Record<string, unknown>[]
```

```
15      diagrams: MergeGroupDiagram[]
16    }
```

Since merging a *TextDiagram* does not make much sense, the example used for the merge group is to merge multiple *ChartDiagrams*, such as bar chart and line chart diagrams (see Figure 6.4).

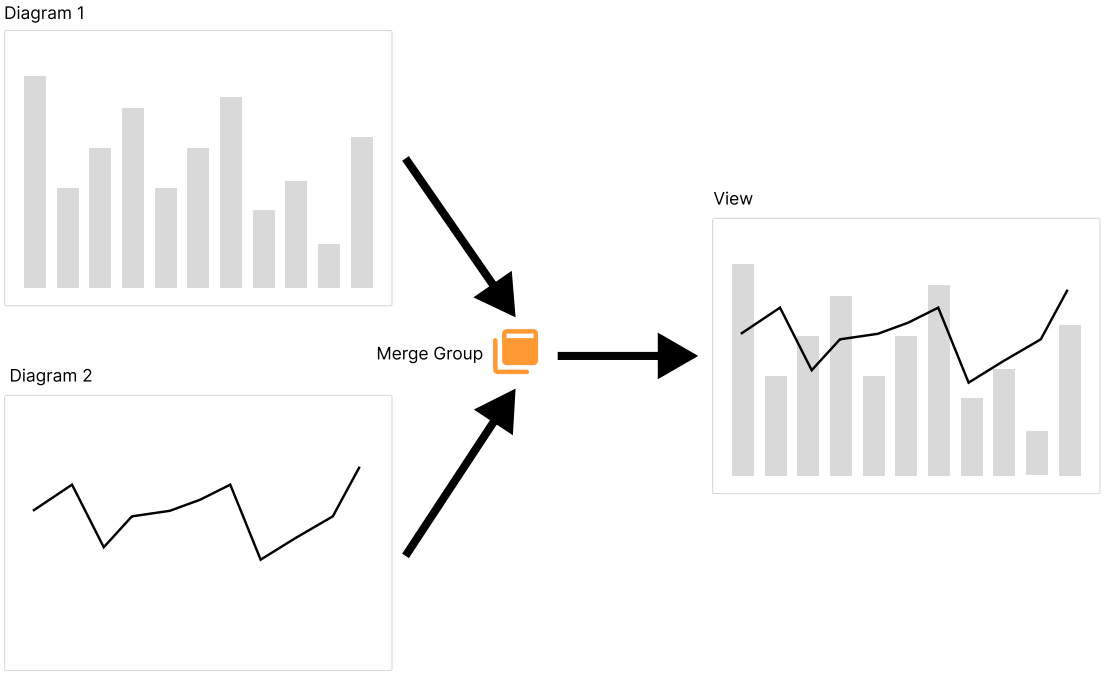

Figure 6.4: Merging Multiple Charts

To do this, the *ChartMergeGroup* component renders all the charts on top of each other to create a unified chart (see listing 6.25). It uses the components defined by the chart diagram types and CSS to place them on top of each other (for simplicity, the full CSS definitions for this are not shown in the example).

Listing 6.25: *MergeGroup* **Example**

```
1   // ChartMergeGroup.ts
2   class ChartMergeGroup {
3     component = ChartMergeGroupComponent
4   }
5
6   // ChartMergeGroupComponent.tsx
7   const ChartMergeGroupComponent: React.FC<MergeGroupComponentProps> = ({
8     diagrams,
9     data,
10  }) => {
11    return (
12      <div className="renderOnTopGroup">
```

```
13          {diagrams.map((d) => (
14            <div className="renderOnTopComponent">
15              {d.renderComponent(data)}
16            </div>
17          ))}
18        </div>
19      );
20  };
```

Columns Used in the Diagram

Since the diagram itself should not care about fetching data, it must implement a function that returns all columns that should be fetched from the backend (see Listing 6.26). This cannot be done generally because this information is dynamic and determined by the configuration.

Listing 6.26: Abstract *DiagramType* Class

```
1  abstract class DiagramType<Configuration> {
2    //...
3    abstract getUsedColumns(configuration: Configuration): string[]
4  }
```

The example implementation is shown in Listing 6.27. It extracts the string templates from the text and returns them since they represent the columns needed by this diagram.

Listing 6.27: Example *getUsedColumns* Implementation

```
1  class TextDiagram extends DiagramType<TextDiagramConfiguration> {
2    //...
3    getUsedColumns(configuration: TextDiagramConfiguration): string[] {
4      return getTemplateStrings(configuration.text)
5    }
6  }
```

Register Diagram Types

Since the application needs to know what diagram types exist and find their configurations, diagram types must be registered. For that, all types are defined in a global record with their ID as key and the class as value, as shown in the Listing 6.28. To register a merge group, there is a similar record with the merge groups.

Listing 6.28: Register a Diagram Type or Merge Group

```
export const diagramTypes: Record<string, DiagramType> = {
  textDiagram: TextDiagram,
  //...
}

export const megeGorups: Record<string, MergeGroup> = {
  chartMergeGroup: ChartMergeGroup,
  //...
}
```

Filter

Similar to the diagram implementation, the filters' implementation uses a generic approach. There is an abstract base filter class that defines the functions/variables needed for a filter, which can then be implemented by the specific filter classes (see Figure 6.5).

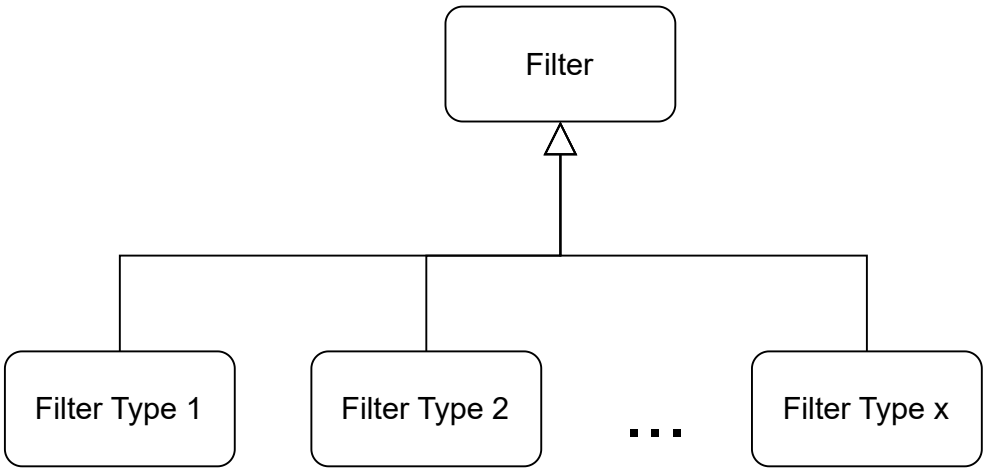

Figure 6.5: Filter Classes

For implementation examples a *TextFilter* is used. The *TextFilter* should render a text box that allows the user to enter any text that will be used to filter the data. For type selection, the filter class should contain basic information such as a name, description, and icon (see Listing 6.29).

Listing 6.29: Abstract *Filter* Class

```
abstract class Filter<Configuration> {
  abstract name: string
  abstract description: string
```

```
4     abstract icon: Icon
5     //...
```

To configure the filter upon creation, the filter should define the input files that should be rendered for filter-specific configurations and the *matchTypes*. This can also be done using the dynamic input definition described above (section 6.4.2). The *matchTypes* define how the filter values should be compared, as described in the concept, the filter class defines the allowed match types as an object with an ID and a label. These are then displayed in a select field when the filter is created.

Listing 6.30: Abstract *Filter* Class

```
1     //...
2     abstract configuration: FilterConfigurationInput[]
3     abstract matchTypes: FilterMatchTypes[]
4     //...
```

A sample implementation of the *TextFilter* is shown in Figure 6.31. It defines a configuration to specify the label to be displayed when the filter is rendered. It also defines the match type equals, which checks for complete equality.

Listing 6.31: Example *TextFilter* Class

```
1   class TextFilter extends Filter<TextFilterConfiguration> {
2     //...
3     configuration = [
4       {
5         id: "label",
6         type: "textField",
7         props: {
8           label: "Label of the Filter",
9         },
10      },
11    ];
12    matchTypes = [
13      {
14        id: "equals",
15        label: "Equals",
16      },
17      /...
18    ];
19    //...
```

The Dynamic Inputs configuration is also used to define the component that renders the filter (see Listing 6.32). But instead of a list of inputs, it defines only a single input. Since not all the information needed for the dynamic input is available when defining

the filter type, such as the input's ID, this information can be omitted. This information is added when the filter is rendered, and all filters from a report are combined into a single dynamic input. The previously described filter-specific configurations are also inserted as properties of the dynamic input, so that they can be used to customize the rendered filter.

Listing 6.32: Abstract *Filter* class

```
1     //...
2     abstract component: FilterConfigurationInput
3   }
```

The sample implementation for the *TextFilter* component, shown in Figure 6.31, contains only one type of input, since all other information is inserted via the filter-specific configuration options.

Listing 6.33: Example *TextFilter* Class

```
1   class TextFilter extends Filter<TextFilterConfiguration> {
2     //...
3     component = {
4       type: "textField"
5     };
6   }
```

7 Evaluation

The *Architecture Analysis Method (SAAM)* by Kazman et al. [Kaz96], which uses scenarios to evaluate software architectures, is used to evaluate the proposed concept. SAAM has five steps for evaluating an architecture. First, the architecture of the system should be described, which was done in the previous chapter. Secondly, scenarios are created that illustrate all types of system activities or future activities. The third step is to evaluate the scenarios. For each scenario, it is determined whether the architecture can execute it directly or whether a change is needed (indirectly). For indirect scenarios, the required changes and their costs are identified. The fourth step is to determine the number of indirect scenarios that affect the same service, called interactions. SAAM favors architectures with the fewest scenario conflicts. The final step includes an overall evaluation that weights the different scenarios and interactions to calculate an overall ranking. [Kaz96]

The scenarios were developed by employees of the energy supplier. They have been created for two roles, the normal users who use the graphical user interface and the administrators. The scenarios with their direct or indirect assessment and potential changes are listed in Table 7.1 and Table 7.2. As the tables show, 78% of the scenarios can be handled directly with the current approach, and only four require system changes.

	Description	Direct	Changes
1	Addition of a new user to the system.	✓	
2	Addition of a new system user to the system.	✓	
3	Change a user's permissions.	✓	
4	Add a new runner to the system.	✓	
5	Add a new single sign-on login method.	✗	To add a new SSO login method, the *Auth* service must be modified.
6	Connect a new external system to the application.	✓	

Table 7.1: SAAM Scenarios (administrators)

	Description	Direct	Changes
1	View stored data in a table.	✓	
2	Visualize stored data in a diagram.	✓	
3	Visualize stored data in a diagram type that is not included in the diagram type selection.	✗	To add a new diagram type, the service *Analysis* must be modified by adding the new diagram type.
4	Display of a table with a diagram.	✓	
5	Export the displayed data to a CSV file.	✓	
6	Filter the data displayed in a report.	✓	
7	Reuse views from one report on another report.	✓	
8	Share a report with a co-worker.	✓	
9	Display data from several tables in a single view.	✓	
10	Calculate metrics based on data changes.	✓	
11	Calculate metrics with a non-existent runner based on data changes.	✗	To calculate new metrics for a non existent runner, a new runner must be programmed to perform the calculation.
12	Receive notifications of erroneous or missing values.	✗	To send notifications for erroneous or missing values, the *Analysis* service must be modified to configure the notifications, and the *Scheduler* must be modified to detect and send them.

Table 7.2: SAAM Scenarios (users)

The interactions and number of changes between indirect scenarios and services are shown in Table 7.3. Only one in ten services is affected by interactions, and only three services and a new runner require changes to fulfill all scenarios. Since Kazman et al. [Kaz96] flavor architectures with few conflicts, this is a reasonable result.

Service	Number of Changes	Interactions
Analysis	2	1
Scheduler	1	-
Auth	1	-
Runner	1	-

Table 7.3: SAAM Scenario Interactions

The evaluation shows that the concept can fulfill most of the scenarios without any changes. Only four changes were required, with only a single intersection created. The service with the most changes is the *Analysis* service. The limitation is that new chart types cannot be added to this service without code changes, which would then require a software update to deliver these changes. The other limitation is that it is not possible to configure alerts or notifications.

The other two services with only one change are the *Scheduler* service and the *Auth* service. This is because the *Scheduler* service does not have the ability to monitor changes for notifications. And if a new SSO provider is required, the *Auth* service must be adopted. The other service that requires changes is the runner, which does not directly require changes to an existing service, but may require the addition of a new service.

The two main limitations are that the *Analysis* and *Scheduler* services are limited to notifications and that it is not possible to add new chart types more easily. The limitation of the *Auth* service is not so important as it is not often necessary to add a new SSO provider. And even if it were needed more often, the *Auth* service could be built to make this change easier. The runner limitation is also not very important, as this limitation is intended to give the source system more flexibility. Since a graphical interface to create all kinds of calculations would be very complex, making it unusable for a normal user, or incomplete. This limitation is further reduced by the runner configuration options described in the concept, as this also allows generic runners to calculate multiple tasks without code changes.

8 Conclusion

8.1 Discussion

In order to achieve the objective of creating a concept from an analytics platform with complex requirements, a number of requirements were first derived. These requirements are based on the reference system of a local energy supplier and existing analytics platforms. The existing systems were then evaluated, with the result that none of them could fulfill the requirements, or only with too many limitations or high costs. Subsequently, the relevant literature was presented and the concept and ideas were derived from it, which were later considered for the concept. The concept was then presented, consisting of a service-based architecture with ten services. To make the platform usable by non-technical people, this includes an application that allows reports and measures to be created using a low-code approach. This includes reports for viewing and analyzing data, pages for creating data abstractions, and user and configuration settings for calculating metrics. In order to provide full customization for complex metrics, a special subsystem called Calculations has been proposed. This subsystem detects data changes and uses a container runtime to calculate the new metrics with custom code. Then, the challenging parts of the local utility's reference implementation are described. This includes software choices for services that can be implemented by an existing system and implementation details of some services. Finally, an evaluation based on the SAAM module using scenarios was conducted.

As this evaluation shows that the concept fulfills most of the scenarios with only a few limitations, these limitations are presented in the next section. The concept allows companies to develop an analytics platform for complex use cases. Which can be used by non-technical people. It is scalable and modular and uses open source components to reduce development costs. The concept is very flexible and allows all kinds of reports/dashboards and metrics to be calculated. The metrics calculation also includes machine learning algorithms, making it future-proof as they are increasingly used in the analytics context. This thesis also includes implementation hints and software choices for some services, allowing easier adoption and implementation of the proposed concept.

8.2 Limitations and Future Research

As discovered in Chapter 7, the main limitations of the concept lie in the *Analysis* and *Scheduler* services. For the *Analysis* service, one limitation is that diagram types cannot be added without code changes and re-deployment of the application. However, this is not so serious because the modularity shown in the implementation chapter makes it very easy to add new diagram types in the source code, and this is also possible without extensive knowledge of the entire application. To overcome this limitation, future research could consider creating a diagram type builder UI that allows dynamic code to be added, or a plugin system that allows the diagram types to be extended dynamically at runtime. Another limitation and future research point is to allow the application to monitor changes for notifications of problems etc. This should be easy to incorporate into the system as changes are already detached, and it would only require a page to configure the notifications and changes to the *Scheduler* service which checks the conditions and sends the notifications.

Bibliography

[Aci14] Acito, Frank and Khatri, Vijay: Business analytics: Why now and what next? *Business Horizons* (2014), vol. 57(5):pp. 565–570

[Ama24] Amazon Web Services, Inc.: AWS Website (2024), URL `https://aws.amazon.com`, accessed: 03-08-2024

[Amp24] Amplitude Inc.: Everything You Need to Know About Data Analytics Platforms (2024), URL `https://amplitude.com/explore/data/data-analytics-platform#what-data-analytics-platform`, accessed: 06-08-2024

[AN15] Al Nuaimi, Eiman; Al Neyadi, Hind; Mohamed, Nader and Al-Jaroodi, Jameela: Applications of big data to smart cities. *Journal of Internet Services and Applications* (2015), vol. 6(1)

[Ana20] Anagnostopoulos, Achilleas: *Hands-On Software Engineering with Golang*, Packt Publishing, Limited, Birmingham (2020)

[Apa24] Apache Software Foundation: Apache Kafka Website (2024), URL `https://kafka.apache.org`, accessed: 28-07-2024

[Apo24a] Apollo Graph Inc.: Apollo Client (2024), URL `https://www.apollographql.com/docs/react`, accessed: 02-08-2024

[Apo24b] Apollo Graph Inc.: Apollo Server (2024), URL `https://www.apollographql.com/docs/apollo-server`, accessed: 02-08-2024

[Ara24] ArangoDB: ArangoDB Website (2024), URL `https://arangodb.com`, accessed: 24-07-2024

[Ave23] Aveiro, David; Mendes, João; Pinto, Duarte and Freitas, Vítor: A Comparative Analysis of Open-Source Business Intelligence Platforms for Integration with a Low-Code Platform, in: *International Conference on Information Systems Development*, ISD 2023, Instituto Superior Técnico

[Ben06] Benantar, Messaoud (Editor): *Access Control Systems*, SpringerLink, Springer Science+Business Media, Inc, Boston, MA (2006)

[Ben10] Benlian, Alexander and Hess, Thomas: Comparing the relative importance of evaluation criteria in proprietary and open-source enterprise application software selection - a conjoint study of ERP and Office systems: Comparing evaluation criteria for proprietary and OS EAS. *Information Systems Journal* (2010), vol. 21(6):pp. 503–525

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

[Bos09] Bose, Ranjit: Advanced analytics: opportunities and challenges. *Industrial Management Data Systems* (2009), vol. 109(2):pp. 155–172

[Bri20] Brito, Gleison and Valente, Marco Tulio: REST vs GraphQL: A Controlled Experiment, in: *2020 IEEE International Conference on Software Architecture (ICSA)*, IEEE

[Bro14] Brown, Simon: *Software Architecture for Developers*, Software Design & Development Conference 2014 (2014)

[Bro24a] Broadcom: RabbitMQ Website (2024), URL `https://www.rabbitmq.com/`, accessed: 05-07-2024

[Bro24b] Brown, Simon: The C4 model for visualising software architecture (2024), URL `https://c4model.com/`, accessed: 06-03-2024

[Cas24a] Casbin Organisation: Casbin Documentation (2024), URL `https://casbin.org/docs`, accessed: 24-07-2024

[Cas24b] Casbin Organisation: Casbin Website (2024), URL `https://casbin.org`, accessed: 24-07-2024

[Cha21] Chapple, Mike: *Access control and identity management*, Information systems security & assurance series, Jones & Bartlett Learning, Burlington, MA, 3rd edn. (2021), revision of: Access control, authentication, and public key infrastructure / Bill Ballad, Tricia Ballad, and Erin K. Banks. 2014

[Cho18] Choi, Tsan-Ming; Wallace, Stein W. and Wang, Yulan: Big Data Analytics in Operations Management. *Production and Operations Management* (2018), vol. 27(10):pp. 1868–1883

[Clo24a] Cloud Native Computing Foundation: NATS go client (2024), URL `https://github.com/nats-io/nats.go`, accessed: 28-07-2024

[Clo24b] Cloud Native Computing Foundation: NATS Website (2024), URL `https://nats.io`, accessed: 28-07-2024

[Clo24c] Cloud Native Computing Foundation: Open Policy Agent Documentation (2024), URL `https://www.openpolicyagent.org/docs`, accessed: 20-07-2024

[Clo24d] Cloud Native Computing Foundation: Open Policy Agent Website (2024), URL `https://www.openpolicyagent.org`, accessed: 20-07-2024

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

[Dgr24] Dgraph: Dgraphph Website (2024), URL `https://dgraph.io/`, accessed: 04-07-2024

[Doc24] Docker Inc: Docker Website (2024), URL https://www.docker.com/, accessed: 06-03-2024

[Dud20] Dudjak, Mario and Martinović, Goran: An API-first methodology for designing a microservice-based Backend as a Service platform. *Information Technology And Control* (2020), vol. 49(2):pp. 206–223

[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

[Eng24] Engineering: Knowage Website (2024), URL https://www.knowage-suite.com, accessed: 03-08-2024

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

[Gar22] Gartner: Low-code development platform market size worldwide 2024. *Statista* (2022), URL https://www.statista.com/statistics/1226179/low-code-development-platform-market-revenue-global/

[Gol22] Goldberg, Josh: *Learning TypeScript*, O'Reilly Media, Beijing (2022)

[gol24] golang-jwt maintainers: jwt-go Website (2024), URL https://github.com/golang-jwt/jwt

[Goo24a] Google LLC: Go Website (2024), URL https://go.dev/, accessed: 19-07-2024

[Goo24b] Google LLC: Google Cloud Website (2024), URL https://cloud.google.com, accessed: 03-08-2024

[Gou02] Gourley, David: *HTTP*, Definitive Guides, O'Reilly Media, Sebastopol (2002)

[Gou22] Gough, James: *Mastering API architecture*, O'Reilly, Beijing, 1st edn. (2022)

[Gra24] Gravitee.io: gravitee.io Website (2024), URL https://www.gravitee.io, accessed: 29-07-2024

[Gun23] Gunklach, Jonas; Jacob, Katharina and Michalczyk, Sven: Beyond Dashboards? Designing Data Stories for Effective Use in Business Intelligence and Analytics

[Hu15] Hu, Vincent C.; Kuhn, D. Richard; Ferraiolo, David F. and Voas, Jeffrey: Attribute-Based Access Control. *Computer* (2015), vol. 48(2):pp. 85–88

[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/

[Jin18] Jin, Brenda: *Designing Web APIs*, O'Reilly, Beijing, 1st edn. (2018)

[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

[Kha22] Khatuwal, Vishnu Singh and Puri, Digvijay: Business Intelligence Tools for Dashboard Development, in: *2022 3rd International Conference on Intelligent Engineering and Management (ICIEM)*, IEEE

[KNI24a] KNIME: KNIME Hub Pricing (2024), URL https://www.knime.com/knime-hub-pricing, accessed: 04-08-2024

[KNI24b] KNIME: Knime Website (2024), URL https://www.knime.com, accessed: 03-08-2024

[Kon24] Kong Inc.: Kong Website (2024), URL https://konghq.com, accessed: 29-07-2024

[KRA24] KRAKEND S.L.: KrakenD Website (2024), URL https://www.krakend.io/, accessed: 29-07-2024

[Law21] Lawi, Armin; Panggabean, Benny L. E. and Yoshida, Takaichi: Evaluating GraphQL and REST API Services Performance in a Massive and Intensive Accessible Information System. *Computers* (2021), vol. 10(11):p. 138

[Lin24] LinceBI: LinceBI Website (2024), URL https://www.lincebi.com, accessed: 03-08-2024

[Lod24] Lodderstedt, Torsten; Bradley, John; Labunets, Andrey and Fett, Daniel: OAuth 2.0 Security Best Current Practice, Internet-draft, Internet Engineering Task Force (2024), URL https://datatracker.ietf.org/doc/draft-ietf-oauth-security-topics/29/

[Mad21] Madden, Neil: *API Security in Action*, Manning Publications, 1st edn. (2021)

[Mar24] MariaDB Foundation: MariaDB Website (2024), URL https://mariadb.org/, accessed: 24-07-2024

[Mat22] Matigo: How long can $DISPLAY environment variable value be? (2022), URL https://askubuntu.com/a/1385554, accessed: 10-08-2024

[Men22] Menon, Pradeep: *Data Lakehouse in Action*, Packt Publishing Limited, Birmingham, 1st edn. (2022)

[Met24a] Meta Open Source: React Website (2024), URL https://react.dev/, accessed: 19-07-2024

[Met24b] Metabase: Metabase Website (2024), URL https://www.metabase.com/, accessed: 03-08-2024

[Mic24a] Microsoft: TypeScript Website (2024), URL https://www.typescriptlang.org/, accessed: 19-07-2024

[Mic24b] Microsoft Corporation: Microsoft Azure Website (2024), URL https://azure.microsoft.com, accessed: 03-08-2024

[Mob24] Moby Project: Docker go client (2024), URL https://github.com/moby/moby/tree/master/client, accessed: 28-07-2024

[New15] Newman, Sam: *Microservices*, mitp Verlags, Germany, 1st edn. (2015), translation of: Building Microservices, published by O'Reilly Media (2015)

[NKK22] Nyoman Kutha Krisnawijaya, Ngakan; Tekinerdogan, Bedir; Catal, Cagatay and Tol, Rik van der: Data analytics platforms for agricultural systems: A systematic literature review. *Computers and Electronics in Agriculture* (2022), vol. 195:pp. 106–813

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

[Ora24] Oracle: MySQL Website (2024), URL https://www.mysql.com, accessed: 24-07-2024

[Ory24a] Ory: Keto Website (2024), URL https://github.com/ory/keto, accessed: 24-07-2024

[Ory24b] Ory: Ory Self-hosting Documentation (2024), URL https://www.ory.sh/docs/ecosystem/projects, accessed: 24-07-2024

[Ory24c] Ory: Ory Website (2024), URL https://www.ory.sh/, accessed: 24-07-2024

[Oso24a] Oso Security, Inc.: Oso Deprecation notice (2024), URL https://www.osohq.com/docs/oss/getting-started/deprecation.html, accessed: 24-07-2024

[Oso24b] Oso Security, Inc.: Oso Website (2024), URL https://github.com/osohq/oso, accessed: 24-07-2024

[Pan06] Pankaj, Pankaj; Hyde, Micki and Rodger, James: Business Dashboards-Challenges and Recommendations, in: *AMCIS 2006 Proceedings*, p. 184

[Pel24] Pellekoorne, Timon: *Building a Low-Code Platform for versatile Data Integration*, Master's thesis, Technische Hochschule Mittelhessen (University of Applied Science) (2024)

[Pfl24] Pfleeger, Charles P.: *Security in computing*, Addison-Wesley, Boston, 6th edn. (2024)

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

[Pos24] PostGraphile: PostGraphile Website (2024), URL https://www.graphile.org/postgraphile/, accessed: 07-04-2024

[Rag14] Raghupathi, Wullianallur and Raghupathi, Viju: Big data analytics in healthcare: promise and potential. *Health Information Science and Systems* (2014), vol. 2(1)

[Red24] Redis: Redis Website (2024), URL https://redis.io/, accessed: 28-07-2024

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

[Rid19] Ridzuan, Fakhitah and Wan Zainon, Wan Mohd Nazmee: A Review on Data Cleansing Methods for Big Data. *Procedia Computer Science* (2019), vol. 161:pp. 731–738, the Fifth Information Systems International Conference, 23-24 July 2019, Surabaya, Indonesia

[Rip23] Rippon, Carl: *Learn React with TypeScript*, Packt Publishing Limited, Birmingham, 1st edn. (2023)

[SAP24] SAP Deutschland SE & Co. KG: Sap Website (2024), URL `https://www.sap.com`, accessed: 03-08-2024

[Shr24] Shrivastava, Saurabh: *Solutions architect's handbook*, Expert insight, Packt Publishing Ltd., Birmingham, UK, 3rd edn. (2024)

[Sin14] Singh, Dilpreet and Reddy, Chandan K: A survey on platforms for big data analytics. *Journal of Big Data* (2014), vol. 2(1)

[Sno24a] Snowflake Inc.: Snowflake Pricing (2024), URL `https://docs.snowflake.com/en/user-guide/cost-understanding-overall`, accessed: 04-08-2024

[Sno24b] Snowflake Inc.: Snowflake Website (2024), URL `https://www.snowflake.com`, accessed: 03-08-2024

[SQL24] SQLite Consortium: SQLite Website (2024), URL `https://www.sqlite.org`, accessed: 24-07-2024

[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

[Tai08] Taivalsaari, Antero; Mikkonen, Tommi; Ingalls, Dan and Palacz, Krzysztof: Web Browser as an Application Platform, in: *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, IEEE, pp. 293–302

[The24a] The Apache Software Foundation : Apache Cassandra Website (2024), URL `https://cassandra.apache.org`, accessed: 24-07-2024

[The24b] The Apache Software Foundation: Apache APISIX Website (2024), URL `https://apisix.apache.org`, accessed: 29-07-2024

[The24c] The Apache Software Foundation: Apache Superset Website (2024), URL `https://superset.apache.org`, accessed: 03-08-2024

[The24d] The Apache Software Foundation: CouchDB Website (2024), URL `https://couchdb.apache.org/`, accessed: 24-07-2024

[The24e] The GraphQL Foundation: GraphQL Specification (2024), URL `https://spec.graphql.org/`, accessed: 25-04-2024

[The24f] The Guild: GraphQL Mesh Website (2024), URL `https://the-guild.dev/graphql/mesh`, accessed: 28-07-2024

[The24g] The PostgreSQL Global Development Group: PostgreSQL Website (2024), URL `https://www.postgresql.org/`, accessed: 24-07-2024

[Tis19] Tisi, Massimo; Mottu, Jean-Marie; Kolovos, Dimitrios; de Lara, Juan; Guerra, Esther; Ruscio, Davide Di; Pierantonio, Alfonso and Wimmer, Manuel: Lowcomote: Training the Next Generation of Experts in Scalable Low-Code Engineering Platforms (2019)

[Tyk24] Tyk Technologies: Tyk Website (2024), URL `https://tyk.io`, accessed: 29-07-2024

[Wan23] Wang, Xinchen; Wang, Peng; Hu, Jian; Wang, Xiang; Zhao, Yuxiao; Wang, Shaolei; Diao, Liujian and Zhou, Shijie: Design and Implementation of a Low-Code Platform in the Power Sector, in: *Proceedings of the 2023 4th International Conference on Big Data Economy and Information Management*, BDEIM 2023, ACM

[Was19] Waszkowski, Robert: Low-code platform for automating business processes in manufacturing. *IFAC-PapersOnLine* (2019), vol. 52(10):pp. 376–381

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

[Wun24] WunderGraph Inc.: WunderGraph Website (2024), URL `https://wundergraph.com`, accessed: 29-07-2024

[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