Analyzing Environmental Data with the Titan Platform

Master's Thesis

Cedric Tsatia Tsida

October 22, 2020

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Sören Henning, M.Sc.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 22. Oktober 2020

Abstract

The natural environment is one of our biggest treasures — if not our biggest treasure — and it must be preserved. Therefore, understanding our natural environment is becoming increasingly important as we struggle to respond to the implications of climate change and new diseases. Climate change is the greatest challenge faced by almost every species. With the help of environmental data, we can understand most of the complex interrelationships present in the climate and consequently take preventive measures. However, with the rise of new technologies, the collection of environmental data has become very rapid and complex, and therefore, it is difficult or even impossible to process them using traditional methods. To overcome this problem, it is important to extract pieces of information from the data as soon as they are produced. We used Titan, a software that applies stream processing, to analyze the environmental data collected across Germany.

Titan comes with a set of solutions that facilitate the processing of huge amounts of data and allow software components to consume and supply data. In this work, three data sources have been connected directly to the Titan platform to collect diverse types of data. Once these connections were established, we analyzed the incoming data on the Titan platform using a set of data flows that consist of modular components connected to each other. Each modular component is responsible for a single task, such as data collection, data filtering, or data transformation. Moreover, we connect an external database to the platform to save the results of our analysis. Finally, we used a visualization tool to, create a dashboard for each flow implemented on the platform.

We evaluate the scalability of the Titan platform based on the number of incoming records. To do so, we continuously increase the amount of workload into the Titan platform and determine the minimum number of component instances that are necessary to process data without reaching high traffic of data across the platform.

Contents

Chapter 1

Introduction

1.1 Motivation

The natural environment is one of our biggest treasures — if not our biggest treasure — and must be taken care of. It is becoming increasingly important to understand our natural environment, as we struggle to respond to the implications of climate change and new diseases in its wake. Recent technological advancements have led to a deluge of environmental data that provides a huge amount of information, but it is complex to process such data due to its sheer volume, velocity, variety, and veracity. These characteristics are associated with the concept of Big Data [Chen et al. 2014], where the term volume refers to the size of the data, velocity indicates the speed of data in and out, variety describes the range of data types and sources, and veracity tells us the accuracy or truthfulness of data. To quickly extract pieces of information from data, it is important to compute them directly as soon as they are produced. This introduces to us the concept of stream processing [Stephens, Robert 1997], which allows us to analyze data in real time. However, to extract useful information from a big amount of data, new system architectures have been developed for data acquisition, transmission, storage, as well as the processing mechanism. Since most of these systems are independent of one another, their integration would be interesting.

For example, to allow software components to consume and supply data to each other, the Titan [Hasselbring et al. 2019] platform can be used to facilitate this process. The Titan platform uses the concept of flow-based programming [Morrison 2010] to provide services for running data flows. It enables developers to define their data flows by implementing new modular software components called *bricks* that can be easily added to the system. It also provides a user interface that, enables the user to graphically model a data flow. Therefore, to create a data flow by connecting bricks to each other, the user does not need any programming skills.

With its scalable architecture, the Titan platform comes with a set of solutions that facilitate the processing of Big Data. Therefore, using the Titan platform, we analyze a large amount of structured and unstructured environmental data collected across Germany. However, having access to a large amount of data, such as environmental data (particulate matter, nitrogen dioxide, weather, wind direction), and also having the possibility to process them, is a good starting point to extract useful information from the collected data. To

analyze the data efficiently, it must be visualized [Meyer et al. 2019]. The main objective of data visualization is to represent knowledge more intuitively and effectively by using different graphs. To convey information easily by providing knowledge hidden in the complex and large-scale data-sets, both the aesthetic form and functionality are necessary [Pai et al. 2017]. To achieve this, visualization tools must also be integrated into data analysis systems [Chen and Zhang 2014].

To analyze data, we must first have access to it. Hence, we use the OK Lab [Luftdaten Info 2015], Umweltbundesamt [Umweltbundesamt 2020b], and the Deutscher Wetterdienst [Deutscher Wetterdienst 2015] as main data sources to collect diverse environmental data from Germany. We essentially collect measurements from sensors, which provide particulate matter, nitrogen dioxide, weather, and the wind direction.

1.2 Goals

This thesis aims to develop a system that automates the analysis of particulate matter, nitrogen dioxide, weather, and wind direction data in Germany. The system helps us and domain experts to understand and analyze environmental data. To achieve this, we are going to develop a system architecture, implement the system with respect to the architecture, and use the implementation for evaluating the scalability of the Titan platform, as described in the following.

1.2.1 Identifying Data Sources

Before we start analyzing data, many pre-processing steps are needed. The first step of the entire process is to have access to data, for this, we need to have access to various data sources. There are many data sources available for Germany's environmental data. But since we are only interested in data sources that provide free access to their data, we, use the Umweltbundesamt (UBA), the Deutscher Wetterdienst (DWD), and the OK Lab as our main data sources. For each data source, we extract specific types of information. The first data source we consider is the DWD data source. This data source provides diverse environmental data such as air temperature, precipitation, wind, and solar data. However, considering our use case, we are interested only in air temperature and wind data; we, therefore, extract only those two types of data from this data source. The next data source is OK Lab a.k.a. Luftdaten.info. This data source also provides various types of environmental data, including air temperature and wind data. But since such data is already available within the first data source, we use the OK Lab data source to essentially extract particulate matter and humidity data. The UBA is the third data source. Among all the data sources listed above, this is the only source that provides nitrogen dioxide data. Therefore, the UBA will be used to collect nitrogen dioxide data. The data sources mentioned above provide different ways to collect data (see Table 1.1). How this is done has been explained in the following sections.

Table 1.1. Overview of the data sources

Data source	Protocol used for data collection	Collected data
Deutscher Wetterdienst (DWD)	FTP	- Air temperature - Wind direction
Luftdaten.info (OK Lab)	HTTP	- Particulate matter - Humidity
Umweltbundesamt (UBA)	HTTP	Nitrogen dioxide

1.2.2 Designing a Data Processing Flow

We design a data flow that receives metrics from the data sources mentioned in the previous section and analyze them. First of all, we implement for each data source one or more independent data flow(s). However, the data flow(s) implemented for each data source can be joined at a certain level to form a unique flow comprising the data flows of the three data sources. Each data flow must identify incomplete, incorrect, inaccurate, or irrelevant parts of the data. For that, we are going to define a workflow that, exchanges data across predefined connections by using message passing [Honghui Lu et al. 1995]. A suitable way of achieving this is by implementing the data flows in the Titan platform. The Titan platform enables us to implement bricks that can be easily connected, each brick would be responsible for one task. Each data flow consists of four steps:

- **Data Collection:** First, the data needs to be retrieved from the data sources. This is done at the beginning of the flow by the first brick.
- **Data Cleaning:** After the data has been retrieved, it is forwarded to the next brick for cleaning. This step consists of fixing or removing the discovered anomalies.
- **Data Transformation:** It consists of transforming the data into a format that can be accepted by the chosen storage system.
- **Data Storage:** At this stage, the transformed data are save into a time series database. This marks the end of each flow.

1.2.3 Data Visualization

Big Data visualization is among the most important components of working with various Big Data analytics solutions. Once raw data received from the data sources is cleaned, transformed, and saved to a database, it is represented with images to facilitate decision-making. Furthermore, data visualization is needed because a visual summary makes it easier to identify data patterns and trends. However, a good visualization tool comes with a minimum of features such as processing multiple types of incoming data, applying various filters to adjust the results, interacting with data-sets during the analysis, connecting

to other software to receive incoming data provided as input for them, and providing collaboration options for the user. Grafana [GrafanaLabs 2020] is an open-source analytics and interactive visualization web application that provides all the features listed above and even more. Consequently, it is used for data visualization in this thesis.

1.2.4 Evaluation of Scalability of the Titan Platform

The large amount of environmental data collected across Germany is mostly analyzed on the Titan platform. Therefore, we evaluate the scalability of the Titan platform by observing its behaviour on increasing workloads. Scalable systems are often based on design patterns, such as the microservice architecture pattern [Taibi et al. 2018]. However, microservices communicate with each other through stream processing engines. Therefore, comparing different stream processing engines can help us choose the more appropriate one for a particular use case. However, in many use cases (including ours), the stream processing engine has already been chosen. Therefore, it can be important to find an appropriate configuration or execution environment for the employed stream processing engine. In this sense, we adopt the Theodolite method [Henning and Hasselbring 2020b] to evaluate the scalability of the Titan platform.

1.3 Document Structure

The rest of this thesis is structured as follows: Chapter 2 presents the specific knowledge and technologies necessary to collect data from diverse data sources. Moreover, we present a description language used to facilitate the communication between the components within the Titan platform. Chapter 3 presents the architecture of the system used to analyze Germany's environmental data. Chapter 4 presents the implementation of diverse flow use to analyze Germany's environmental data. Furthermore, this chapter also presents the implementations of the flows used to evaluated the Titan platform. Consequently, Chapter 5 presents the evaluation of the Titan platform. In Chapter 6, we compare our approach with some other works. Chapter 7 summarizes the thesis and points out the aspects to be addressed by future research.

Chapter 2

Foundations and Technologies

2.1 Big Data Value Chain

A Big Data system is complex and can be considered as the association of many subsystems, where each subsystem describes a distinct phase with an associated entry point [Cheng et al. 2014]. At the same time, the system provides functions to deal with different phases in the digital data life-cycle, ranging from its creation to its destruction. A typical Big Data system can be decomposed into four consecutive phases, including data generation, data acquisition, data storage, and data analytics as illustrated in Figure 2.1 [Chen et al. 2014]. If we consider data as raw material, data generation and data acquisition are the exploitation process, data storage is the storage process, and data analysis is the production process that utilizes the raw material to create new value. Here we start with the exploitation process.

Figure 2.1. Big Data Value Chain

2.1.1 Data Generation

Data generation is concerned with how data is generated, and many Big Data systems begin at this stage. It includes the extraction of raw data from anywhere where information is generated and stored in a structured or unstructured format. Raw data is designated to mean large, diverse, and complex datasets generated from various sources. For example, it can be generated by humans, organizations, and machines [Ghotkar and Rokde 2016].

Human-generated data is consciously or unconsciously generated and refers, for example, to the vast amount of social media data (e.g., tweets, status updates, photos, etc.), emails, and many other documents we daily produce. The data generated in this category is normally much unstructured. Furthermore, the problem in the processing of this data is its rapid growth, multiple formats, and volume.

In contrast to human-generated data, data generated by organizations is structured and often siloed. This type of data is close to what businesses have; it is usually stored using traditional methods such as relational databases. Organization-generated data is stored for

current and future use, as well as for analysis of the past.

Machine-generated data refers to the information automatically generated by a computer process, application, or another mechanism without the active intervention of a human. It mostly unstructured; it is often created on a defined schedule or in response to a state change, action, transaction, or other events. This data is vast, complex, and contains a wealth of useful information such as information about the environment. In this thesis, we analyze machine-generated data produced from diverse environmental sensors. Sensor data is an integral component of the Internet of Things (IoT). To meet the demands of analysis and processing, the currently acquired data and the historical data within a certain time frame should be stored. Therefore, the data generated by IoT is characterized by large scales. Moreover, such data is also heterogeneous due to the diversity of the sensors.

2.1.2 Data Acquisition

As the second stage of the Big Data value chain, Big Data acquisition is commonly abbreviated as either DAQ or DAS, which includes data collection, data transmission, and data pre-processing [Chen et al. 2014]. Since data may come from different sources, data collection refers to the application of data collection technology that acquires raw data from a specific data production environment. Once raw data is collected, it is forwarded to a proper storage system. Finally, the collected data (raw data) might contain redundant or useless data that, unnecessarily increases the storage space and affects the subsequent data analysis. Thus, it is necessary to perform some data pre-processing operations to remove meaningless data and achieve efficient storage.

As mentioned above, data collection utilizes data collection techniques to acquire raw data from different data generation environments. Depending on how data is being generated, there exist diverse methods to collect them. For instance, the methods to collect data from log files or networks are different from those that are used to collect data coming from sensors. But, since we are mainly interested in sensor data, we would take a closer look at the methods responsible for collecting sensor data. According to the processes of data acquisition and transmission in the IoT, its network architecture may be classified into the sensing layer, the network layer, and the application layer. The sensing layer is responsible for data collection and essentially consists of sensor networks. The network layer is responsible for information transmission and processing, where close transmission may rely on sensor networks, and remote transmission depends on the Internet. Finally, the application layer supports specific IoT applications. However, at the sensing layer, signals that measure real-world physical conditions are converted into digital values which can be manipulated by a computer. Modern digital data collection systems include analogue-to-digital converters, sensors, signal conditioning, and computers with the DAQ software [DEWESoft 2020]. First, the main purpose of the analogue-to-digital converters within a data collection system is to convert analogue signals from the environment into discrete levels that can be interpreted by a processor. These discrete levels correspond to the smallest detectable change in the signal being measured. Second, sensors, often called transducers,

convert a physical phenomenon like temperature into voltage or current signals that can be used as inputs to the analogue-to-digital converters. Third, to make quality measurements on sensors, an additional component is often needed between the sensor and analogue-to-digital converters. This component is generally referred to as signal conditioning and includes amplification/attenuation, filtering, etc. Finally, computers with the DAQ software are essential for signal logging and analysis.

Upon the completion of raw data collection, the data is transferred to a data center for processing and analysis. Data transmission is referred to as the process of transferring data between two or more devices. It consists of inter-data center networks and intra-data center networks transmission. Inter-data center network transmission is used to transfer data from the source of generation to the data center, while intra-data center network transmission describes the data communication flows within data centers.

The data collection process tends to be sub-optimal for analytics as the collected datasets vary with respect to noise, redundancy, and consistency, among others; it is undoubtedly a waste to store meaningless data. Therefore, data needs to be pre-processed before being stored. Data pre-processing methods and techniques are used to discover knowledge from the data before mining. Data pre-processing includes many steps, such as data integration, data cleaning, and data normalization.

Data integration is defined as the combination of data from multiple sources in order to provide the user with a uniform overview [Lenzerini 2002]. Different methods can be used for data integration. One such method is the data warehouse, which is based on the ETL (Extract, Transform, Load) process. At the extraction step, all source systems are connected and data is collected and analyzed. Through the transformation stage, a set of rules is performed to transform the data into standard formats. Loading means importing the extracted and transformed data into the target storage. Generally, data integration methods are accompanied by a processing engine [Cafarella et al. 2009].

Data cleaning is the process of detecting and correcting or removing incomplete, redundant, corrupt, or inaccurate records to improve data quality. However, data cleaning includes five elementary stages [Maletic and Marcus 2000], which are defining and determining error types, searching and identifying errors, correcting errors, documenting error examples and error types, and modifying data entry procedures to reduce future errors. Furthermore, it is also essential to perform data normalization, where any skewed data is removed by transforming all variables in the data into a specific range for better analytics processing.

2.1.3 Data Storage

Data storage concerns persistently storing and managing large-scale datasets while achieving the reliability and availability of data access. With the rise of Big Data, new technologies have been developed to capture and store data [Oliveira et al. 2012]. This includes new data storage devices, new data architectures, and a new data access mechanism. However, a data storage system can be divided into hardware infrastructure and data management.

The hardware infrastructure consists of a pool of shared information and communication technology resources organized in an elastic way for various tasks, in response to their instantaneous demand. The hardware infrastructure should be able to scale up and out; it should be able to be dynamically reconfigured in order to address different types of application environments. Data management software is deployed on top of the hardware infrastructure to maintain large-scale datasets. Since Big Data is mostly unstructured, it breaks and transcends the rigidity of normalized RDBMS schemas and adopts NoSQL databases [Han et al. 2011]. NoSQL databases are becoming the core technology for Big Data, and there exist diverse types of NoSQL databases such as key-value databases and time series databases.

A key-value database is a database storage paradigm designed for storing associative arrays and data structures such as dictionaries or hash tables. In contrast to RDMS, key-value databases do not establish a specific schema and treat data as a single collection, with the key representing an arbitrary string. Every key is unique, and the users may input queried values according to the keys. Such databases feature a simple structure and are characterized by high expandability and a shorter query response time than those of relational databases. Over the past few years, many key-value databases, such as Apache Cassandra or Redis, have appeared.

A time series database is a database that is optimized for storing and serving time series through associated pairs of times and values. As mentioned in Section 2.3, InfluxDB is count among the most efficient time series databases, that is why we choose to use it in this thesis.

2.1.4 Data Analysis

Data analysis is described as the use of proper statistical methods to analyze massive data in order, to concentrate, extract, and refine useful data hidden in a batch of chaotic datasets; it, helps to identify the inherent law of the subject matter; it also helps, to maximize the value of data [Chen et al. 2014]. Data analysis plays a huge role in many fields, such as environmental science. With the high interest in Big Data, people are concerned about how to rapidly extract information from massive amounts of data to bring values for enterprises and individuals. However, the first impression of Big Data is its volume, with the biggest and most important challenge being scalability. Hence, in the last few decades, researchers have paid more attention to accelerate analysis algorithms in order to cope with the increasing volumes of data. Nevertheless, according to timeliness requirements, Big Data can be classified into real-time analysis and offline analysis.

Real-time analysis refers to the analysis of data as soon as it becomes available. In other words, the user get insights or can draw conclusions immediately after the data enters their systems. To realize real-time, analytics components such as aggregator, broker (e.g., Kafka broker), analytics engine (e.g., Titan platform), and stream processor, are needed. The aggregator compiles real-time streaming data from many different data sources. The broker makes the data available in real-time for use. Analytics engines correlate values and

join data streams while analyzing the data. Finally, the stream processor executes real-time application analytics and logic by receiving and sending data streams.

In contrast to real-time analysis, offline analysis is used for applications without high requirements on response time.

Since images speak louder than words, data visualization is especially important when it comes to data analysis. Therefore, to facilitate the work of domain experts in this thesis, data would be visualized using the Grafana tool (see Section 2.4).

2.2 The Industrial DevOps Platform Titan

With the continuously increasing amount of data, new technologies and frameworks have emerged for computing data continuously and in real-time. This called for agile software development due to the increasing software velocity. For example, it is now crucial to break the development-operations barrier to automate and integrate the processes between software development and IT teams have. This new practice is known as DevOps. DevOps combines software development (Dev) and information-technology operations (Ops). It aims to shorten the systems development life cycle and provide continuous delivery with high software quality. This solution can be applied to small and medium-sized enterprises to facilitate and improve their production chains. Applied to the industrial context, DevOps becomes Industrial DevOps [Hasselbring et al. 2019]. With the application of Industrial DevOps, the development and operations of industrial systems can be considerably improved, and expanded, thereby breaking the boundary between software development and software operations. However, some major principles need to be fulfilled to efficiently apply Industrial DevOps. Most importantly, Industrial DevOps is a continuous and cyclic process composed of four stages. The first step consists of monitoring and analyzing the production and software system. Through the second step, the information gain from the analysis in Stage 1 is used to identify new requirements. Based on the new requirements, the system is, then, modified. Owing to the modification of the system, some tests need to be performed. Therefore, the third stage of the process consists of automatically testing the new system. After passing all the tests, the fourth stage consists of replacing the old system with the new one and the cycle continues. This aims to enable continuous adaptions and improvements in the software. Moreover, Industrial DevOps intends to keep the organizational structure of an enterprise. Therefore, instead of enterprises adapting to their production process when applying external software solutions, Industrial DevOps support the alignment of the software with the production process. An example of the Industrial DevOps application is the Titan platform, which is based on the microservice architecture pattern [Hasselbring and Steinacker 2017] (see Figure 2.2).

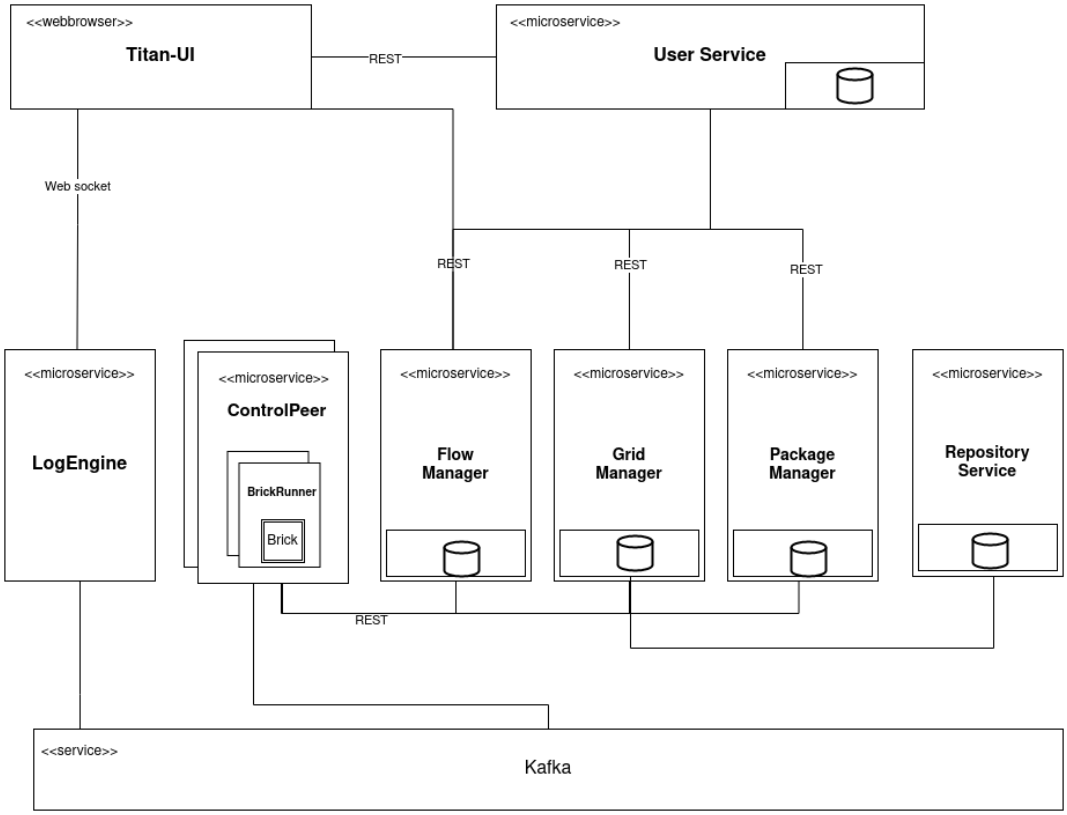


Figure 2.2. The Titan platform architecture (source: https://doc.industrial-devops.org/titanPlatform)

Titan is a software platform for integrating and monitoring industrial production environments using Industrial DevOps. It helps to collect the most diverse data formats, to convert them into comparable data materials, to correlate them, and to obtain information from them. Next, this information is converted into the data formats that are required at the output points. For the Titan platform to be able to perform such data modification, the platform, first of all, allows the user to collect data from various data sources such as sensors, machines, etc. Once the data is available on the platform, it is transformed by applying diverse operations. This is achieved by defining a set of data flows that are executed by the Titan platform. However, data flows are defined by connecting several modular software components that are known as bricks. These bricks are responsible for defining the tasks within the flows, and each bricks is reusable. At the end of the flow, the transformed data can, for example, be written to database or can be directly consumed by the software waiting for such input. The Titan platform also provides a user interface,

which enables users to graphically model flows by drag and drop once the bricks have been implemented. Each flow can be started or stopped by just one click (see Figure 2.3). Moreover, the user does not need any programming skills to model a flow.

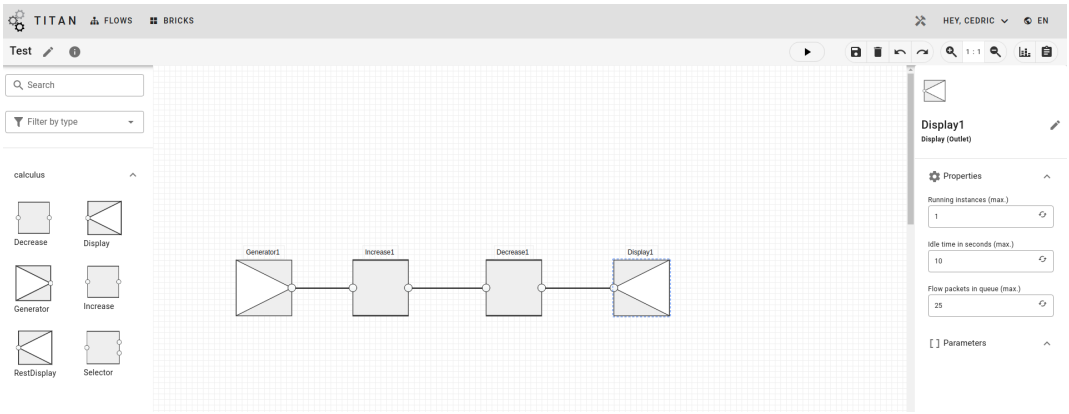


Figure 2.3. Screenshot of the Titan platform user interface

Independent of the goal to achieve, written bricks are the most likely implementation of a family of five bricks, compose of *inlet brick*, *outlet Brick*, *filter brick*, *selector brick*, and *general brick*. Depending on the chosen type of brick, a corresponding image (see Figure 2.4) will be shown in the graphical user interface after the brick has been successfully installed.

Inlet bricks are used to integrate data from various external data sources and are generally the origin of the flow. Since they are at the origin of the flow, they do not receive inputs from any other brick and are also responsible for generating one or more flow packet(s) from an external source and sending them into the flow.

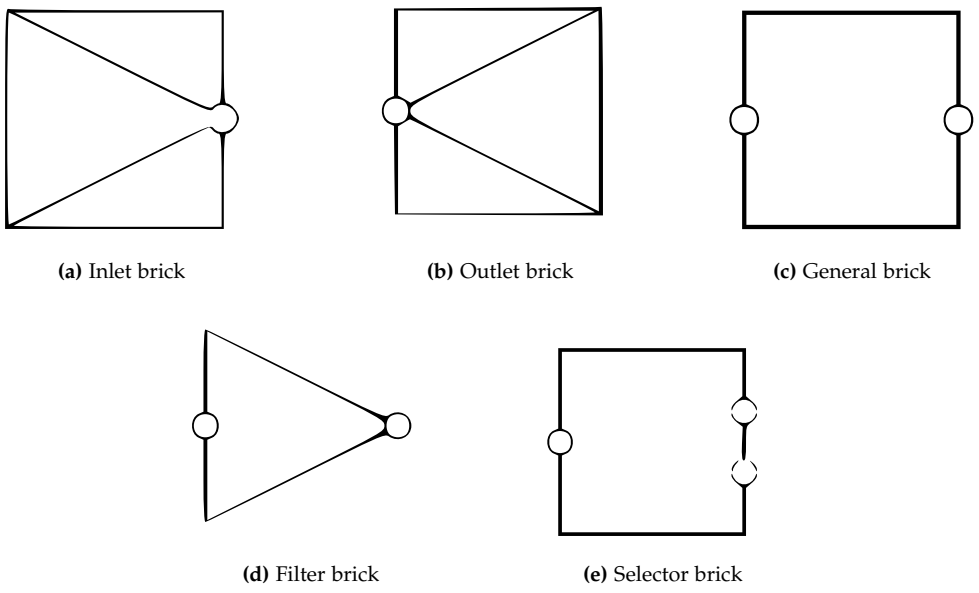
In contrast to inlet bricks, outlet bricks are generally at the end of the flow and do not have any output port. However, they can, for instance, be used to write data into a database.

As the name suggests, filter bricks are mostly used to apply filter operations on data. Once the input data matches the condition(s) implemented in the filter, it is forwarded to the output port.

Selector bricks have at least two output ports; they are capable of forwarding input data to one of the multiple ports, depending on the given condition.

Finally, the most generic brick is the general brick, it is mostly used for data transformation.

For each data source (DWD, UBA, Luftdaten.info), we create a flow composed of diverse types of bricks in order to analyze the collected data on the Titan platform.



(a) Inlet brick **(b)** Outlet brick **(c)** General brick

(d) Filter brick **(e)** Selector brick

Figure 2.4. Different type of bricks

2.3 The Time Series Database InfluxDB

Time series is an ordered time sequence of data points. It characterizes a quantitative event taken at equal time intervals [Fu 2011]. Thus, it is a sequence of discrete time data. For instance, timestamped data, such as air temperature and particulate matter, can be considered time series. Since there is a dependency between time and measurements, ordering data and changing the order could change the meaning of the data [Naqvi et al. 2017]. For example, time series could be used for hourly measurements of temperature at a specific weather station. Changing their order will lead to errors. Time series are used in various domains such as time series analysis, regression analysis, time series forecasting, and many more.

Time series analysis comprises methods for analyzing time series data to examine how a given variable changes over time. For instance, considering a weather station, we can consider the determination of the air temperature at each hour of a single day as a time series analysis.

On the other hand, regression analysis is often employed in such a way as to test theories that the current values of one or more independent timeseries affect the current value of another time series. In order words, it is used to examine how the changes associated with a specific variable can cause shifts in other variables over the same period. For example,

we can use regression analysis to examine how the concentration of particulate matter in the air of a specific region changes over time in regarding to the wind direction.

In the classical statistical handling of time series, making predictions is characterized as extrapolation. However, modern disciplines focus more and more on the topic and refer to it as time series forecasting. Such forecasting involves developing models to fit historical data and the associated patterns to predict future observations. As described above, these analyses are only possible if the data is in a time series format and, therefore the data should be saved in databases that can easily handle this type of data. These types of databases are called time series databases [Last et al. 2001].

2.3.1 Time Series Databases

As mentioned in the previous section, time series data is generated continuously and is usually just appended to the "old" ones. This leads to a large amount of data that has to be stored. One can be tempted to store such data in a relational database management system (RDBMS), where data with uniform characteristics is organized in tables and linked by relationships [Makowsky 1986]. However, time series data is not organized through many relationships and is generally written in a pre-defined order because it has a natural time order. Besides, data storage is less of a problem than efficiently retrieving data. A traditional relational database system is not sufficient when it comes to efficient time series data retrieval. The overhead generated by unused transaction management and query optimizers does not allow for efficient response times. Moreover, the scaling of an RDBMS is hardly possible. The solution can be presented in the form of specialized time series databases (TSDBs), based on open source NoSQL technologies. A TSDB is a software that is optimized for storing and serving time-stamped or time series data. It is built specifically for handling metrics, events, or measurements that are time-stamped. A TSDB enables its users to read, write, query, update, organize, and destroy time series data more efficiently. Therefore, for a database to be considered as a TSDB, it should at least fulfill some basic requirements.

Time series data is more frequently written than read, with 95% - 99% of operations being written [1]. Therefore, a TSDB should focus more on the ability to write. In most cases, a TSDB must be able to support highly concurrent and high throughput writes operations. However, this does not exclude the fact that a TSDB should also have a high-read performance.

Since time series data increases very quickly, it is, necessary for a TSDB to have the ability to handle changing demands by adding/removing resources. Therefore, time series databases must be designed to take care of scaling by including functionalities that are only possible when time is treated as the main focus.

A TSDB should also include functions and operations that are common to time series data analysis [Bader et al. 2017]. For instance, it should provide data retention policies,

[1] https://www.alibabacloud.com/blog/key-concepts-and-features-of-time-series-databases_594734

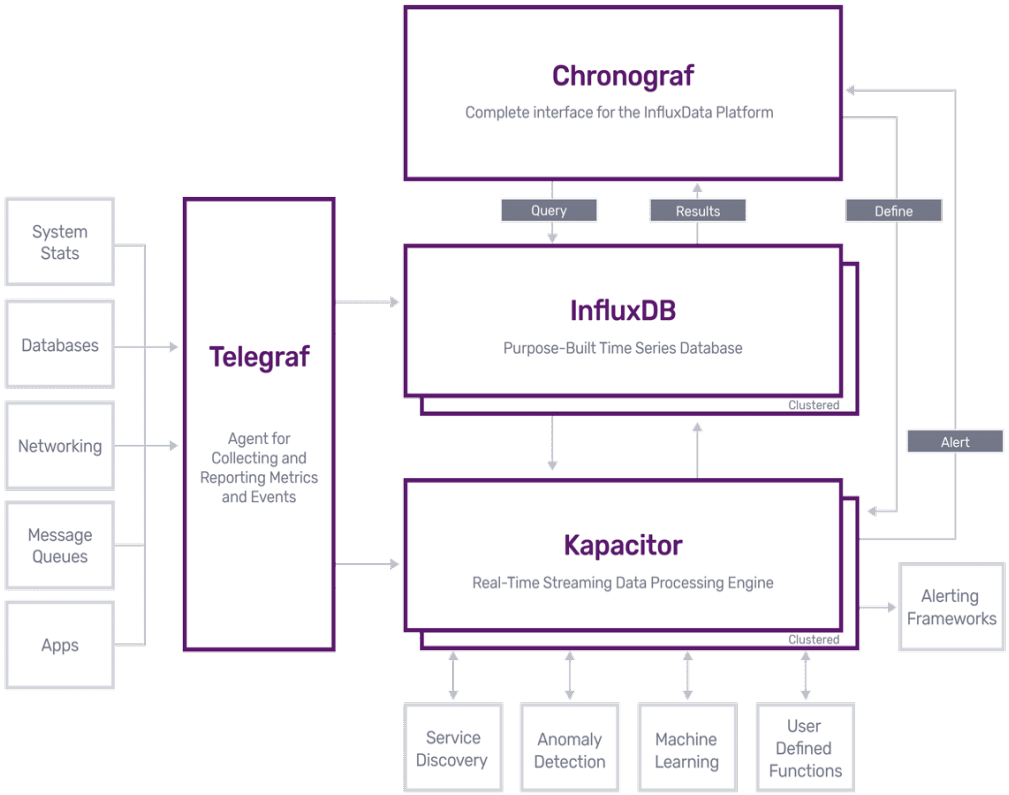


Figure 2.5. The TICK Stack (source: https://www.influxdata.com/time-series-platform/)

flexible time aggregations, range queries, continuous queries, etc. Additionally, it should also offer the mathematical querying functionality that enables the user to understand their data in ways familiar to the analysis of time series data. Consequently, this increases the usability by improving the user experience in case of dealing with time-related analysis.

2.3.2 InfluxDB

Lately, an open-source schemaless time series database, InfluxDB, has gained popularity. InfluxDB fulfills all the requirements described in Section 2.3.1 and even more. It is a relatively new database product as its development started in 2013. InfluxDB 1.8 is the latest version available. It is written in the Go programming language and optimized for storing time series data. InfluxDB has a free open-source and a paid enterprise version. The enterprise version brings clustering, manageability, and security features on top of the

open-source version. It has no external dependencies and provides an SQL-like language (InfluxQL), with built-in time-centric functions for querying a data structure composed of measurements, series, and points.

InfluxDB is part of the TICK Stack (see Figure 2.5). The TICK Stack is an acronym for a platform of open-source tools developed by InfluxData [2] in order to make the collection, storage, graphing, and alerting of time series data incredibly easy. The "I" in TICK stands for InfluxDB. The other components in the platform are: Telegraf (a metrics collection agent), Chronograf (a UI layer for the whole TICK Stack), and Kapacitor (a metrics processing and alerting engine). However, we use only the InfluxDB component and it is, therefore, important to understand some of its key concepts.

An InfluxDB database stores points, and each point has a measurement, a tagset, a fieldset, and a timestamp. Since InfluxDB is a time series database, it makes sense to start with what is at the center of everything; time. The time column is included in every InfluxDB database and stores timestamps that show the date and time in the default InfluxDB timestamp. However, the timestamp can be converted using the time precision RFC3339 UTC (see Figure 2.6). Each timestamp is associated with specific data. The next columns are mandatory and are fieldsets; they are made up of field keys and field values. A field key is of the type string, and a field value can be of the type string, integer, float, or boolean. A field represents data that we want to save and it is always associated with a timestamp. In Figure 2.6, we have just one field named *value*, which saves the air temperature of some weather stations in the state of Schleswig-Holstein in Germany. It is also important to note that fields are not indexed and therefore queries that use field values as filters must scan all the values that match the other conditions in the query. This leads to performance problems. To solve these problems, it is recommended that tagsets be added. Tags are optional and are also made up of tags keys and tags values. Both tag keys and tag values are stored as strings and record metadata. The tag key in Figure 2.6 are *geohash* and *station_name*. The measurement acts as a container for tags, fields, and the time column; its name is a string that describes the data stored in the associated columns. Measurement is generally referred to as tables in RDBMS. In the sample example (Figure 2.6), the measurement name is *air_temperature*.

We analyze the collected environmental data on the Titan platform and save the result to InfluxDB for further use.

2.4 The Visualization Software Grafana

The adage that "a picture is worth a thousand words" suggest that complex and sometimes multiple ideas can be conveyed more quickly and efficiently by a single image. Big Data visualization exploits this fact by turning data into pictures, thereby representing data more intuitively and effectively in pictorial or graphical format. In this sense, decision makers can

[2] https://www.influxdata.com/

air_temperature			
time	**value**	**geohash**	**station_name**
2020-06-10T00:00:00Z	11.8	u1ts5w6b0z	Schleswig-Holstein_2115
2020-06-10T00:10:00Z	10.2	u1ts5w6b0z	Schleswig-Holstein_2115
2020-06-10T00:20:00Z	9.7	u1ts5w6b0z	Schleswig-Holstein_2115
2020-06-10T00:30:00Z	9.1	u1ts5w6b0z	Schleswig-Holstein_2115
2020-06-10T00:00:00Z	8.4	u1wvp0ec7q	Schleswig-Holstein_6105
2020-06-10T00:10:00Z	8.2	u1wvp0ec7q	Schleswig-Holstein_6105
2020-06-10T00:20:00Z	7.9	u1wvp0ec7q	Schleswig-Holstein_6105
2020-06-10T00:30:00Z	7.7	u1wvp0ec7q	Schleswig-Holstein_6105

Figure 2.6. Sample time series data

quickly identify new patterns in a set of data [Chen and Zhang 2014]. A defining feature of Big Data visualization involves scalability. Therefore, Big Data visualization tools rely on a robust and scalable computer system capable of ingesting a large amount of data and processing it in order to generate a graphical representation that makes the understanding easy for humans. Moreover, to achieve efficient visualization, visualization tools are aimed to improve the clarity and aesthetics of data while considering the cognitive and perceptual properties of the human brain [Olshannikova et al. 2015]. There are dozens, if not hundreds, of visualization tools available for large sets of data. Many of them are basic and have a lot of overlapping features. However, there are standouts, such as Grafana[3], which are significantly easier to use than the others [Champman 2019].

Grafana is open-source visualization and analytics software [GrafanaLabs 2020]. It can be easily connected with various databases such as InfluxDB, Graphite, Prometheus, ElasticSearch, MySQL, PostgreSQL, etc. It enabled the user to query, visualize, put the alert on, and explore different types of metrics. It also allows users to create dashboards (see Figure 2.7) that can be reused for lots of different use cases such as time series analytics. The Grafana dashboard makes it easy to construct the right queries and customize the display properties in such a way that each panel can interact with data from any configured data source. The dashboard can contain a gamut of visualization options such as geographical maps, line charts, heat maps, histograms, and many others.

Since we save the results of our analysis in an InfluxDB database, we, connect Grafana to our database for visualization purposes.

[3] https://grafana.com

Figure 2.7. Example of Grafana dashboard (Screenshot)

2.5 RESTful Web Services

Since we want software components to interact with each other, the use of APIs is a good method to achieve this. An API or application programming interface is a computing interface that defines interactions between multiple software intermediaries [Emery 2011]. It is a set of rules that allow programs to address each other. The developer creates the API on the server and allows the client to interact with it. There are many types of APIs, but, we focus only on Web APIs in this thesis.

Before diving deeper into Web APIs, let us take a look at HTTP, which stands for Hypertext Transfer Protocol. HTTP is the protocol that enables the sending of documents back and forth on the web; it defines a set of rules that determines which messages can be exchanged and which messages are appropriate responses to others. The messaging is done between two computers called server and client. For instance, the client can initiate the communication and the server would reply with a message made up of the header and the body. The header contains metadata, while the body contains the requested data.

Web APIs are APIs that can be accessed using the HTTP protocol [Zhao 2002]. It is a concept and not a technology. The API defines endpoints, as well as valid request and response formats. Moreover, Web APIs include communication with the browser and may also provide services such as web notifications and web storage. In the category of Web APIs, there exist different types of API architecture. One of the most commonly used types of API architectureis the REST architecture [Rodriguez, Alex 2008].

REST determines how the API looks like. It stands for Representational State Transfer

and is a software architectural style that defines a set of constraints to be used for creating Web services [Fielding and Taylor 2000]. Web services that conform to the REST architectural style are called RESTful Web services and provide interoperability between the computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate the textual representations of Web resources by using a uniform and pre-defined set of stateless operations. It is a set of rules that developers follow while creating their API. A concrete implementation of the REST Web service follows four basic design principles: to be stateless; to expose directory structure-like URIs; to transfer XML, or Javascript Object Notation (JSON), or both; and the explicit use of HTTP methods. Each call to a URI is called a request, while the data sent back is called a response.

An endpoint is a location where a service can be accessed or it can also be defined as one end of a communication channel. When an API interacts with another system, the touchpoints of this communication are considered endpoints. For APIs, an endpoint can include the URI of a server or service (e.g., `UBA-Data`). Each endpoint is the location from which APIs can access the resources to carry out their functions.

The next component of a REST request is the type of method used to access data. REST guidelines ask developers to use HTTP methods on calls made to the server. There are five methods:

- The `GET` method is used to retrieve a resource from a server. When a `GET` request is performed, the server looks for the requested data and send them back to the client. In other words, a `GET` request performs a read operation.
- The `POST` method is used to create a new resource on a server. If a `POST` request is performed, the server creates a new entry in the database and tells the client whether the creation was successful. In other words, a `POST` request performs a create operation.
- `PUT` and `PATCH` methods are used to update a resource on a server. When a `POST` or `PATCH` request is performed, the server updates an entry in the database and tells the user whether the update was successful.
- The `DELETE` method is used to delete resources from a server.

It is also possible to include headers into a request. Headers are used to provide information to both the client and the server. They can be used for many purposes, such as authentication and providing information about body content. The data (sometimes called body or message) contains information that we want to send to the server. This option is only used with `Post`, `PUT`, `PATCH`, or `DELETE` requests. In the rest of the thesis, we use the REST API as a method to get access to data from our chosen data sources.

2.6 The File Transfer Protocol (FTP)

In addition to Web APIs, there are many other ways of retrieving or saving data. One of them is the use of an FTP-server. FTP stands for File Transfer Protocol and is a standard

network protocol used for the transfer of computer files between the client and the server on a computer network. An FTP-server is a computer that has an FTP address and is dedicated to receiving an FTP connection. The objectives of FTP are to promote the sharing of files (computer programs and/or data), to encourage indirect or implicit (via programs) use of remote computers, to shield a user from variations in file storage systems among hosts, and to transfer data reliably and efficiently [Postel and Reynolds 1985].

To be able to transfer data from an FTP-server to a client, a connection between them must be established. A data transfer is only possible if a connection to the appropriate ports is established and the parameters for the transfer are well chosen. The parameters are, for example, username and password. They are sent using the `USER` and `PASS` commands. If the information provided by the client is accepted by the server, the server will send a greeting to the client and the session will commence [Kozierok 2005]. If the server supports it, the user may also log in without providing login credentials. Since some of our data sources save data via an FTP-server, we implement this protocol to retrieve the needed data from those types of servers.

2.7 Big Data Analytics and Flow-Based Programming

Big Data is defined as data that contains a greater variety arriving in increasing volumes and with ever-higher velocity [Gartner Incs. 2001]. Furthermore, Big Data also refers to the combination of structured, semi-structured, and unstructured collected data. Owing to these facts, it is difficult to process Big Data by using traditional methods. Therefore, there is a need for different methods like flow-based programming.

Flow-based programming (FBP) is a programming paradigm, developed by J. Paul Rodker Morrison in the late ’60s, that uses a data processing factory metaphor for designing and building applications [Morrison 1994]. FBP defines applications as networks based on the concept of multiple asynchronous processes communicating through data streams. Nevertheless, processes do not communicate directly with their neighbours. Communications are done across pre-defined connections that are specified externally to the processes. This enables us to apply the concept of the "black box" on processes in such a way that, they can be reconnected to form different applications without having to be changed internally. A good application of this method is the Titan platform.

2.8 The UJO Data Object Notation

The Titan platform enables the communication of software components (bricks). One of the main goals of communication is the transmission of information. Fluid communication demands a unifying language. Therefore, the Titan platform uses a description language know as UJO Schema to facilitate the communication between components. The UJO Binary Data Object Notation is intended to be used as a data exchange format through network

connections and for storing data in files. The primary goal of UJO Schema is to provide many different data types to support efficient and reliable automatic conversion without the loss of information. UJO means a container, and a container in the context of UJO is a list, a map or dictionary, and a table [Wobe-system 2016]. It is also possible to have nested containers this means a UJO container can recursively contain a list, a map, or dictionary. The advantage of UJO is that it uses as few as possible bytes to encode data, depending on the type. UJO is much simpler and more efficient than any text format, such as JSON or XML, no matter how much effort is put into optimizing. Furthermore, no additional names are added to any data field unless an associative array is using the text keys. It is also easy to implement and allow flexible data modelling due to hierarchical containers. Consequently, to ensure reliable communication between components across the Titan platform, we would wrap each data package inside a UJO object before sending it to the next component.

Chapter 3

An Architecture for Environmental Data Analysis

Software architecture has become a central subject for software engineers. As software systems have become more complex, the algorithms and data structures relating to computation no longer constitute any major design problems. Therefore, software architecture serves as a blueprint; it provides an abstraction to handle the system complexity and describes its major components, their relationships, and how they interact with each other [Garlan and Shaw 1993]. To model our system, we use a unified modeling language (UML) component diagram, which is essential for visualizing, specifying, documenting, and modeling the physical aspects of the system. Our software comprises three principal components, as shown in Figure 3.1. The first two components (Titan platform and InfluxDB) are responsible for the backend and the last component for the frontend.

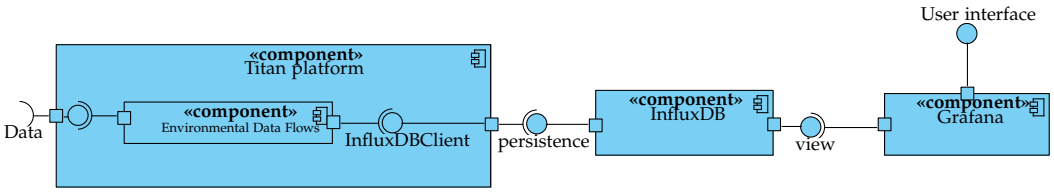


Figure 3.1. Approach of the thesis: Component diagram

3.1 Backend

The backend contains two principal components, as shown in Figure 3.1. The first component is the *Titan platform*, which is responsible for establishing connections with external data sources through the required interface *Data*. Once the connections to the data sources are established, data is forwarded to the inner component *Environmental Data Flows*. In this component, we analyze the data collected from the data sources by implementing various data flows. A data flow is responsible for analyzing the data collected from a single or more data sources. Moreover, the *Environmental Data Flows* component, through its diverse data flows, is also responsible for transforming the data to match the format required by the next component. At the end of each data flow, data is written to a database. Therefore, the

Titan platform also establishes a connection with the component *InfluxDB* that is responsible for data storage.

3.2 Frontend

The frontend is responsible for visualizing the data provided by the backend. Since the component *Grafana* handles the frontend, it, establishes a connection to the database. Once data is available in the database, it is automatically read and visualized by *Grafana*. This is achieved via the interface *view*. To get data for the visualization, *Grafana* uses query editors (see Figure 3.2), and thereby helps to write queries depending on the data source. In *Grafana*, we create and configure a dashboard, where each panel receives data from one or more measurements from the database.

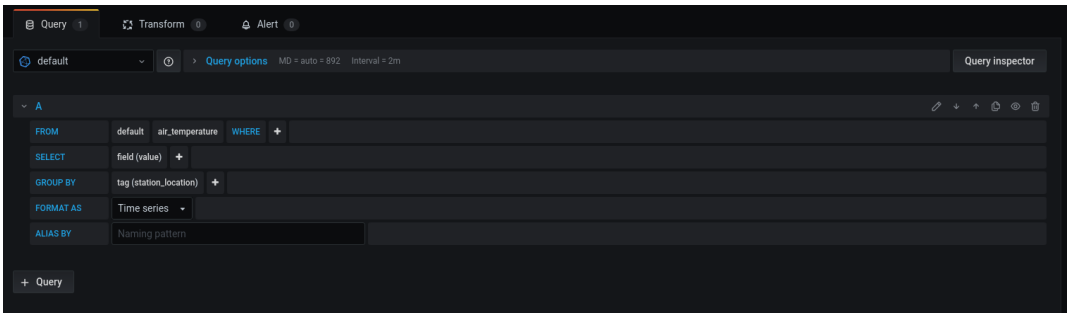


Figure 3.2. The Grafana query editor (Screenshot)

Chapter 4

Flows for Environmental Data Analysis

Based on the architecture described in Chapter 3, we implement and integrate all the components presented in it. This chapter describes how each component has been implemented, integrated, or configured to satisfy our requirements. First, we start with the implementation of bricks on the Titan platform. The Titan platform enables us to implement five types of bricks to build a flow. As described in Section 1.2.1, we extract data from three different data sources. Therefore, we implement three data flows that would handle each data source. The data flow would be joined at a certain level, for example, to write information into a common database. Each flow starts with an inlet brick that enables the Titan platform to communicate with the outside world (data sources). Second, we integrate InfluxDB to save the data at the output of the Titan platform and finally, it is visualized using Grafana.

4.1 The German Weather Service (DWD) Flow

4.1.1 Data Source and Acquisition

The first data source we consider is DWD, which stands for Deutscher Wetterdienst and provides weather and climate information free of charge to the Open Data Server[1]. The data is organized in a well-defined hierarchy and stored as zip files on an FTP Server. DWD also categorizes its data into three main categories. First, we have the historical data category that covers the period 01.01.1991 - 31.12.2017. Next, we have the recent data category that, covers a period from the last 500 days until yesterday. Finally, we have data categorized as "now", which covers a period from yesterday until now. Since we are more concerned about data that is close enough to real-time data, we would focus only on the category "now". In this category we collect air temperature and wind direction information.

4.1.2 The DWD Collector Brick

Each flow on the Titan platform starts with an inlet brick and gets connected to external data sources. Hence, we implement an inlet brick (*DWD Collector*) that establishes a connection between the Titan platform and the data source (DWD server). This can be done by using the Python library ftplib[2]. Data is saved in zip files and after successfully logging into the

[1] `https://opendata.dwd.de`

[2] `https://docs.python.org/3/library/ftplib.html`

server, we first need to extract the zip files to read their content. Each zip file is related to a station. After an extraction, a text file is generated containing the data collected in a station. Since the text file is organized in such a way that each line contains data for a timestamp, each line of the text file is read and sent to the next brick inside a UJO wrapper. Moreover, data gets generated approximatively every 10 minutes. We, therefore, establish a new connection to the DWD server every 10 minutes in order to have access to the newly generated data. The extracted files are saved in a temporary directory that is automatically deleted once their content has been read.

4.1.3 The DWD Filter Brick

After the data is extracted from the server and put into the Titan platform, the next step is to forward the raw data to a filter brick in such a way that incorrect data can be detected and removed from the flow. Each data value is marked with a quality level. The quality level (QN) describes the quality inspection process and refers to a complete set of parameters for a specific date. Different quality inspection procedures (on different levels) decide which values are wrong [Kaspar et al. 2013]. There are three quality levels. The first quality level $QN = 1$ uses formal verification methods on data when it is decrypted and loaded. The second quality level $QN = 2$ checks the data according to some individual criteria. Finally the third quality level $QN = 3$ verifies and corrects the data using the QUALIMET method [Spengler, Reinhard 2002]. We filter the data according to the quality level $QN = 2$ because striking errors can be included in the data of quality level $QN = 1$ and those of quality level $QN = 3$ are less available. Furthermore, the error values are marked with -999. Therefore, we drop all the data points having this value. The above-described data cleaning has been implemented in a filter brick (*DWD Filter*) that is directly connected to the inlet brick (*DWD Collector*).

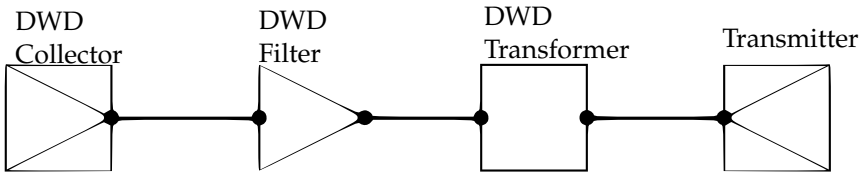


Figure 4.1. The DWD service data flow

4.1.4 The DWD Transformer Brick

After the data is filtered, we save it into a time series database (InfluxDB) before its visualization. Since our visualization software is Grafana, it is necessary to format the data in such a way that it can be accepted both as the InfluxDB input and the Grafana input. Furthermore, we use the Grafana world map panel plugin to display the data on

a map across Germany. The Worldmap Panel [3] is a tilemap that can be overlaid with circles representing data points from a query. It can be used with time series metrics, with Grafana data from Elasticsearch [Divya and Goyal 2013] or data in the table format. If the data is saved in a time series format, the metric name needs to match a key from a list of locations that are provided either as JSON files with locations and their coordinates or as JSON endpoints that return a list of locations and their coordinates. Considering the last use case, it is obvious that additional work, such as configuring an endpoint has to be done. Hence, we transform the data into a table format before forwarding it to InfluxDB. The table format is similar to the line protocol format, which is the standard format for InfluxDB. Table data is organized in columns and rows. While using this format we add a column which is a geohash or two columns that contain the latitude and longitude to specify the location. The knowledge of the geographic coordinate enables us to plot data on the map of Germany, where each station is situated. As a consequence, we implement an additional brick (*DWD Transformer*) to transform our data into the table format before persisting them. In keeping with the terminologies of the InfluxDB, the metrics are saved as fields and geographic information as tags.

4.1.5 The DWD Transmitter Brick

To write the results of our analysis to the InfluxDB database, we need to implement an outlet brick (*Transmitter*). Since writing data point by point to InfluxDB makes our entire system very slow, data was written in a batch of 10 000 data points to achieve high performance. To be able to connect to InfluxDB, we use the python library InfluxDBClient[4]. It provides two protocols for writing data into the InfluxDB database. The first protocol is the JSON protocol, where each point is sent to InfluxDB as a JSON object; the second one is the line protocol. A line protocol is nothing more than a text-based format for writing points into a database. Points must be in the line protocol format for InfluxDB to successfully parse and write them. However, if data is sent to InfluxDB with the JSON protocol format, InfluxDB internally transforms that data into the line protocol format before parsing and writing it. We use the line protocol to optimize the writing time.

Finally, we have a flow comprising four bricks (see Figure 4.1). The first brick (*DWD Collector*) collects raw data from the DWD server, wrap them point by point into UJO containers, and forwards them to the second brick. The second brick is a filter brick (*DWD Filter*). The filter brick controls the quality of each data point with regard to the above described criterion. Data points that pass the quality test are sent to the next brick, and the others are dropped. The third brick is a general brick and has the purpose to transform each received data point into a table format (*DWD Transformer*). This format enables us to send data to the InfluxDB efficiently . At the end of the flow, we have an outlet brick (*Transmitter*) that is responsible for writing data into the database.

[3] `https://grafana.com/grafana/plugins/grafana-worldmap-panel`

[4] `https://influxdb-python.readthedocs.io/en/latest/api-documentation.html`

Figure 4.2. The DWD service visualization (Screenshot)

4.1.6 Visualization

Once data is available in the InfluxDB, it is read and visualized by Grafana. For the DWD flow, we create a dashboard composed of five panels (see Figure 4.2). The first two panels (Geo Wind Direction, and Geo Air Temperature) uses the *Worlmap Panel Plugin* for Grafana to represent data points with their description on the map of Germany (see Figure 4.3). The next two panels are line graphs representing information of the first two panels in more detail (see Figure 4.4 and Figure 4.5). The last panel is also a line graph that shows the correlation between the air temperature and wind direction in each station (See Figure 4.6).

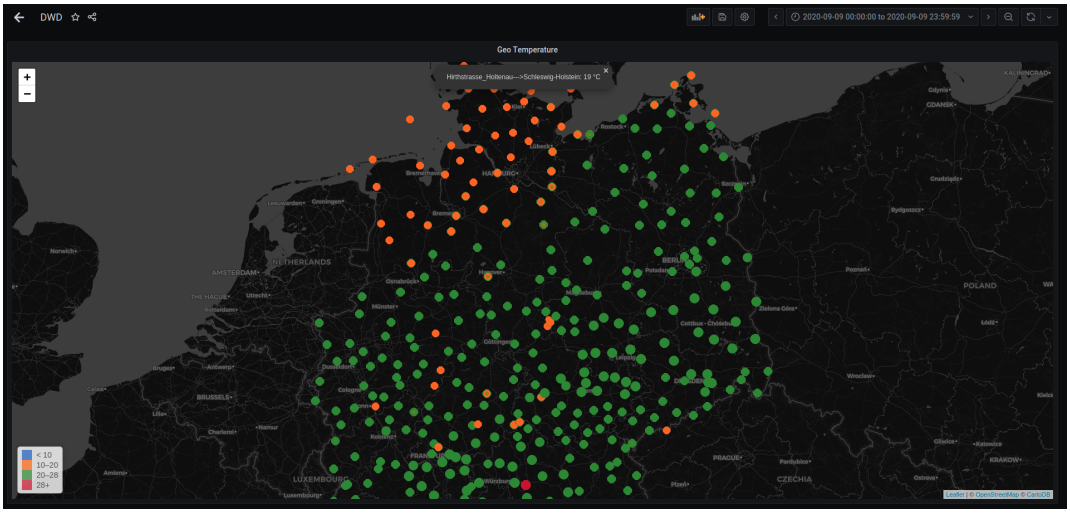


Figure 4.3. Air temperature over Germany (Screenshot)

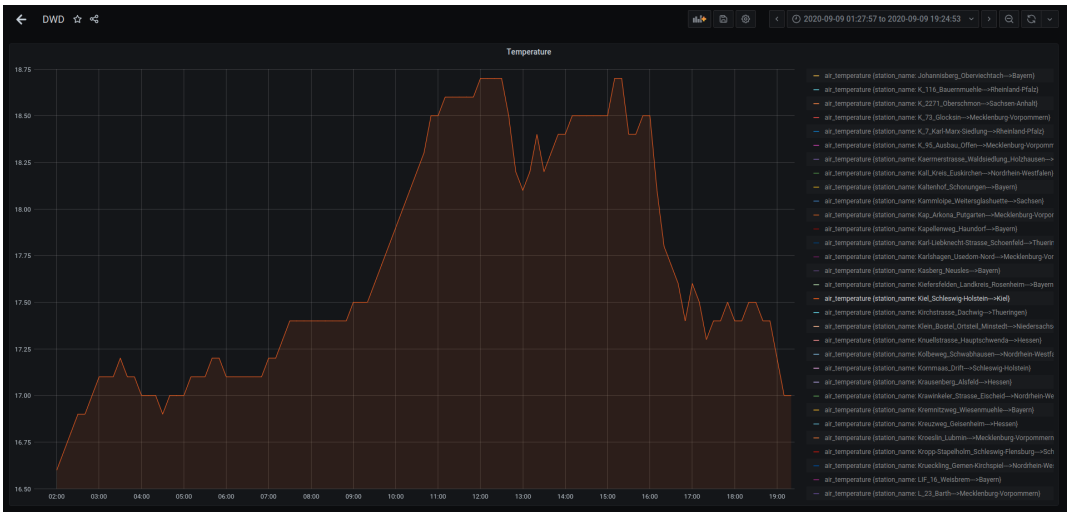


Figure 4.4. Air temperature in Kiel on 2020-09-09 (Screenshot)

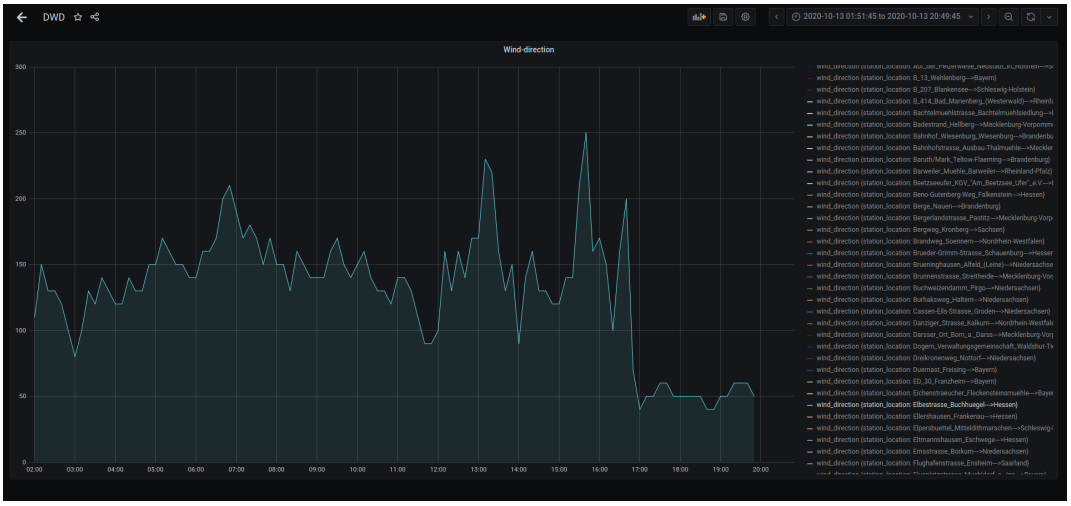


Figure 4.5. Wind direction in Hessen Elbestrasse on 2020-09-11(Screenshot)

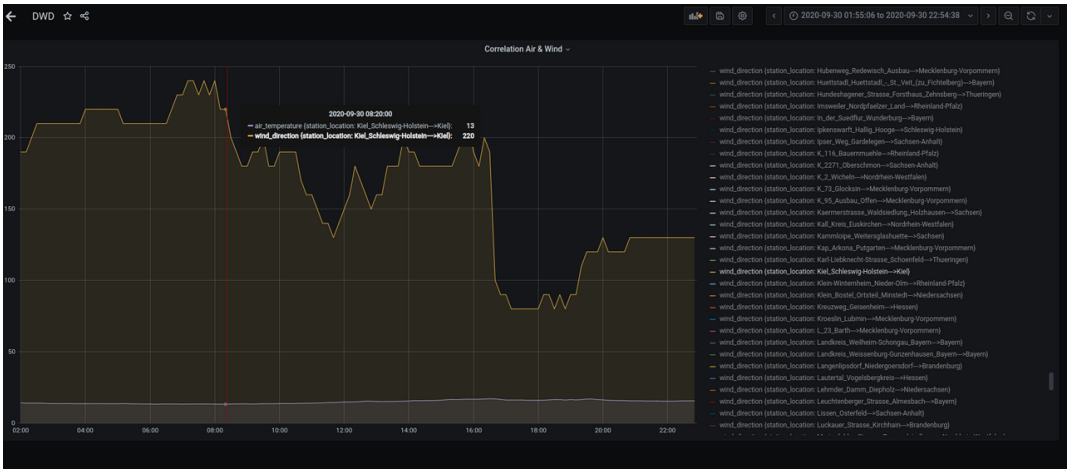


Figure 4.6. Correlation of air temperature and wind Direction in Kiel on 2020-09-30 (Screenshot)

4.2 The Luftdaten.info Flow

4.2.1 Data Source and Acquisition

The Luftdaten.info is a community-driven project for collaboratively monitoring air quality through a large number of low-cost DIY (Do It Yourself) stations deployed all around the

world [Bungartz et al. 2018]. For our use case, we restrict the area to Germany. Besides the measurements of air temperature and humidity, Luftdaten.info also includes the most important classes of particulate matter (PM10 and PM2.5). PM10 is a type of particulate matter that can have a size up to 10 micrometers, while those of size up to 2.5 micrometers are PM2.5. According to the World Health Organization (WHO) [World Health Organization and others 2006] particles of any size, less than 10 micrometers can get deep into the lungs and some may even get into the bloodstream. Of these, particles less than 2.5 micrometers in diameter or PM2.5 pose the greatest danger to health. Therefore, the smaller the particulate matter, the more dangerous they are for human health, that is why we would focus more on PM2.5. Luftdaten.info provides data in two different ways: current data, typically covering the last five minutes (provided via API) and historical data provided in the CSV (comma-separated values) files via HTTP with one file per station and day.

4.2.2 The Luftdaten.info Collector Brick

The Luftdaten.info uses different types of sensors such as SDS011 (for particulate matter data), BME280 (for humidity, temperature, and air pressure data), and DHT22 (for humidity and temperature data), among others to collect weather data. First, we extract historical data since we are interested only on particulate matter data, we address only the data coming from SDS011 sensors. However, the SDS011 sensor provides reliable data only if it is located in a region where the humidity is less than 70% [Meyer et al. 2019]. To be able to filter out wrong values (or less reliable values) after the data collection, it is essential to have information about the humidity in each region where we have the SDS011 sensor. Nevertheless, to construct a common base for the analysis, we need to integrate all SDS011 sensors and humidity sensors into a single data set. Therefore, we need to merge the humidity data with the particle concentration data in terms of the sensor location and time. Since data is saved on an HTTP Server, we execute an HTTP request to retrieve the needed data. For this, we implement an inlet brick called *Luftdaten Collector* that uses the Python library *requests*[5] to retrieve data. To have direct access to CSV files and optimized the processing time, we include some pre-processing that fetches the direct links of each needed CSV file and save them locally into a text file. Locations are represented as integers value, and particulate matter sensors and humidity sensors located at the same place differ of plus-minus one in their location. This makes it more comfortable to determine the particulate matter and humidity data of the same location. After the pre-processing phase, we use the HTTP `GET` method with each link saved in the text file as an argument to read the CSV files content. If the creation date of the current text file on the system is older than yesterday, the file gets deleted and a new one with the links of the newly generated data is generated. This is necessary because new data is generated approximately every 24 hours. Afterwards, each line of the CSV files (CSV file containing humidity data and CSV file containing particulate matte data) are read and join together, after which they are wrapped

[5] `https://requests.readthedocs.io/en/master/`

into a UJO container and sent to the next brick.

4.2.3 The Luftdaten.info Filter Brick

After joining the humidity and particle data and feeding it as input to the Titan platform, the next step is to clean the data. For data cleaning, we implement a brick called *Luftdaten Filter* that is directly connected to the *Luftdaten Collector*. In the *Luftdaten Filter* we filter the data in order to narrow the result to Germany. This allows us to filter out a large amount of data that is not needed for our study case. To do so, we check for each data point if its latitude and longitude happens to be situated in the geographic coordinates of Germany. Additionally, we identify and remove incorrect data by finding measurements taken under unstable conditions. We obtain variables to identify unstable conditions in the collected datasets by looking into the sensors' data sheet [6]. So removing the invalid measurement values is straightforward. For instance, the values generated by the particle sensors must belong to the range [0, 999]. Therefore, any values above or under this range are discarded. Furthermore, we remove all particle concentrations where humidity is greater than 70%, because, the particle sensors do not provide reliable readings above this value.

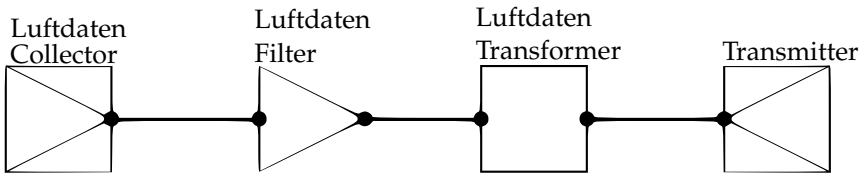


Figure 4.7. The Luftdaten service (with only historical data) data flow

4.2.4 The Luftdaten.info Transformer Brick

After filtering, data must be saved into the database. But before this, it has to be transformed in, such a way that it should respect the format described in Section 4.1.4. To keep the implementation of each brick as simple as possible, we implement a separate brick (*Luftdaten Transformer*) that is different from the *DWD Transformer*, which is directly connected to the filter brick. Since the *Transmitter*, is only used to write data into the database and bricks are reusable across the Titan platform, there is no need to implement it twice. Therefore, after transformation, data is written into the database by the *Transmitter Brick* as described in Section 4.1.5. The flow describing the entire process is similar to the DWD Flow, and it is also composed of four bricks (see Figure 4.7). To be able to run the entire flows (DWD and Luftdaten.info flow (historical data)) once, we join the two flows to form a unique flow. (see Figure 4.8).

[6] https://cdn-reichelt.de/documents/datenblatt/X200/SDS011-DATASHEET.pdf

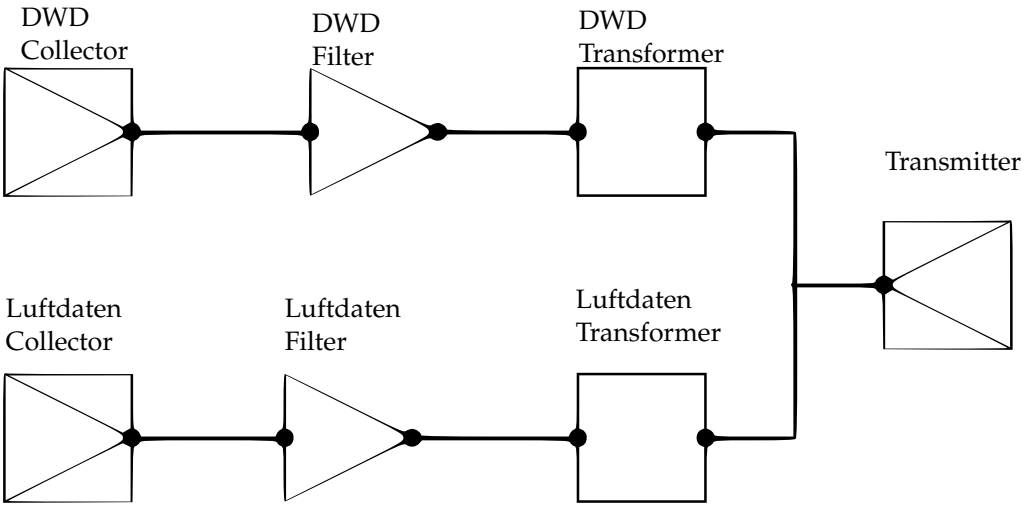


Figure 4.8. The DWD and Luftdaten service (with only historical data) data flow

4.2.5 The Luftdaten.info Real-Time Collector Brick

The Luftdaten.info service provides historical and current data in two different ways. Since we are done with historical data, we now incorporate current data to the current flow. Current data is provided via the REST API, and is generated every five minutes. Therefore, we implement an inlet brick called *Luftdaten Real Time Collector* which requests data every five minutes from the Luftdaten.info server. Owing to a huge amount of data received from each request, we first narrow the result down to Germany and then group all sensors according to their respective locations. Each group is then forwarded to the next brick inside a UJO container. One can now be tempted to use a unique brick to clean the data coming from both the *Luftdaten Collector* brick and the *Luftdaten Real Time Collector* brick. But owing to the enormous difference in the structure of the data sent by those bricks, cleaning such data with a unique brick could lead to inefficiency problems or make the brick very complex to understand. Hence, we implement a filter brick called *Luftdaten Real Time Filter*.

4.2.6 The Luftdaten.info Real-Time Filter Brick

The *Luftdaten Real Time Filter* cleans each packet coming from the *Luftdaten Real Time Collector* brick by retaining only the data about humidity and particle matter. This is important because sensors provide reliable particle matter data only up to a certain humidity in the air. After this, we filter the particles (PM10 and PM2.5) with regards to the humidity (particle situated in a region with the humidity above 70% are dropped) and send them

point by point to the next brick.

Now that the data produced by both Luftdaten filter bricks have more or less the same structure, they can be transformed by the same brick. To do so, we improve the *Luftdaten Transformer* brick in such a way that it can transform the data coming from the *Luftdaten Real Time Filter*. The *transmitter* brick is appended at the end of the flow in such a way that the data is written into the database (see Figure 4.9).

Finally, we joined the flow of real-time data with the flow of historical data. They are joined before the transformer brick in such a way that the data coming from both filters are transformed by a unique brick. At the end of the flow, we use the existing *Transmitter* brick to write data into the database. The entire flow for the Luftdaten.info service is shown in Figure 4.10 . Now that all flows (DWD flow and Luftdaten.info flow) are functioning, they can be joined to form a unique flow (see Figure 4.11) that can be run with just one click.

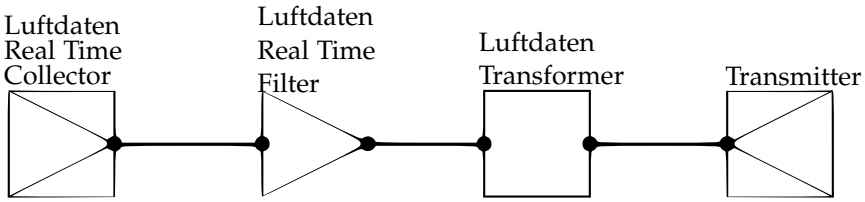


Figure 4.9. The Luftdaten service (with only real-time data) data flow

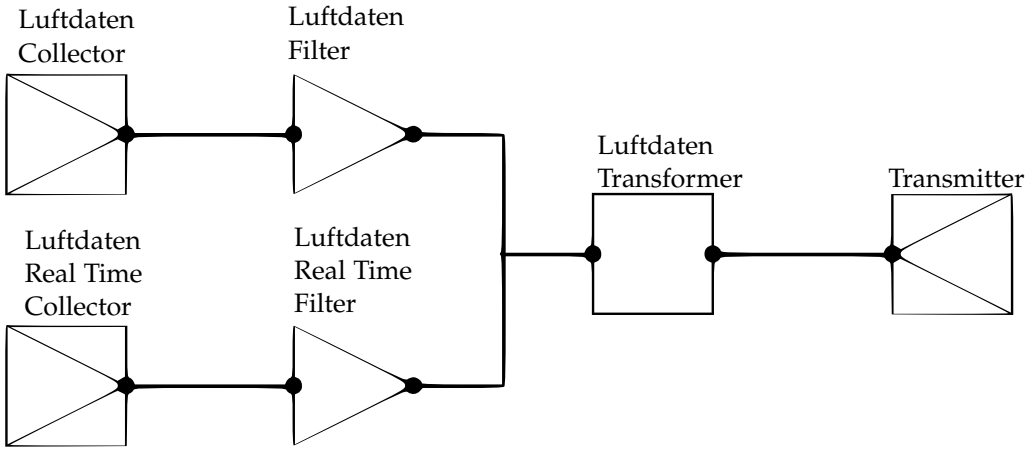


Figure 4.10. Luftdaten service data flow

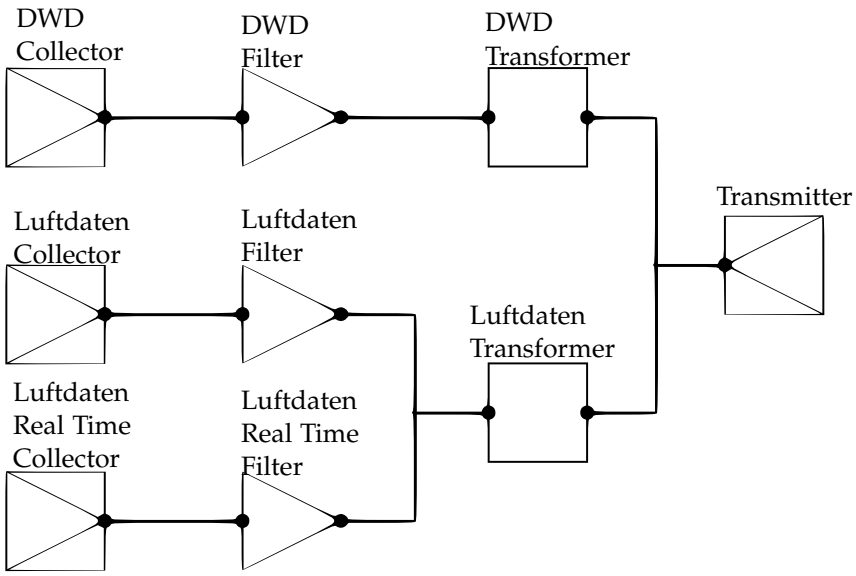


Figure 4.11. The DWD and Luftdaten service data flow

4.2.7 Visualization

Once the data is available in the InfluxDB, it is read and visualized by Grafana. Similar to the DWD flow, we also create a new dashboard for Lufdaten flows. (see Figure 4.12). The first two panels (Geo Particle, and Geo Humidity) uses the Worlmap Panel Plugin for Grafana to represent data points with their description on the map of Germany (see Figure 4.13). The next panels are line graphs representing the information of the first two panels (see Figure 4.14) in more details. The following panel is also a line graph that shows the correlation between particulate matter and humidity in each station. Moreover, the last two panels (not completely visible in this case) are the line graphs for real-time data.

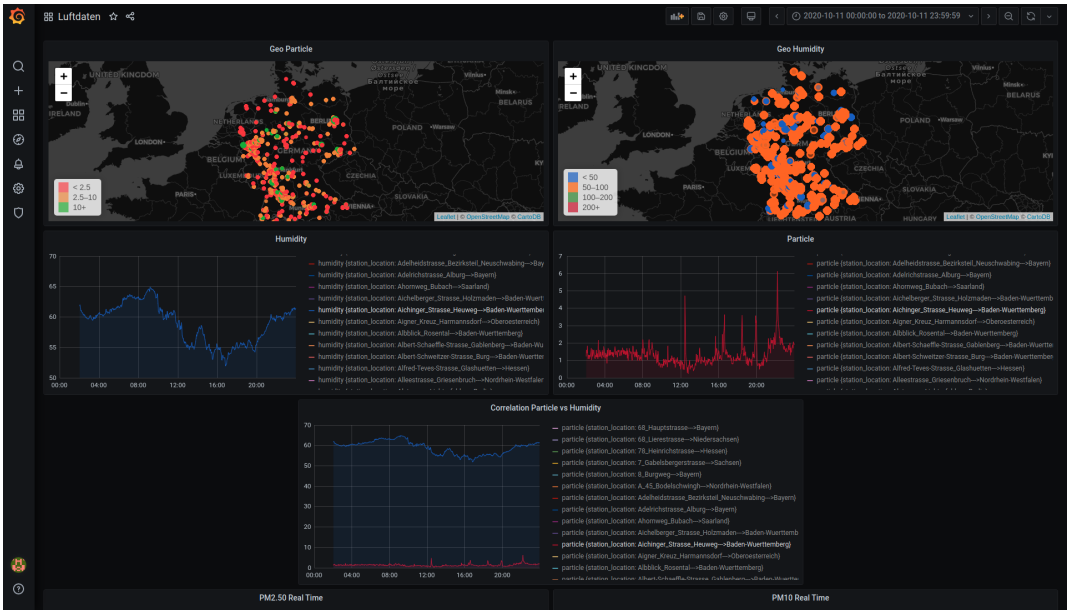


Figure 4.12. Luftdaten.info dashboard (screenshot)

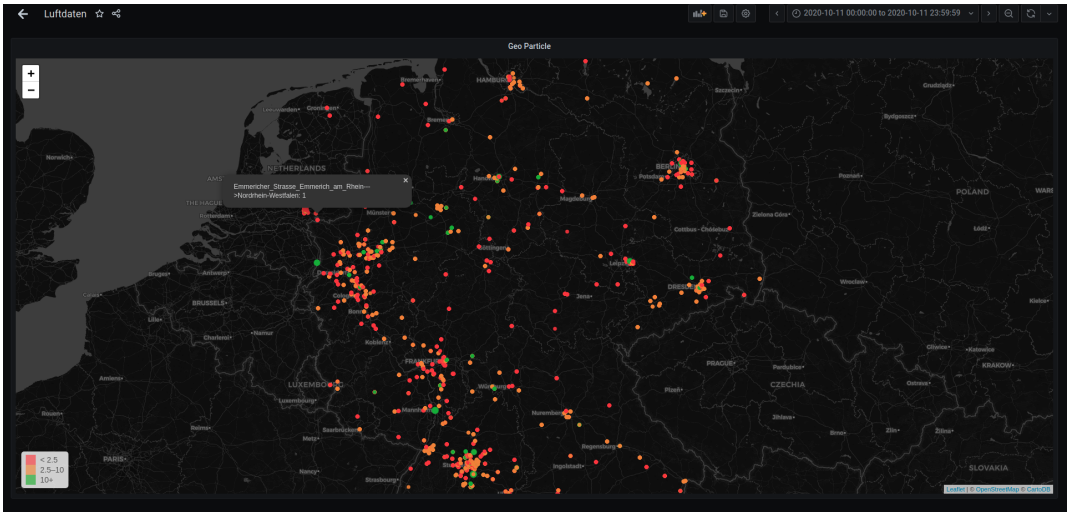


Figure 4.13. Particulate matter over Germany on 2020-09-11 (screenshot)

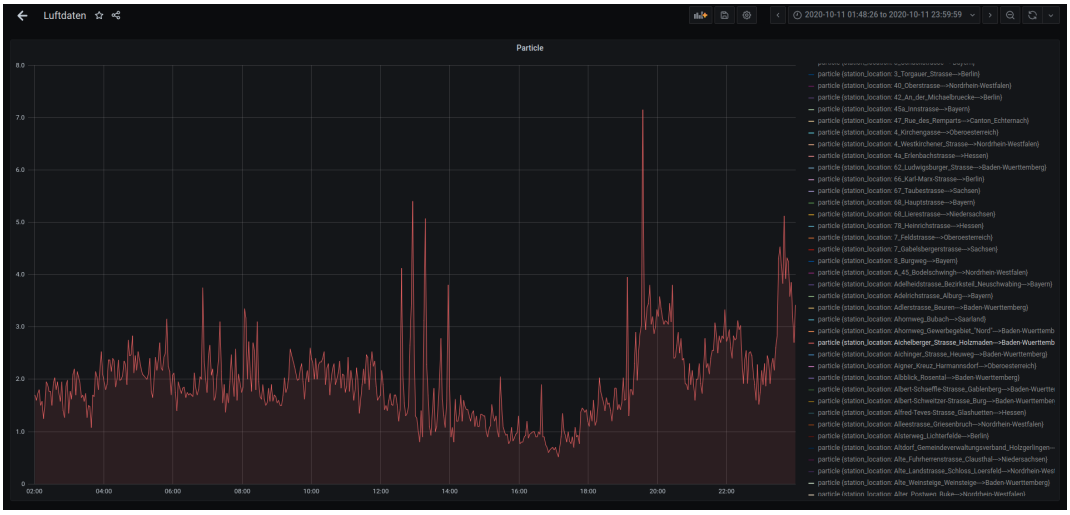


Figure 4.14. Particulate matter in Aichelberger Strasse Baden Wuerttemberg on 2020-09-11 (screenshot)

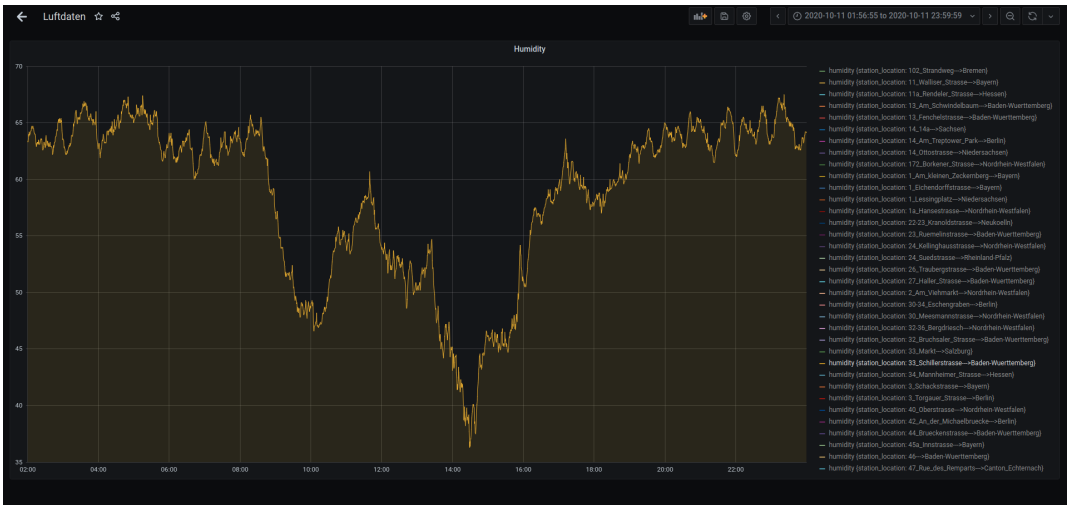


Figure 4.15. Humidity in Schillerstrasse Baden Wuerttemberg on 2020-09-11 (screenshot)

Figure 4.16. Correlation between Particulate matter and humidity in Aichelberger Strasse Baden Wuerttemberg on 2020-09-11 (screenshot)

4.3 The German Environment Federal Office (UBA) Flow

4.3.1 Data Source and Acquisition

The Umweltbundesamt (UBA) is the central environmental authority of Germany. The ultimate goal of the UBA is to ensure that there is a healthy environment in Germany in which people can live free from harmful environmental influences like pollutants in the air or water as much as possible [Umweltbundesamt 2020a]. To achieve this goal, the UBA collects environmental data to research the relationships between them and to forecast the future or advise the government. To do so, the UBA collects different categories of data such as climate data, energy data, air data, water data, and many others. However, the collected data is available and free of charge to any end-user. This is the reason why it is chosen as one of our data sources. For our use case, we only focus on air data, especially on nitrogen dioxide (NO_2). NO_2 is one of a group of highly reactive gases known as oxides of nitrogen or nitrogen oxides (NO_x) and is an air quality indicator of combustion emissions associated with respiratory diseases, particularly asthma, leading to respiratory symptoms such as coughing, wheezing or difficulty breathing [Lamsal et al. 2013]. Furthermore, nitrogen oxides ($NO_x = NO_2 + NO$) contribute to the formation of fine particulate matter (PM2.5) and ozone, both of which are harmful to human health and the environment. To extract the NO_2 data, we implement a flow similar to those described above. The flow is composed of four bricks (see Figure 4.17).

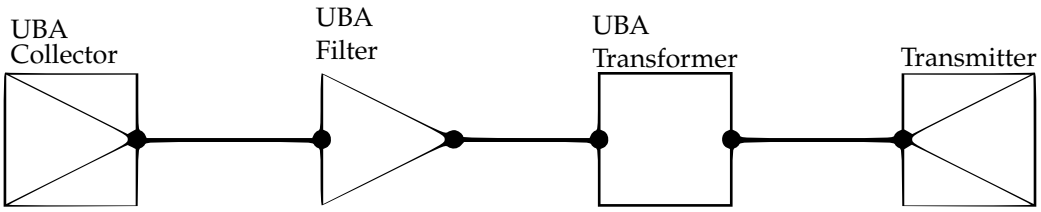


Figure 4.17. The UBA data flow

4.3.2 The UBA Collector Brick

The UBA provides a REST API to request data from their servers. Therefore, we, first of all, implement an inlet brick (*UBA Collector*) that connects to the UBA server via REST. This is done with the Python library requests. After the connection to the brick is established, we first collect metadata about each station in Germany such as station id, street name, and city name. The collected metadata is then used to extract air data such as carbon monoxide, ozone, particulate matter, sulfur dioxide, and nitrogen dioxide. Since the data is all mixed and we are only interested in nitrogen dioxide, we, send the receive packets to the next brick for them to be filtered.

4.3.3 The UBA Filter Brick

To filter the data, we implement a filter brick (*UBA Filter*) that would be connected to the inlet brick. Since the UBA does not provide any information about the reliability of the data, we, filter the data just by extracting the nitrogen dioxide data from all the others. After filtering, the data is forwarded to the next brick to be transformed in, such a way that it should respect the format described in Section 4.1.4.

4.3.4 The UBA Transformer Brick

None of the above-mentioned transformer bricks is capable of transforming the data coming from the UBA filter brick. Therefore we implement a new brick for this purpose. We implement a new brick (*UBA Transformer*) which is directly connected to the UBA filter brick. Each incomming data point is then transformed with respect to the table format as describe in Section 4.1.4. After the transformation, the data is written into the database by the *Transmitter* brick (see Section 4.1.5). The last step consists of joining all the flow to form a unique flow as shown in Figure 4.18.

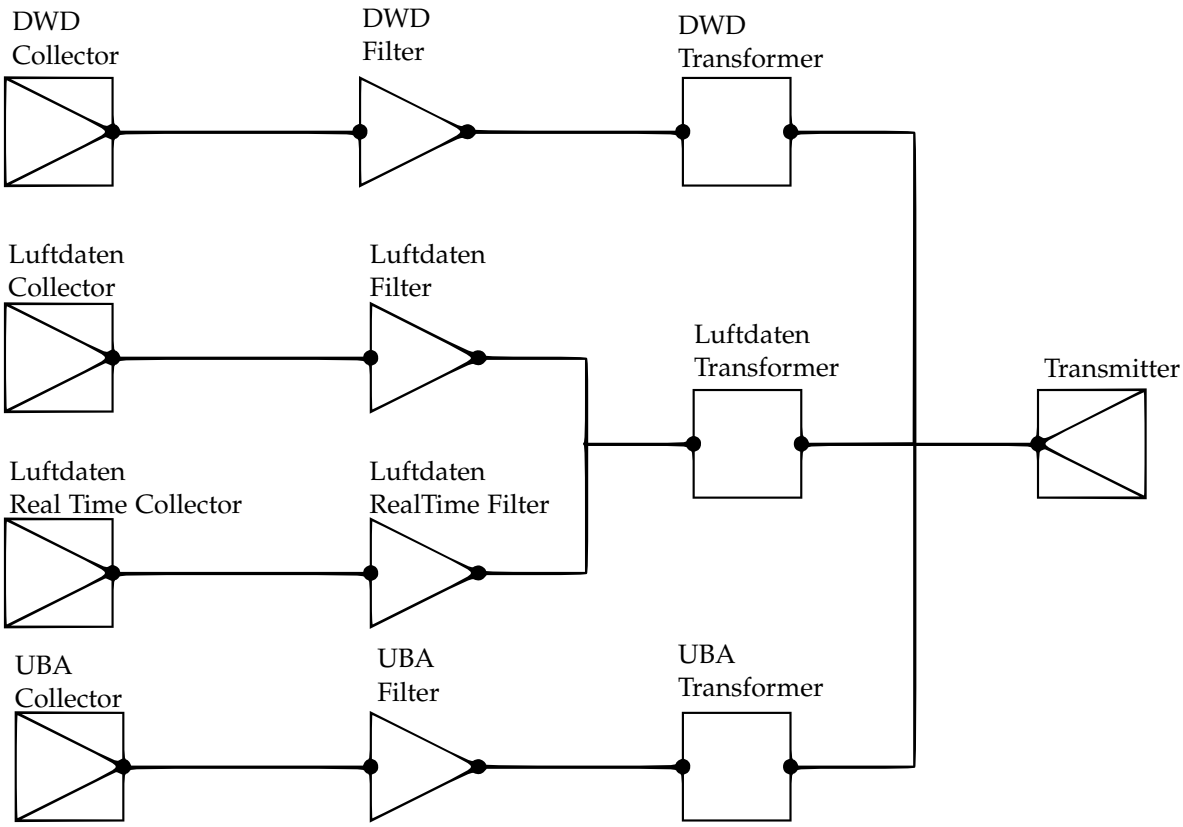


Figure 4.18. The DWD, UBA, and Luftdaten service data flow

4.3.5 Visualization

At the end of the UBA flow, data is saved into the InfluxDB database and read by the Grafana tool for visualization. To make the entire visualizations easier to use, a panel with the title NO_2 is added to the UBA dashboard. Furthermore, a line chart that visualized the values of NO_2 over time is drawn in the panel. This is done by using the Grafana user interface, which enables us to make queries to the database. With the visualization we are able to see the changes of NO_2 in particular stations (see Figure 4.19).

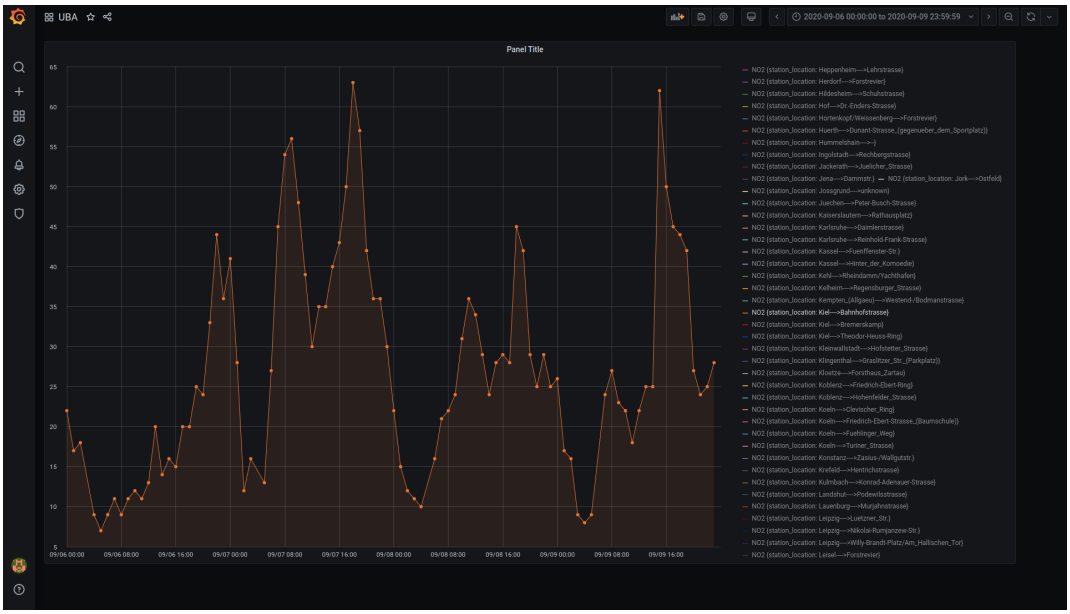


Figure 4.19. Visualization of NO_2 at the station situated in Kiel at Bahnhostrasse (2020-09-06 to 2020-09-09)

It is also possible to choose many stations and observe the correlation between them (see Figure 4.20).

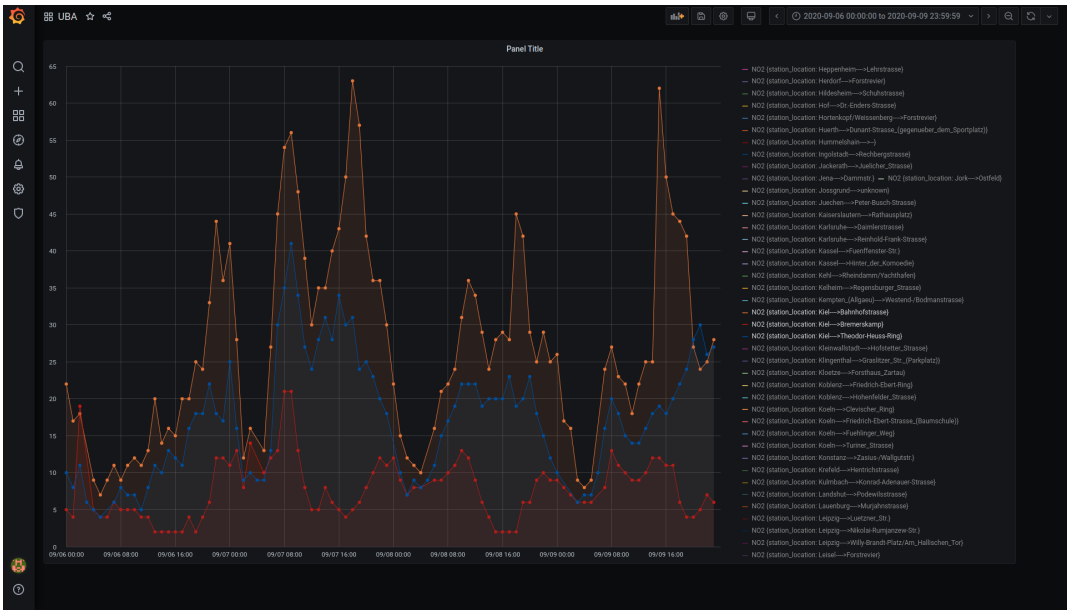


Figure 4.20. Visualization of NO2 at the stations situated in Kiel at bahnhofstrasse and bremerskamp (2020-09-06 to 2020-09-09)(screenshot)

Orange color:	Station at Bahnhofstrasse
Blue color:	Station at Theodor-Heuss-Ring
Red color:	Station at Bremerskamp

4.4 Implementation of Use Cases for the Experimental Scalability of the Titan Platform

In this section, we implement all the flows involved in the experiment of Chapter 5. All the flows are implementations of the use cases identified in the Theodolite method [Henning and Hasselbring 2020b]. For each use case, there exists a corresponding dataflow architecture a.k.a topology. Therefore, each implemented use case is carried out with regard to the corresponding topology. The use cases are of different complexity and a good starting point to evaluate the scalability of the Titan platform. However, most of the implementations are adaptations to those presented in Section 4.2.

4.4.1 Use Case UC1: Database Storage

The first use case we implement is UC1. In UC1, data events or messages are stored permanently, for instance, in an NoSQL database. But before the storing stage, they are different operations that need to be performed. First, raw data is read from external data sources or generated by different programs and put at the beginning of the dataflow. After this, they are transformed by the stream processing engine into different formats — often required by the chosen storing system — and are written into the database.

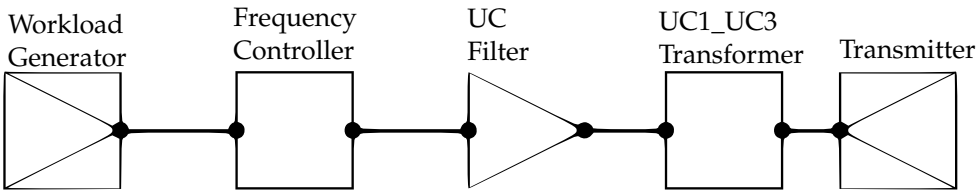


Figure 4.21. Our Dataflow architecture for UC1: Database Storage

Most of the dataflow architectures presented above are similar to the dataflow architecture for UC1. Hence, we mostly modify the existing dataflows rather than implementing new ones. For the implementation of the UC1 dataflow, we chose and modifier the Luftdaten.info dataflow. The modifications consist of removing the restriction in the *Luftdaten Collector* brick, which involves the, collection of the data of the past 24 hours. Since the Luftdaten.info provides historical data up to the year 2015, we now use the *Luftdaten collector* (renamed to *Workload Generator*) brick to extract data from a time range that can be set by the user. However, the time range must be situated between 2015 and yesterday. This is done to increase the frequency of incoming records in each brick of the flow.

To be able to control the frequency of records injected into the flow, we append a new brick to the *Workload Generator* brick. We name the brick *Frequency Controller*, and it enables the user to set the frequency (number of records per second) across the flow. For instance, if the user sets the frequency to 1 000, 1 000 records will be sent to the next each second.

Most of the historical data comes in different formats, and therefore, *Luftdaten Filter* (renamed to *UC Filter*) and *Luftdaten Transformer* (renamed to *UC1_UC3 Transformer*) are also modified to receive all incoming formats (see Figure 4.21). Owing to the stateless topology of this use case, we avoid implementing a real database in the *Transmitter* brick. In our experience, this avoids primarily testing of the writing capabilities of the database.

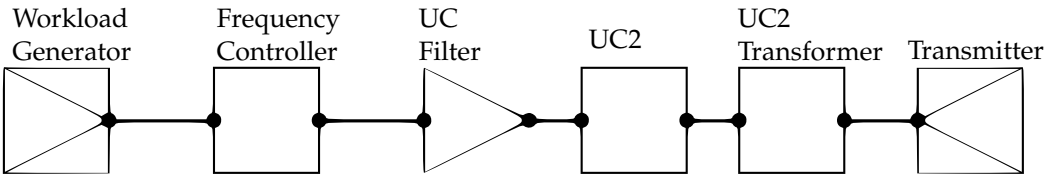


Figure 4.22. Our Dataflow architecture for UC2: Hierarchical Aggregation

4.4.2 Use Case UC2: Hierarchical Aggregation

Data aggregation is referred to as the process of gathering data and presenting it in a summarized format. Typically, any aggregation computes the average, maximum, or minimum of measurements, and it is useful for saving resources or providing better user experience. The next use case we consider consists of aggregation of data in a hierarchical way. For instance, the measurements of sensors can be aggregated to groups of sensors; these groups can again be aggregated to larger groups and so forth [Henning and Hasselbring 2020a]. For the experiment in Chapter 5, we implement a simplified version of this use case. However, more details about the entire use case can be found in [Henning and Hasselbring 2020b].

Our implementation of UC2 aggregate data hierarchically, and the hierarchy is composed of three levels. The lowest level is composed of a list of all sensors across Germany, while all the sensors of a city are grouped together at the second level. After checking if all the sensors of the chosen city are available in the group, a statistical operation (maximum) is applied to their values to generate a unique value for that city. Since each city in Germany is now marked with a unique value, the cities are recursively grouped into states. Finally, the same statistical operation, which is applied to compute the value of each city, is applied to compute those of each state. This implementation is done in a separate brick call *UC2*. Since the outgoing data of this brick does not match the incoming data of the *UC1_-UC3 Transformer* brick, a new brick for the data transformation (*uc2 Transformer*) is also implemented. The entire dataflow of this use case is shown in Figure 4.22.

4.4.3 Use Case UC3: Downsampling

The main idea of this use case is to aggregate multiple records within consecutive and non-overlapping time windows. However, UC3 requires that data should be aggregated within the windows of a fixed (but configurable) size that succeed each other without gaps. Consequently, the message frequency is considerably reduced depending on the size of the time window. Therefore, UC3 is also referred to as downsampling.

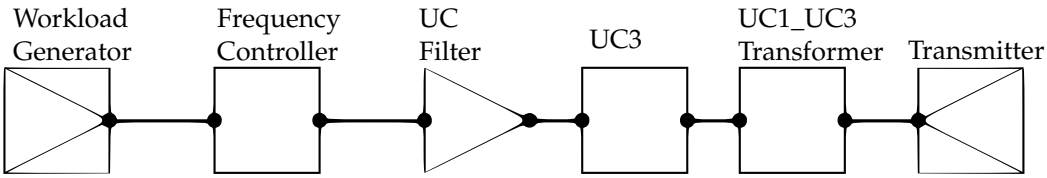


Figure 4.23. Our Dataflow architecture for UC3: Downsampling

The topology of UC3 requires that, data should be read records after records from an input stream and assign to a time window. Afterwards, statistical operations such as maximum and minimum are applied to each time window to compute a single value from the group of value received. Finally, the results of the aggregations are written to an output stream. To implement this use case, we modified the implementation of the UC2 dataflow by replacing the brick *UC2* with the brick *UC3*. This brick is responsible for setting the time window size (done by the user), aggregating records into groups of values, and applying aggregation operations to each group. Moreover, the outgoing data from the brick *UC3* are similar to the incoming data of the transformer brick in UC1. That is why we use a unique brick for both transformations (*uc1_uc3 Transformer*). Figure 4.23 shows the dataflow architecture of our implementation.

4.4.4 Use Case UC4: Aggregation based on Time Attributes

The last use case we consider is UC4. This use case is similar to the above described use case UC3 in the sense that it also implements a temporal aggregation. Records with the same time attribute are aggregated together, and a time attribute can, for instance be the hour of day, day of week, or day in the year. Unlike non-overlapping windows in UC3, the aggregation in UC4 is done within overlapping windows.

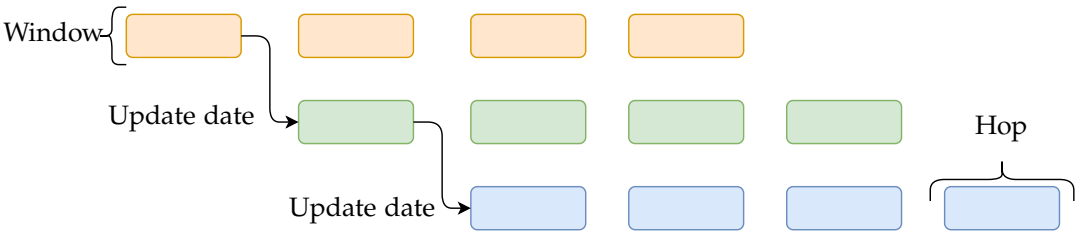


Figure 4.24. Illustration of sliding windows

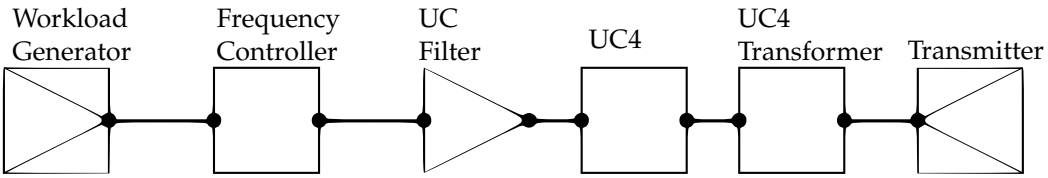


Figure 4.25. Our Dataflow architecture for UC4: Aggregation based on Time Attributes

To implement UC4, we must first understand the idea of sliding windows, which are windows based on time intervals and are characterized by the window size and its advance interval (a.k.a. hop). The advance interval specifies how much a window moves forward relative to the previous one. Therefore, a data record may belong to more than one window. UC4 is based on sliding windows, its topology requires that records should first be read from the input stream, and then the selected time attribute is extracted from the record's timestamp, and a key for each record is set. Next, the records are duplicated for each sliding window it is contained in. All measurements with the same key and the same time attribute are aggregated for each sliding window by computing, for instance, the summary statistics sum, count, minimum, maximum, or average. Finally, the results of the aggregations are written to an output stream.

Our implementation of this use case involves the geographic coordinate of each sensor as a key and time attributes as the days of a week. To implement sliding windows, we define a window as a list of hops. This means that the window size is equal to the maximum number of hops that a window can contain. Each hop is defined as a dictionary containing data from a specific time interval. A dictionary is used to ensure unique keys. Once the length of the list is equal to the window size, all hops of the list are combined to compute aggregations belonging to that window. After the aggregation has been computed, the computation of the next window starts. The next window starts one hop after the start of the previous window (see Figure 4.24). Therefore, we reuse the list of hops of the previous window by deleting the first hop. Now, the list of hops of the next window has a size equal to the window size minus one, and we just need to collect the data of the last time interval of the window to have complete data for the next window. This process is carried out continuously as long as data is available. The implementation described above is done in a separate brick called *UC4*, and data coming out of this brick are of different formats and cannot be transformed by any of the existing bricks. Therefore, we implement a new brick call *UC4 Transformer* to transform the data into a format that can be accepted by the database. The overall dataflow is shown in Figure 4.25.

Chapter 5

Experimental Scalability Evaluation

Software evaluation is a type of assessment that seeks to determine if any software or a combination of software programs is the best possible fit for the needs of the end user. The demand for qualitative, reliability, and easy to integrate software for existing systems is continuously growing. Besides, software programs in the context of Big Data must be able to scale depending on the workload. Consequently, the evaluation of such software aspects is important. In this chapter, we pay more attention to the scalability of the Titan platform.

In Section 5.1, we present the methodology used across the experiment. Section 5.2 describes the evaluation environment and Section 5.3 the results of the experiment. Finally, Section 5.4 presents the validity of the experiment.

5.1 Methodology

5.1.1 Goals

The widespread growth of Big Data and the evolution of Internet of Things (IoT) technologies has increased the need for real-time insights into data. In response to this, many stream processing engines were developed and several benchmarks conducted to identify which of such engines perform best in which scenario. However, as in our case, the stream processing engine has already been chosen, and comparing it with other such engines does not help us any further. Therefore, we consider different approaches, such as benchmarking different deployment options to find the optimal ones or benchmarking the scalability of different execution environments to help find a suitable execution environment. Moreover, scalability is the centerpiece when dealing with Big Data. Therefore, it is very important to evaluate how the number of processing instances evolves with the increasing workload.

As mentioned above, scalability benchmarking on stream processing engines typically compares two or more different such engines to determine the suitable one for a specific use case. But owing to new researches, a new method to benchmark a single stream processing engine has been developed. Theodolite is a method used to benchmark the scalability of stream processing engines in microservices [Henning and Hasselbring 2020b]. It uses a specification-based approach, which means use cases are defined on functional or business requirements instead of the technical ones. For our evaluation, we implement in Section 4.4 all the use cases identified in the Theodolite method in order to measure the scalability of

the Titan platform. The scalability measurement is, then, done by applying diverse and increasing workload dimensions to observe how the demand for resources evolves in each implemented use case.

5.1.2 Workload Dimension

A system is considered scalable if it does not need to be redesigned to maintain effective performance during or after an increasing amount of workload. By increasing the amount of workload, additional resources are sufficient to maintain the performance of a scalable system [Concepta 2020]. Therefore, to evaluate the Titan platform, we apply the workload dimension *Message Frequency* to the implemented use cases in Section 4.4. The workload dimension message frequency describes how many messages are read from an input stream per time unit. This workload can be applied to all the implemented use cases.

5.1.3 Description of the Experiment

In this section, we describe how the individual parts of our approach is going to be evaluated. As mentioned above, we apply a modified version of the Theodolite method for our evaluation. This method is based on use cases, and its application requires that each implemented use case should be benchmarked individually. For each use case, we, therefore, determine the minimum number of brick instances necessary to process a workload without reaching high traffic of data across the Titan platform. High traffic of data in our case means a persistent queuing up of records within the Titan platform. Since the size of queues continuously changes in such a system, we consider records to persistently queue up if the number of messages in queues increases over a large period. To determine whether this is the case, we use the monitoring functionality of the Titan platform. To monitor a flow, the Titan platform comes with a Grafana dashboard (see Figure 5.1), which shows diverse metrics of flows at runtime. These metrics also include the number of elements in the queue between two bricks. Therefore, if the length of a queue increases over a large period, we conclude that messages are queuing up within the Titan platform. This process enables us to determine the minimum number of instances that is necessary to process a workload without having high traffic of data across the Titan platform.

Figure 5.1. The Titan platform monitoring dashboard

On the Titan platform, each flow of Section 4.4 is built separately and configured. By default, each brick comes with three parameters: the `maximum number of running instances`, the `maximum idle time` in seconds, and the `maximum flow packet` in the queue of the considered brick. These parameters have the default values of 1, 10, and 25, respectively. At the beginning of the experiment, we leave the parameter `maximum number of running instances` unchanged and continuously inject a constant frequency of messages into the flow to the chosen use case. If the length of the queues does not continuously increase over time (see Figure 5.2), we increase the message frequency in the *Frequency Counter* brick. But, if the length of the queues grows up continuously over time (see Figure 5.3), we reduce the message frequency in the *Frequency counter* brick. We execute both actions (increasing and decreasing the message frequency) interchangeably until we determine the minimal number of instances that is sufficient to process a workload without reaching high traffic of data. After the determination of the minimum number of brick instances for a chosen frequency, we manually increment the parameter `maximum number of running instances`.

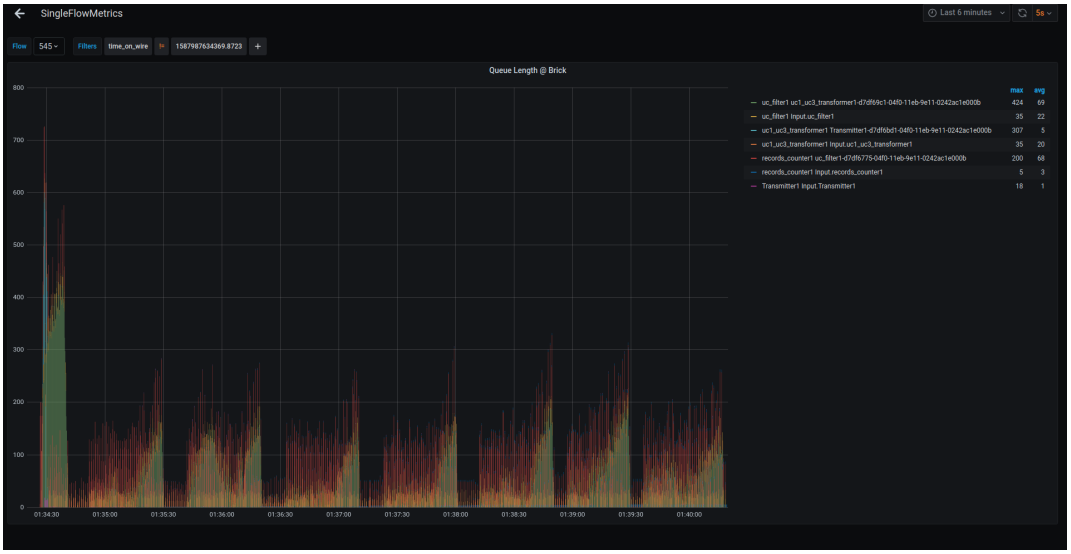


Figure 5.2. Representation of a non-increasing length of queue across a flow

In contrast to the first parameter, the values of the next two parameters are modified at the beginning of the experiment and kept unchanged throughout an experiment. If the value of the parameter `maximum idle time` at runtime is above the value set by the user, it causes the brick to restart. This leads to loss of information. To avoid this behavior, we set the values of this parameter to the highest integer values that it can accept. As for the parameter `maximum flow packet`, its value is set to 250. Setting this value to 250 enables us to quickly activate the automatic scaling mechanism of the Titan platform. The Titan platform only starts scaling when the queues between the bricks highly increase to the point that their capacities can be exceeded. Moreover, we set the inlet brick *Workload Generator* to collect data from the years2016 to 2020 so that we have enough records for our experiment. In all experiments, we determine the minimal number of instances necessary to compute a workload without having high traffic of data on the Titan platform. After the minimal number of brick instances capable of processing a load is determined, we consider $P(x, y)$ as a point with the coordinate $x =$ number of record per seconds (load) and $y =$ number of instances. Afterwards, we manually increment the number of brick instances and the process is continuous. Finally, all the points P(load, instances) collected during an experiment are plotted on a graph. In the following, we describe the individual scenarios in detail.

Scenario 1 This scenario serves to evaluate all the implemented use cases individually by using the workload dimension message frequency. As mentioned in Section 4.4, since storing data in a real database can become the bottleneck for use case UC1, we do not implement a real database in the transmitter brick. For the use case UC2, to avoid

aggregating data over four years, we set the time window to 24 hours. Use case UC3 is configured with an aggregation time window of 30 days. In UC4, the time window size is set to two weeks by starting a new window every seven days. The time attribute, for which data is aggregated is the day of the week.

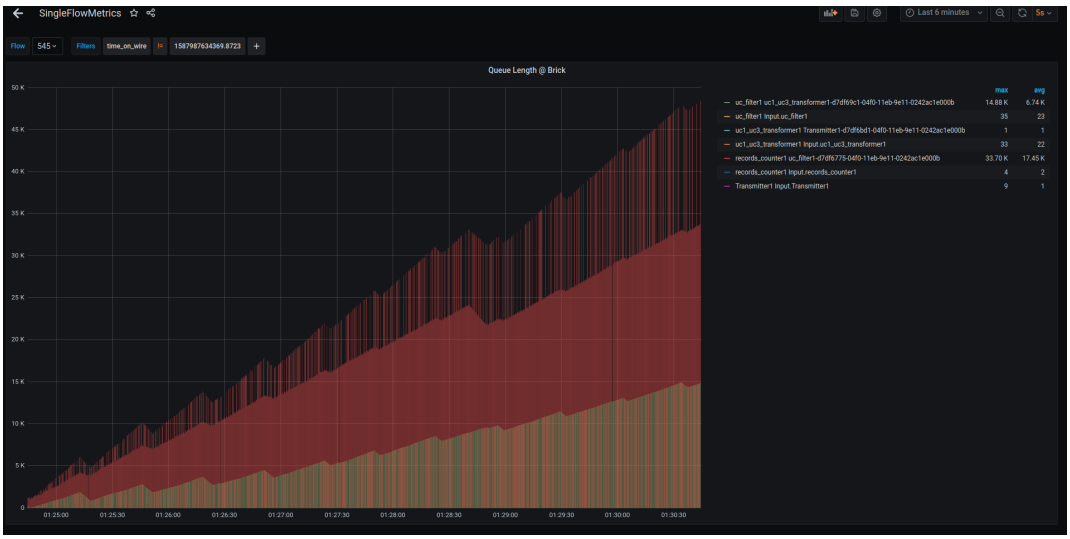


Figure 5.3. Representation of a increasing length of queue across a flow

Scenario 2 The next scenario we consider is to change the parameter `Time Window Size` in use cases UC3 and UC4. For this, we update the time window of UC3 from 30 days to 60 days; for UC4, the time window size is set to four weeks by starting a new window every seven days. The time attribute for which data is aggregated is the day of the week.

Scenario 3 Our last scenario consists of increasing the overlapping windows in use case UC4. For this, the time window size is set to six weeks by starting a new window every seven days. The time attribute for which data is aggregated is the day of the week.

5.2 Experiment Setup

In our evaluation, we use a Docker Compose[1] file that contains all the images necessary to run our software. To monitor the length of queues within the Titan platform, the Titan platform saves metrics about each queue into an Elasticsearch[2] database. Therefore, we also add an image for Elascticsearch in our Docker Compose file. Afterwards, we pull and

[1] `https://docs.docker.com/compose/`

[2] `https://www.elastic.co/blog/found-elasticsearch-as-nosql`

start all services on a private cloud of *Kiel University's Software Performance Engineering Lab*[3], which has the configurations listed Table 5.1.

Table 5.1. Hardware configuration of the private cloud

CPU	Intel(R) Xeon(R) CPU E5-2650
Clock Frequency	2.00 GHz
Number of Cores	32
Number of Threads	64
RAM	128 GB
OS	Debian GNU/Linux 10 (buster)
Kernel Version	4.19.0-8-amd64

The visualization runs in the user web browser. We evaluated it on a desktop with the configurations listed in Table 5.2

Table 5.2. Hardware configuration of the user

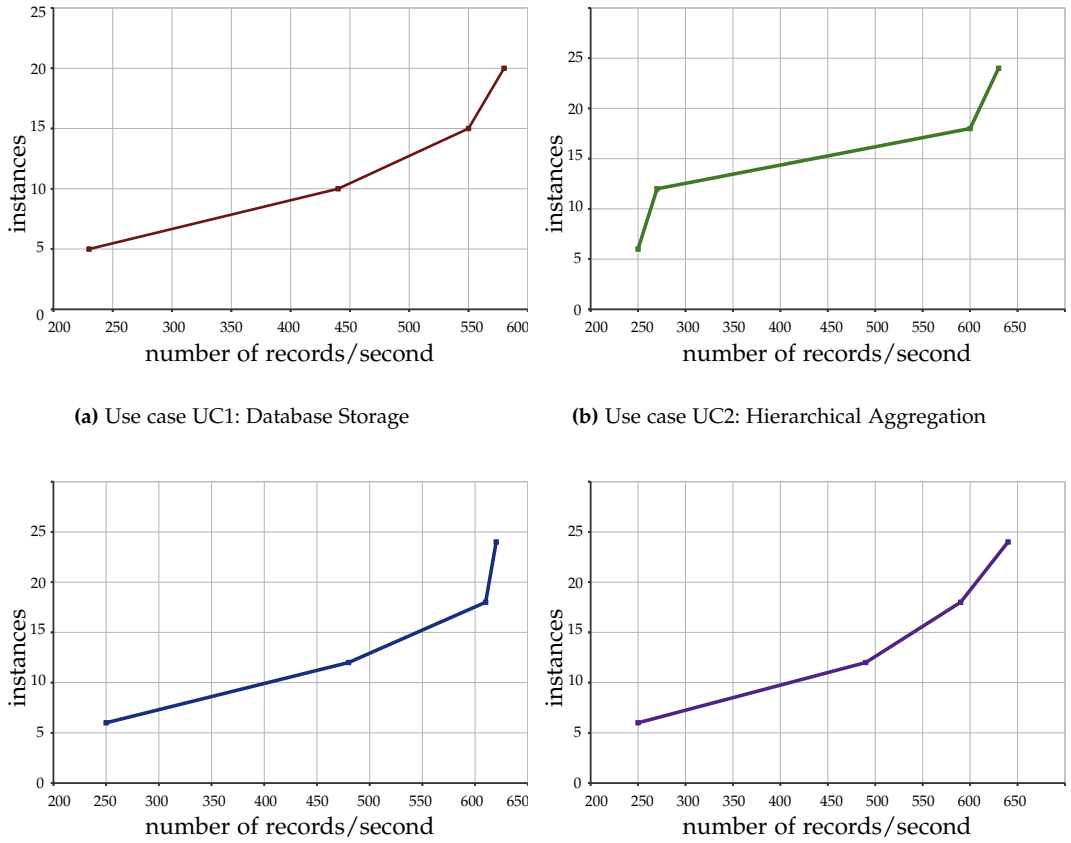
CPU	Intel(R) Core(TM) i5-4460T
Clock Frequency	1.90GHz
Number of Cores	4
Number of Threads	4
RAM	8 GB
OS	Ubuntu 18.04.5 LTS
Kernel Version	5.4.0-48-generic
Browser	Google Chrome Version 85

5.3 Results and Discussion

5.3.1 Scenario 1

Figure 5.4 shows the results of the experiments in the first scenario. We observe that for all use cases, when increasing the number of brick instances, we are able to process more messages. However, the number of messages processed by a specific number of brick instances highly depends on the chosen use case. To have a better understanding of this behavior, we determine the number of active instances of each use case configuration at runtime. Setting the parameter `maximum number of running instances` of each brick to a number n does not necessarily mean that each brick uses n instances at runtime. Some bricks need more processing power than the others. Therefore, we also present the exact number of brick instances activated at runtime in Table 5.7. The number of active instances is determined using the Grafana dashboard of the Titan platform.

[3] `https://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab-spel`



(a) Use case UC1: Database Storage

(b) Use case UC2: Hierarchical Aggregation

(c) Use case UC3: Downsampling

(d) Use case UC4: Aggregating Time Attributes

Figure 5.4. Scalability benchmark results for scenario 1

In all the tables, we observe that independent of the message frequency, the number of active instances for the Workload Generator and frequency controller brick is always set to 1. We explain this aspect as follows: to ensure the permanent availability of data points in the frequency controller brick, the Workload Generator sends a batch of at least 100 000 data points as a list to the frequency controller brick. While The frequency controller brick consumes chunks (e.g., 250 msg/s) of the batch of data, the Workload Generator starts the production of a new batch of data points. Nevertheless, this means that the queue between the Workload Generator and the frequency controller is of size $250 \times 100\,000$. Therefore, autoscaling is never activated at this point and hence the number of active instances is always 1.

Table 5.7. Number of active instances of use case configurations in scenario 1

Table 5.3. Number of active instances at runtime for UC1

Load	Workload Generator	Frequency Controller	UC Filter	UC1_UC3 Transformer	Transmitter
230 msg/s	1	1	1	1	1
440 msg/s	1	1	2	2	2
550 msg/s	1	1	3	3	3
590 msg/s	1	1	4	4	4

Table 5.4. Number of active instances at runtime for UC2

load	Workload Generator	Frequency Controller	UC Filter	UC2	UC2 Transformer	Transmitter
250 msg/s	1	1	1	1	1	1
270 msg/s	1	1	2	1	1	1
600 msg/s	1	1	3	2	2	2
630 msg/s	1	1	4	2	2	1

Table 5.5. Number of active instances at runtime for UC3

load	Workload Generator	Frequency Controller	UC Filter	UC3	UC1_UC3 Transformer	Transmitter
250 msg/s	1	1	1	1	1	1
480 msg/s	1	1	2	2	1	1
610 msg/s	1	1	3	2	2	1
620 msg/s	1	1	4	2	2	1

Table 5.6. Number of active instances at runtime for UC4

load	Workload Generator	Frequency Controller	UC Filter	UC4	UC4 Transformer	Transmitter
250 msg/s	1	1	1	1	1	1
490 msg/s	1	1	2	2	1	1
590 msg/s	1	1	3	2	2	2
640 msg/s	1	1	4	3	1	1

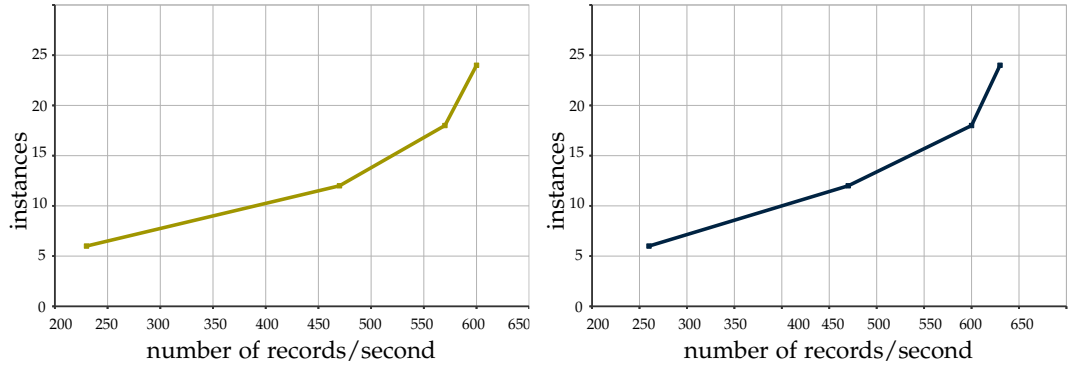
Table 5.3 shows the number of brick instances for use case 1. We observe that each brick situated after the frequency controller uses its maximum number of instances set by the user. This is explained by the stateless topology of UC1. Since use case UC1 is stateless, the number of instances used by the brick $B1$ to produce n messages is the same number of brick instances use by the brick $B2$ (adjacent to $B1$) to consume n messages. In the other use cases (UC2, UC3, UC4), we observe that only the UC filter activates all its instances at runtime. We explain this behavior by the fact that the bricks that are connected directly to the UC filter have a state. Having a state enables the bricks to consume data faster than the UC filter can produce them. Hence, the number of activated brick instances is less than the maximum available instances set by the user. However, to maintain a large amount of incoming data, the UC filter in these cases behaves similar to the UC filter in use case UC1 and activates all its available instances at runtime. Moreover, the transformer and the transmitter bricks need less (or in some cases the same amount) brick instances to process the amount of data coming from the bricks UC2, UC3, and, UC4. Since the bricks UC2, UC3 and, UC4 aggregate data (hierarchically or over a time window), this causes the outgoing data of those bricks to be considerably reduced compared to their incoming data. Hence, the number of activated brick instances of both the transformer and transmitter brick is also reduced.

We also observe that the length of the queues keeps growing even after the number of brick instances is set to a number greater than 4. Therefore, we consider four instances per brick as the bottleneck of our experiment.

5.3.2 Scenario 2

The results of Scenario 2 are depicted in Figure 5.5. The first observation is: in comparison with the first scenario, adding the window size does not have an immense effect on the trend of the resulting graph. However, the number of messages changes for each configuration. To have a better understanding of this behavior, we also determine the number of active brick instances at runtime. Table 5.10 shows our results.

Comparing Scenario 1 with Scenario 2, we observe that the behavior of the UC filter does not change. Furthermore, the application of a larger window size to UC3 and UC4 increases the number of brick instances activated by the bricks UC3 and UC4 in some configurations. A larger window size means for the bricks UC3 and UC4 more data to maintain and therefore more computing power. Moreover, the number of brick instances activated by the transformer and transmitter bricks are reduced. Adding the window size means that the bricks UC3 and UC4 generate fewer data points at runtime; therefore, the amount of data to be transformed also reduces.



(a) Use case UC3: Downsampling

(b) Use case UC4

Figure 5.5. Scalability benchmark results for scenario 2

Table 5.10. Number of active instances of use case configurations in scenario 2

Table 5.8. Number of active instances at runtime for UC3

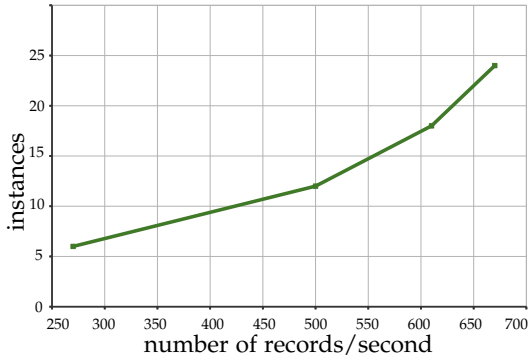
load	Workload Generator	Frequency Controller	UC Filter	UC2	UC2 Transformer	Transmitter
280 msg/s	1	1	1	1	1	1
520 msg/s	1	1	2	2	1	1
620 msg/s	1	1	3	2	1	1
650 msg/s	1	1	4	3	1	1

Table 5.9. Number of active instances at runtime for UC4

load	Workload Generator	Frequency Controller	UC Filter	UC3	UC1_UC3 Transformer	Transmitter
260 msg/s	1	1	1	1	1	1
470 msg/s	1	1	2	2	1	1
600 msg/s	1	1	3	3	1	1
630 msg/s	1	1	4	2	1	1

5.3.3 Scenario 3

Figure 5.6 shows the result of Scenario 3. The trend of the resulting graph is mostly similar to those of UC4 in Scenarios 1 and 2. In this Scenario, we also observe an increase in the message frequency for each configuration. Table 5.11 shows the number of activated brick instances at runtime. Similar to Scenarios 1 and 2, the behavior of the UC filter brick does not change. With a window size larger than the one in Scenario 2, we observe an increase in the activated number of instances of the brick UC4. This behavior meets our explanation of Section 5.3.2. The transformer and transmitter bricks activate as many instances as the UC4 bricks produce data. For a larger window size, there is less data produced by the UC4 Brick, and therefore, transformer and transmitter bricks activate just one or a maximum of two brick instances at runtime.



(a) Use case UC4: Aggregating Time Attributes

Figure 5.6. Scalability benchmark results for scenario 3

Table 5.11. Number of active instances at runtime for UC4 (Scenarion 3)

load	Workload Generator	Frequency Controller	UC Filter	UC4	UC4 Transformer	Transmitter
270 msg/s	1	1	1	1	1	1
500 msg/s	1	1	2	2	1	1
610 msg/s	1	1	3	3	1	1
670 msg/s	1	1	4	3	2	2

5.4 Threats to Validity

The validity of our results is mostly threatened by several factors. We focus on that in this section.

The use cases UC2, UC3, and UC4 require that the state of the brick should be saved at runtime. However, at the moment of our implementation, the Titan platform did not provide any method on how to save the state of a brick. So, we use a rather unconventional method to save the state of a brick: we, use a variable. The first consequence to use a variable as a state is that the results written by the transmitter bricks to the database are faulty. This behavior is caused by a common problem known in computer science as a race condition [Netzer and Miller 1992]. In case of a race condition, the variable used to save the state is called shared data. Since the use cases UC2, UC3, and UC4 use multiple instances at runtime, the shared data can be accessed by a brick instance while a different instance is currently working on it. Such concurrent processes often lead to data inconsistency. The second consequence of using a variable as a state is that it can also affect the entire evaluation. For instance, if there is high traffic of data across the Titan platform and many bricks instances are activated, each of those instances may try to read the shared data at the same time. This can result in a faulty bottleneck of the evaluation.

Moreover, the results of our experiments are mostly collected from different observations at different time intervals. This makes it difficult to exactly reproduce our results. For instance, for some configurations, we are able to collect results after observing the behavior of the Titan platform for 5 minutes. For others, results have been collected after observing the behavior of the Titan platform for 8 minutes or less. The observation time is not fixed, but it is situated in an interval of 3-8 minutes. Therefore, there is no guarantee that for a configuration, the conclusion we draw, for example, after 5 minutes of observation will be the same after 8 minutes of observation. We do not define a fixed number of waiting minutes before collecting the information because some configurations delivered results that were explicit from the very beginning, while the others needed more time.

Chapter 6

Related Work

In the domain of Big Data analytics, many works have been published to present methods on how to analyze environmental data. In this chapter, we focus on the presentation of those related to our approaches.

Meyer et al. [2019] presents a data science challenge that took place at the University of Rostock in Germany. The challenge was taken up by students and researchers of different universities. The ultimate task was the analysis of particulate matter pollution in major German cities and the associated driving ban and air pollution. The challenge was won by three groups of students. In the following, we give an overview of their approaches and explain how their work is related to ours.

The first group used machine learning to predict air pollution. But first, they needed to have access to data. For this reason, they chose Lufdaten.info as their main data source to collect temperature and humidity data from DHT22 sensors [Luftdaten Info 2015]. Particulate matter data was collected from SDS011 sensors [Luftdaten Info 2015]. To have more attributes for their analysis, they also collected wind direction, precipitation, and air pressure data from the German Weather Service (DWD). After the collection of the data from both the DWD server and the Luftdaten.info Server, they did their analysis in four steps, which are data cleaning, data fusion, data visualization, and finding insights. By looking into the datasheets of the sensors, they could determine their ideal working environment, which helped them in data filtering. Moreover, most of the collected data was filtered out as they decided to restrict their area of interest to Stuttgart. This step is comparable with what we do in our filter bricks. For data fusion, they joined tables containing SDS011 and DHT22 sensors' data to obtain an integrated dataset for the visualization. The visualization used google maps and heatmaps in such a way that choosing a region on the map generated heatmaps of the PM10 concentration in that zone. This is similar to our visualization with the Grafana world map panel. By using their application, they could discover useful insights that helped them suggest helpful recommendations to reduce accidents in future. For instance, they observed that the average PM10 value is four times higher in winter when compared to summer.

The second group of students decided to evaluate the impact of a driving ban in the city of Stuttgart and Dresden. To achieve this, they divided their analysis into three main parts, which are data cleaning, data visualization, and finally, they gave a short introduction to a forecasting model. To filter particulate matter data, they needed information about the humidity of the area from where the data had been collected. As mentioned in our work

(see Section 4.2.3), particulate matter sensors provide reliable information only when the humidity is at a certain level. Therefore, they removed all the collected records from those places where the humidity was above 70%. For the visualization, they used time-series plots and distribution maps. To show that morning and evening rush hours influence the particulate matter concentration, they created a time-series plots of particulate matter sensors over 24 hours. However, their domain expert characterized this correlation as spurious. With more research, they were able to conclude that the solar intensity affects the concentration of particulate matter. For the second visualization, they used a heatmap associated with a geographical map in order to show the concentration of the particulate matter. At the end, they used a long short term memory (LSTM) neural network to build a predictive model for the particle concentration.

The third group of students focused on diverse data sources to explain air pollution. Therefore, besides their main data source (Luftdaten.info), which provides particulate matter data, they also used different data sources to collect weather, street, event, and traffic data. After that, they pre-processed the data. The pre-processing step consists of four principal operations, which are clustering, binning, cleaning, and aggregation. After the pre-processing stage, the data was analyzed by automatically detecting deviating patterns. The analysis differentiated the data points with a high concentration of particulate matter from normal readings and proposed an explanation based on the highly correlated descriptive features. At the end, with the help of external data sources, the group was able to conclude that weather, traffic flows, and public events have an impact on quality of air .

To contribute to the fight against climate change, a group of researchers [Nunes et al. 2013] analyzed climate data to forecast climate change in order to prevent and mitigate the bad consequences of human activities. For the analysis, they collected data coming either from the networks of meteorological stations or from climate models. We relate this work to ours in the sense that it analysed various climate variables, to help domain experts draw conclusion about climate change from the resulting visualizations. However, in their work, they used many other climate variables: for the analysis, they created a multidimensional data stream in such a way that each climate variable defined a dimension of the stream. This was achieved by combining the time-series defined by different climate variables. Moreover, to analyze the streams of data, they applied the fractal theory [Barbara and Chen 2009]. By analyzing the variation of the fractal dimension over time, they were able to identify the existence of the correlations involved. Our work uses a simple approach to understand the correlation between climate variables. Using a Grafana dashboard, we visualize, for instance, particulate matter and humidity data on the same panel, thereby helping domain experts to understand the correlation between the two variables. However, fractal-based approaches usually allow fast processing with linear or quasi-linear scalability regarding the number of data elements and attributes. Moreover, such approaches also rely on the partition of the data space, the individual analysis of each partition, and the integration of the results, following the well-known divide-and-conquer method. Therefore, they concluded that the analysis based on the fractal theory are well-suited for the analysis

of very large collections of data.

With the concern to predict the weather and help people in organizing their daily routines, Adam et al. [2017] proposed a method of analyzing weather data based on the MapReduce algorithm [Dean and Ghemawat 2008]. The MapReduce algorithm is made up of three consecutive steps, which are the *map function*, *shuffle function*, and *reduce function*. The first step consists of collecting datasets and dividing them into smaller datasets. This step is comparable to the implementation of our inlet bricks, which connect to a data source, collect a set of data, and send them point by point to the next brick. However, the map function is also responsible for computing each collected sub-sets in such a way that the output of the entire function is a set of keys and values. At the end of the first step, the map function outputs are used as inputs for the second step. This method is similar to how we build flows in our work. The shuffle function is divided into two sub-tasks, which are merging and sorting. During the merging task, the data-sets having the same keys are merge, and the result is used as input for the sorting task. This method is also similar to our implementation, where each brick is responsible for a task. In the final step (reduce function), the data coming from the shuffle function is reduced in such a way that, at the end, the dataset consists only of points of key and value. Based on the MapReduce algorithm, the authors were able to analyze the weather data and accordingly predict the temperature.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Our environment is always changing and that is why monitoring it can be of great benefit for humanity. In this work, we present an approach to how environmental data can be continuously analyzed using the Titan platform. In our approach, we use three data sources that provide diverse environmental data. We also restrict the region of interest to Germany.

The three data sources used in this work are the German weather service (DWD), the Luftdaten.info, and the German environmental federal office (UBA). Together, they provide wind direction, air temperature, particulate matter, humidity, and nitrogen dioxide data. The connections to the data sources are established with the help of application programming interfaces such as the RESTful Web Services or the File Transfer Protocol. Therefore, we first presented the terminologies of both interfaces and apply them throughout this work. Since we deal with a large volume of unstructured and structured data, it is difficult or almost impossible to process such data using traditional methods. Therefore, we use Big Data methods for the analysis. Consequently, we define the concept of Big Data and apply most of the steps necessary for a Big Data analysis.

Our analysis is done with the Titan platform, which is an Industrial DevOps application. For this reason, we define and explain the concept of Industrial DevOps and describe how it is applied to the Titan platform. Since we deal with Big Data and that scalability is important, we take a closer look at the architecture of Titan to ensure that it can handle a huge amount of data. Furthermore, Titan uses data flows to analyze data; therefore, we define the concept of flow-based programming and explain how it is applied to Titan.

For each above-mentioned data source, we implement one or more dataflows. Each flow is composed of multiple modular components, known as bricks, that are connected. To have a fluid communication between the bricks, the Titan platform uses the UJO data object notation in order to facilitate the communication between bricks. Therefore, we present the UJO language and explain how its use across the platform.

We design the architecture of our software, which is essentially composed of the Titan platform, a database, and a visualization tool. After the analysis on the Titan platform, the results are saved in an external InfluxDB database and finally visualized using Grafana. For each data source, we create a dashboard to visualize the results of our analysis.

We evaluate the scalability of the Titan platform by applying the Theodolite method. For

this, we first implement all the use cases identified by this method. Next, we apply diverse workloads to identify the minimum number of brick instances within a flow necessary to process a workload without reaching high traffic of data across the Titan platform.

7.2 Future Work

We plan to extend our work to all countries across the world. However, for particulate matter data, some regions in the world do not have any sensor to collect the concentration of particulate matter in the air. Therefore, the first step would be to motivate the population of such regions to instal low-cost sensors like the ones proposed by the Luftdaten service[1]. After this step, we extend our approach of Chapter 4 to collect sensor data from all possible stations. Since we receive a huge amount of data, we need more computing and storing resources. This new approach enables a better understanding of air pollution in all regions in the world, and domain experts can use our results to have a global understanding of the climate and accordingly propose preventive measures.

The World Health Organization (WHO) estimates that 4.6 million people die each year from causes directly attributable to air pollution. Since our main goal is to reduce this huge number of deaths, we intend to develop a mobile application that triggers an alert to a user situated in a region where the air quality can be dangerous for human health. With the alert, we would ask the user to either change his/her current position or put on an air purifier mask. To know when the air quality is bad for human health, we would also need the result of the analysis of domain experts.

We also intend to extend the evaluation of the Titan platform by deploying it in a distributed way on multiple nodes. In this way, we can have many control peers running simultaneously. Moreover, we intend to intensively benchmark the Titan platform by effectively applying the Theodolite method. To achieve this, we follow the benchmarking framework architecture of Theodolite, which consists of six steps. The first step requires that the use cases to be benchmarked should be implemented. In case of the implementation of use cases, we would consider the modification of those found in Section 4.4. The modification comprises the use of the interface provided by the Titan platform in order to save the state of a brick. After this, the Titan platform, the execution environment, and the messaging system (Kafka) should be configured. The configuration of the bricks would be similar to those used above. However, for the execution environment, we would use more RAM and more CPU cores to avoid our available hardware of becoming the bottleneck of our experiment. The third step of the framework consists of choosing a workload dimension that would be used for the experiment. Additionally to the workload dimension considered in the experiment of Chapter 5, we also apply different workload dimensions identified in the Theodolite method. For instance, the workload dimension *amount of time attribute values* [Henning and Hasselbring 2020b] is based on the time attribute and can only be applied

[1] https://luftdaten.info/

to the use case UC4. When applying this workload dimension, the number of outputs result is highly influenced by the chosen time attribute. For example, if the selected time attribute is set to the day of the week, each key would produce seven results. The fourth step of the framework consists of implementing a mechanism to generate workloads. For this, we reuse our implemented workload generator (see Section 4.4). In the fifth step, the configured dimension to be tested is associated with a list of workloads. Finally, we would start our experiment to determine the minimum number of brick instances that are needed to avoid high traffic of data across the Titan platform.

Appendix A

Project Overview

Our implementations are save on a git repository and can be found at `https://git.se.informatik.uni-kiel.de/thesis/cedric-tsatia-tsida-msc`. Implementations of the experiment are on the same repository on the branch evaluation.

The artifacts necessary to repeat the conducted experiment and run all flows implemented in this thesis are found at `https://zenodo.org/record/4117704`.

Bibliography

[Adam et al. 2017] K. Adam, M. A. Majid, M. A. I. Fakherldin, and J. M. Zain. A Big Data Prediction Framework for Weather Forecast Using MapReduce Algorithm. *Advanced Science Letters* 23.11 (2017), pages 11138–11143. (Cited on page 59)

[Bader et al. 2017] A. Bader, O. Kopp, and M. Falkenthal. Survey and comparison of open source time series databases. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband* (2017). (Cited on page 13)

[Barbara and Chen 2009] D. Barbara and P. Chen. "Fractal mining-self similarity-based clustering and its applications". In: *Data Mining and Knowledge Discovery Handbook*. Springer, 2009, pages 573–589. (Cited on page 58)

[Bungartz et al. 2018] H.-J. Bungartz, D. Kranzlmüller, V. Weinberg, J. Weismüller, and V. Wohlgemuth. *Advances and New Trends in Environmental Informatics*. Springer, 2018. (Cited on page 29)

[Cafarella et al. 2009] M. J. Cafarella, A. Halevy, and N. Khoussainova. Data integration for the relational web. *Proceedings of the VLDB Endowment* 2.1 (2009), pages 1090–1101. (Cited on page 7)

[Champman 2019] C. Champman. A Complete Overview of the Best Data Visualization Tools (2019). Accessed: 2020-07-08. URL: https://www.toptal.com/designers/data-visualization/data-visualization-tools#. (Cited on page 16)

[Chen and Zhang 2014] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on Big Data. In: *Information sciences*. Volume 275. Elsevier, 2014, pages 314–347. (Cited on pages 2, 16)

[Chen et al. 2014] M. Chen, S. Mao, and Y. Liu. Big data: A survey. *Mobile networks and applications* 19.2 (2014), pages 171–209. (Cited on pages 1, 5, 6, 8)

[Cheng et al. 2014] X. Q. Cheng, X. L. Jin, Y. Wang, J. Guo, T. Zhang, and G. Li. Survey on big data system and analytic technology. *Journal of software* 25.9 (2014), pages 1889–1908. (Cited on page 5)

[Concepta 2020] Concepta. The Importance of Scalability In Software Design (2020). Accessed: 2020-09-07. URL: https://www.conceptatech.com/blog/importance-of-scalability-in-software-design. (Cited on page 46)

[Dean and Ghemawat 2008] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 51.1 (2008), pages 107–113. (Cited on page 59)

[Deutscher Wetterdienst 2015] Deutscher Wetterdienst. DWD OpenData (2015). Accessed: 2020-05-26. URL: ftp://opendata.dwd.de/. (Cited on page 2)

[DEWESoft 2020] DEWESoft. What Is Data Acquisition - DAQ or DAS? (2020). Accessed: 2020-05-28. URL: https://dewesoft.com/daq/what-is-data-acquisition. (Cited on page 6)

[Divya and Goyal 2013] M. S. Divya and S. K. Goyal. ElasticSearch: An advanced and quick search technique to handle voluminous data. *Compusoft* 2.6 (2013), page 171. (Cited on page 25)

[Emery 2011] D. Emery. Standards, APIs, Interfaces and Bindings. *Retrieved on: May* 5 (2011). (Cited on page 17)

[Fielding and Taylor 2000] R. T. Fielding and R. N. Taylor. *Architectural styles and the design of network-based software architectures*. Volume 7. University of California, Irvine, 2000. (Cited on page 18)

[Fu 2011] T.-c. Fu. A review on time series data mining. *Engineering Applications of Artificial Intelligence* 24.1 (2011), pages 164–181. (Cited on page 12)

[Garlan and Shaw 1993] D. Garlan and M. Shaw. "An introduction to software architecture". In: *Advances in software engineering and knowledge engineering*. World Scientific, 1993, pages 1–39. (Cited on page 21)

[Gartner Incs. 2001] Gartner Incs. IT Glossary: Big Data. *https://www.gartner.com/it-glossary/big-data* (2001). Accessed: 2020-04-28. (Cited on page 19)

[Ghotkar and Rokde 2016] M. Ghotkar and P. Rokde. Big data: How it is generated and its importance. *IOSR Journal of Computer Engineering* (2016). (Cited on page 5)

[GrafanaLabs 2020] GrafanaLabs. The analytics platform for all your metrics (2020). Accessed: 2020-05-10. URL: https://grafana.com/. (Cited on pages 4, 16)

[Han et al. 2011] J. Han, E. Haihong, G. Le, and J. Du. Survey on NoSQL database. In: *2011 6th international conference on pervasive computing and applications*. IEEE. 2011, pages 363–366. (Cited on page 8)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pages 243–246. (Cited on page 9)

[Hasselbring et al. 2019] W. Hasselbring, S. Henning, B. Latte, A. Möbius, T. Richter, S. Schalk, and M. Wojcieszak. Industrial DevOps. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE. 2019, pages 123–126. (Cited on pages 1, 9)

[Henning and Hasselbring 2020a] S. Henning and W. Hasselbring. Scalable and Reliable Multi-dimensional Sensor Data Aggregation in Data Streaming Architectures. *Data-Enabled Discovery and Applications* 4.1 (2020), pages 1–12. (Cited on page 42)

[Henning and Hasselbring 2020b] S. Henning and W. Hasselbring. Theodolite: Scalability Benchmarking of Distributed Stream Processing Engines. *arXiv preprint arXiv:2009.00304* (2020). (Cited on pages 4, 40, 42, 45, 61)

[Honghui Lu et al. 1995] Honghui Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In: *Supercomputing '95:Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. 1995, pages 37–37. (Cited on page 3)

[Kaspar et al. 2013] F. Kaspar, G. Müller-Westermeier, E. Penda, H. Mächel, K. Zimmermann, A. Kaiser-Weiss, and T. Deutschländer. Monitoring of climate change in Germany–data, products and services of Germany's National Climate Data Centre. *Adv. Sci. Res* 10.99 (2013), page 106. (Cited on page 24)

[Kozierok 2005] C. M. Kozierok. The TCP/IP Guide v3. 0. *Retrieved From, P10 http://www. tcpipguide. com/free/t_FTPOverviewHistorya ndStandards. htm* (2005). (Cited on page 19)

[Lamsal et al. 2013] L. Lamsal, R. Martin, D. Parrish, and N. Krotkov. Scaling relationship for NO2 pollution and urban population size: a satellite perspective. *Environmental science & technology* 47.14 (2013), pages 7855–7861. (Cited on page 36)

[Last et al. 2001] M. Last, Y. Klein, and A. Kandel. Knowledge discovery in time series databases. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 31.1 (2001), pages 160–169. (Cited on page 13)

[Lenzerini 2002] M. Lenzerini. Data integration: A theoretical perspective. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pages 233–246. (Cited on page 7)

[Luftdaten Info 2015] Luftdaten Info. Stuttgart OL (2015). Accessed: 2020-05-26. URL: https://archive.luftdaten.info/. (Cited on pages 2, 57)

[Makowsky 1986] J. Makowsky. Jeffrey D. Ullman. Principles of database systems. Computer software engineering series. Computer Science Press, Rockville, Md., 1982, vii+ 484 pp.-David Maier. The theory of relational databases. Computer Science Press, Rockville, Md., 1983, xv+ 637 pp.-Ashok K. Chandra and David Harel. Computable queries for relational data bases. Journal of computer and system sciences, vol. 21 (1980), pp. 156–178. *The Journal of Symbolic Logic* 51.4 (1986), pages 1079–1084. (Cited on page 13)

[Maletic and Marcus 2000] J. I. Maletic and A. Marcus. Data Cleansing: Beyond Integrity Analysis. In: *Iq*. Citeseer. 2000, pages 200–209. (Cited on page 7)

[Meyer et al. 2019] H. J. Meyer, H. Grunert, T. Waizenegger, L. Woltmann, C. Hartmann, W. Lehner, M. Esmailoghli, S. Redyuk, R. Martinez, Z. Abedjan, et al. Particulate Matter Matters—The Data Science Challenge@ BTW 2019. *Datenbank-Spektrum* 19.3 (2019), pages 165–182. (Cited on pages 2, 29, 57)

[Morrison 1994] J. P. Morrison. Flow-based programming. In: *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*. 1994, pages 25–29. (Cited on page 19)

[Morrison 2010] J. P. Morrison. *Flow-Based Programming: A new approach to application development*. CreateSpace, 2010. (Cited on page 1)

[Naqvi et al. 2017] S. N. Z. Naqvi, S. Yfantidou, and E. Zimányi. Time series databases and influxdb. *Studienarbeit, Université Libre de Bruxelles* (2017). (Cited on page 12)

[Netzer and Miller 1992] R. H. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.1 (1992), pages 74–88. (Cited on page 56)

[Nunes et al. 2013] S. A. Nunes, L. A. Romani, A. M. Avila, P. P. Coltri, C. Traina Jr, R. L. Cordeiro, E. P. de Sousa, and A. J. Traina. Analysis of large scale climate data: how well climate change models and data from real sensor networks agree? In: *Proceedings of the 22nd International Conference on World Wide Web*. 2013, pages 517–526. (Cited on page 58)

[Oliveira et al. 2012] S. F. Oliveira, K. Fürlinger, and D. Kranzlmüller. Trends in computation, communication and storage and the consequences for data-intensive science. In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE. 2012, pages 572–579. (Cited on page 7)

[Olshannikova et al. 2015] E. Olshannikova, A. Ometov, Y. Koucheryavy, and T. Olsson. Visualizing Big Data with augmented and virtual reality: challenges and research agenda. *Journal of Big Data* 2.1 (2015), page 22. (Cited on page 16)

[Pai et al. 2017] T. Pai et al. Big Data New Challenges, Tools and Techniques. *International Journal of Engineering Research and Modern Education (IJERME), ISSN (Online)* (2017), pages 2455–4200. (Cited on page 2)

[Postel and Reynolds 1985] J. Postel and J. Reynolds. File transfer protocol. *Request for Comments (RFC)* 959 (1985). (Cited on page 19)

[Rodriguez, Alex 2008] Rodriguez, Alex. Restful web services: The basics. *IBM developerWorks* 33 (2008), page 18. (Cited on page 17)

[Spengler, Reinhard 2002] Spengler, Reinhard. The new quality control and monitoring system of the Deutscher Wetterdienst. In: *Proceedings of the WMO Technical Conference on Meteorological and Environmental Instruments and Methods of Observation, Bratislava*. 2002. (Cited on page 24)

[Stephens, Robert 1997] Stephens, Robert. A survey of stream processing. *Acta Informatica* 34.7 (1997), pages 491–541. (Cited on page 1)

[Taibi et al. 2018] D. Taibi, V. Lenarduzzi, and C. Pahl. Architectural patterns for microservices: a systematic mapping study. In: SCITEPRESS, 2018. (Cited on page 4)

[Umweltbundesamt 2020a] Umweltbundesamt. The German Environment Federal Office (2020). Accessed: 2020-07-23. URL: https://www.umweltbundesamt.de/das-uba/was-wir-tun. (Cited on page 36)

[Umweltbundesamt 2020b] Umweltbundesamt. Weather API (2020). Accessed: 2020-05-08. URL: https://www.umweltbundesamt.de/. (Cited on page 2)

[Wobe-system 2016] Wobe-system. UJO Data Object Anotation (2016). Accessed: 2020-06-08. URL: https://www.libujo.org/info/. (Cited on page 20)

[World Health Organization and others 2006] World Health Organization and others. *WHO Air quality guidelines for particulate matter, ozone, nitrogen dioxide and sulfur dioxide: global update 2005: summary of risk assessment*. Technical report. World Health Organization, 2006. (Cited on page 29)

[Zhao 2002] Y. Zhao. client/server two-way communication system framework under HTTP protocol (Aug. 2002). US Patent App. 09/776,478. (Cited on page 17)