Prototype of a Scalable Monitoring Infrastructure for Industrial DevOps

Master's Thesis

Sören Henning

August 1, 2018

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring

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, 1. August 2018

Abstract

The digitalization of the conventional, manufacturing industry poses great opportunities but also challenges. As industrial devices, machines, and plants become more intelligent and autonomous, an increasing challenge for business enterprises is to connect and integrate them. This leads to the fact that software plays an increasing role in the daily production processes. In particular, small and medium-sized enterprises have to take high risks caused by high investment costs and a lack of experience in the development of distributed software systems. Agile and iterative methods, such as those that have long been common in other fields of IT, can be a solution to that. Industrial DevOps is an approach to apply those methods and bridge the gap between the development and the operation of software in industrial production environments. To facilitate this, development and operation are considered as a coherent, cyclic, and continuous process. Therefore, Industrial DevOps requires a continuously monitoring of all different aspects within a production.

In this thesis, we present an approach for a monitoring infrastructure for industrial production environments such as factories. It facilitates the integration of different types of sensors in order to make their measurements comparable. Automatically and in real time, recorded data is analyzed and visualized in a number of ways. Due to its microservice-based architecture, our monitoring infrastructure is designed to scale with the production size and to adapt to frequently changing requirements. This is supported by applying open source big data tools that have demonstrated their capabilities in similar challenges of Internet-scale systems. Moreover, to face the challenges of scalability and real-time data processing, our approach unites the principles of cloud and edge computing.

We implemented a prototype that covers all aspects of the designed architecture. This prototype is deployed in a medium-sized enterprise, where we monitored the electrical power consumption of a server, to show the feasibility of our approach. Furthermore, our approach shows scalability characteristics even though it is not always achieved reliably. Therefore, we present possible solutions to increase the reliability.

Contents

Chapter 1

Introduction

1.1 Motivation

The digitalization of the conventional, manufacturing industry enables a new level of automation in production processes. More and more technical machines and plants become increasingly intelligent and autonomous and, furthermore, they are becoming able to communicate with each other. To benefit from these new opportunities, the integration of systems and processes becomes increasingly important. That is, in particular, work flows have to be defined and implemented in software. However, especially for small and medium-sized enterprises this poses significant challenges as those often do not have appropriate software development departments. External solutions, which have to be expensively purchased, do not guarantee success and, in the worst case, ruin the business. Also establishing an own department or delegating the software development to specialized external companies is risky, as they do not know the internal processes. Defining requirements as well as understanding and implementing them is difficult and often fails.

Agile and iterative methods, such as those that have long been common in other fields of IT, can be a solution to that. DevOps [Lwakatare et al. 2015] is a movement aiming to bridge the gap between the development and the operation of software. It originates in the domain of web services (e.g., e-commerce), where the operation of software is traditionally separated from its development. This leads to several issues due to a lack of communication, collaboration, and integration. An important step for establishing DevOps is to reduce release cycle time. Using techniques such as Continuous Deployment makes it possible to react flexibly on new requirements [Claps et al. 2015].

Industrial DevOps [Titan Project 2018] is an approach to transfer the DevOps method into production environments such as factories. The goal is the same as for web applications: bringing the development of software closer to its operation. Therefore, development and operation should be considered as a coherent, cyclic, and continuous process. During its operation, the system is monitored and analyzed. Based on analysis results, new requirements are identified and, afterwards, implemented in steps as small as possible. After being automatically tested, the new software replaces the old one in the production, where it is monitored again.

An example of what Industrial DevOps should enable is to test alternative techniques and methods in the production. If, for instance, a revised process should be tested, it is often necessary to adjust software accordingly. By means of short release cycles, it is possible to simply bring the new software into production. Then, one can evaluate whether this new process is better than the old one. If that is not the case, one can flexibly revert to the previous one.

The *Titan* research project develops a software platform to realize Industrial DevOps. It aims especially for enabling the dynamical reconfiguration of processes. The goal is that IT-affine skilled workers are empowered to define processes by themselves without necessarily needing a designated software developer. Applying the idea of flow-based programming [Morrison 2010], they should be able to model work flows and specify how components exchange data with each other.

In order to determine whether such reconfigurations are effective, a monitoring of all involved fields is necessary. The nature of continuously generated data is manifold and covers aspects such as resource consumption, utilization of systems, or business information. In the context of the Titan project, the *Titan Control Center* will be developed to provide the necessary infrastructure for monitoring and analysis.

One application scenario for the Titan project is that associated enterprises want to reduce and optimize their power consumption. In particular, load peaks should be removed. Facilitated by the possibility to simply apply reconfigurations and monitor the entire production environment, saving potential should be discovered.

In this work, we present a prototype for the Titan Control Center. This prototype covers all aspects of the desired system, including the monitoring, the analysis, and the visualization of data. However, for each aspect it only comprises a subfield of the envisaged system. We limit this approach to measuring, analyzing, and visualizing the electrical power consumption in a production environment.

1.2 Goals

The superordinate goal of our work is to create a prototype of an infrastructure to monitor real production environments. A key requirement for this infrastructure is to be scalable, which means there is no bound in the amount of processable monitoring data by using enough hardware resources. To achieve this, we are going to develop a scalable architecture, implement a prototype, and evaluate the resulting system as described in the following.

1.2.1 Design of a Scalable Architecture

We design a software system that receives measurement data on power consumption from production environments, analyzes them, and visualizes the analysis results. As a first goal, we conceptualize an architecture for such a system. Already the architecture should be focused on scalability in order to realize a scalable implementation. Even though we only analyze power consumption in this prototype, the desired architecture should be designed in a way that allows other types of analyses without large adjustments. Moreover, the architecture should be flexible and adaptable to changes. Since we develop a prototype, it is likely that we have to replace used technologies and implementations. This applies during the development of this prototype as well as for the shift from a prototype to a later version.

1.2.2 Implementation of a Prototype

In addition to an architectural concept, we also realize an implementation of a prototype. Therefore, we have to implement the corresponding components defined by our concept. These implementations require but also enable a continuous revision of the designed architecture. In particular, we want to examine whether selected technologies are sensible.

The visualization should be realized as a web-based application that runs in a web browser. This provides a simple approach to support several different devices, does not require the installation and update of a local installed software, and, moreover, enables support for latest visualization technologies.

1.2.3 Evaluation of Feasibility and Scalability

Since we develop a prototype for a system that is supposed to run later in real production environments, an extensive evaluation is indispensable. Thus, we will evaluate the feasibility of the developed approach as well as its scalability.

In the feasibility evaluation, we will evaluate whether our implementation works as desired and meets the defined requirements. Moreover, we want to assess our architecture decisions and selections of technologies.

We evaluate the scalability to examine whether our approach is also applicable for large production environments. Therefore, we want to investigate if and where bottlenecks occur, which slow down the whole system. Furthermore, we expect to get an impression of which hardware requirements arise under which load.

1.3 Document Structure

The remainder of this thesis is structured as follows: Chapter 2 outlines foundations of how scalable software systems can be designed and operated in order to process high volume of data. We present common technologies that support this in Chapter 3. Part I presents the realized monitoring infrastructure we developed in this work. For this purpose, we present the general approach in Chapter 4 and the design for an architecture in Chapter 5. Based on this architecture, we implemented a prototype, which is presented in Chapter 6. In Part II, we evaluate our approach (Chapter 7) and compare it to related work (Chapter 10). Part III summarizes this thesis in Chapter 9 and, in Chapter 10, it points out what aspects future work may address.

Chapter 2

Foundations

This chapter describes the foundations that are necessary to design and develop a scalable industrial monitoring infrastructure. In Section 2.1, we explain what scalability is in the context of software systems and how it can be achieved. Moreover, we differentiate scalability from elasticity. Afterwards, we introduce the microservice architectural pattern in Section 2.2, which particularly focuses on making a system scalable. In Section 2.3, we define the term *big data* and present architectural patterns to handle huge amounts of data. Section 2.4 introduces the concepts of cloud and edge computing and outlines their advantages and disadvantages.

2.1 Scalable Software Systems

Referring to the definition of Smith and Williams [2002], a software system is scalable if it is able to meet its response time or throughput objectives independently of the demand for its functions. That means, for instance, a web server is capable to deliver a website even though more and more users try to request it. In the context of this work, scalability refers to the fact that the monitoring infrastructure should work independently of the production environment size it is applied in. To achieve scalability, there exist two typical possibilities, which we present in the following.

2.1.1 Vertical Scaling

Vertical scaling is referred to adding more resources to a running single computer. Such resources can be, for example, a more powerful CPU or additional main memory. The advantage of this variant is that it is clearly simpler to implement. Programming languages and frameworks abstract well from the actual machine software is running on. Therefore, it usually makes no difference on which hardware the application runs in production. A software that runs with a certain performance on certain hardware is likely to run with a better performance on more powerful hardware.

However, vertical scaling has some serious shortcomings. Firstly, more powerful hardware is increasingly expensive. Secondly, vertical scaling is only possible to a certain degree. At some point, a better performance cannot be achieved on a single machine. In the context of CPU, another problem arises. Nowadays, more powerful CPUs do not use significantly higher clock rates but instead consist of multiple cores. Therefore, parallelism has to be considered in the implementation and, thus, using more powerful hardware is often not that simple.

2.1.2 Horizontal Scaling

The other way how software can be scaled is by using additional machines and distributing the load equally between them. That is called horizontal scaling and provides solutions to the issues with vertical scaling. Horizontal scaling is cheaper since a larger amount of average-powered hardware is cheaper than only a few high-performance machines. Thus, it also allows for a more fine-grained scaling. The most relevant advantage of horizontal scaling is, however, that it facilitates an arbitrary scaling.

A software system that scales horizontally is necessarily a distributed one. This has another benefit, namely that the system becomes less susceptible to partial failures. If a single machine fails, its work can be handled by others one or only a single part of the system becomes unavailable instead of the entire one. Moreover, its distributed nature allows a system to deploy part of it redundantly. Using this, loss of data on one machine can be compensated by another machine additionally storing those data.

However, distributed systems are significantly harder to program. As with concurrent programming on the same machine, it is difficult or often impossible to determine exactly how the execution of a system behaves. Thus, requirements such as mutual exclusion must be implemented and problems such as deadlocks must be prevented. Since the individual nodes in a distributed system communicate via a network, however, programming gets even more complicated. Therefore, the characteristics of networks have to be considered. According to Tanenbaum and Steen [2006], these are, for instance, that they are not reliable or secure, their topology may change, or latencies and bandwidth effect the software execution. Furthermore, those network characteristics imply that it is never possible for one machine to know the current state of the others. To deal with these challenges, several patterns and technologies have been developed over the last decades. A recent pattern are mircoservice architectures, which we present in Section 2.2. Some technologies are presented in Chapter 3.

Horizontal scalability can be achieved in different ways, which can also be combined. The *Scale Cube* (Figure 2.1) is a model defined by Abbott and Fisher [2009] to describe the scalability of a system. In this model, each software system can be located within a cube (or three-dimensional coordinate system). The position on each of its three axes corresponds to the degree of scalability in a certain aspect. Scaling on the x-axis means to run multiple identical copies of a system. Scaling on the y-axis is splitting the application into several components by function. Scaling on the z-axis also means to run multiple copies but each copy is responsible for only a subset of the data (called *sharding*). These three aspects can also be combined with each other. A software that is located in the origin point of the cube is a monolithic one consisting of only a single instance. An entirely scaled software (located far away from the origin point) is divided into several individual components, with several

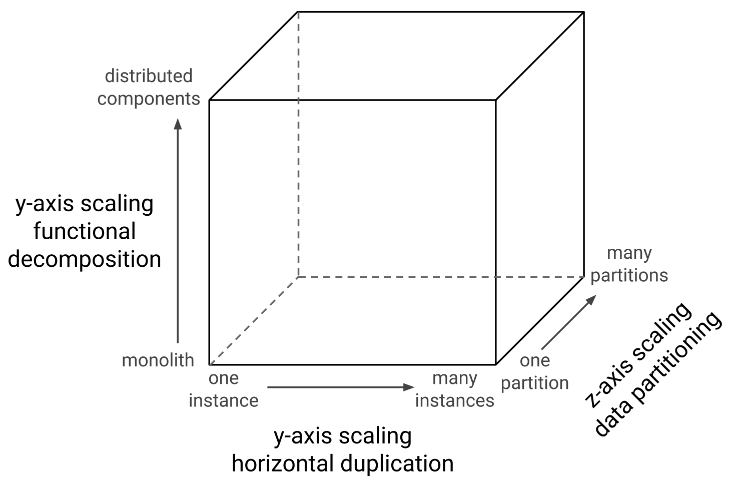

distributed components
y-axis scaling functional decomposition
monolith
one instance
many instances
y-axis scaling horizontal duplication
one partition
many partitions
z-axis scaling data partitioning

Figure 2.1. The Scale Cube (figure based on Richardson [2018]).

instances of each component running, which are responsible for different data as well as being simple copies of each other.

A step beyond scalability is elasticity. It is the ability to scale a system automatically based on the workload. That means for each point in time the required resources correspond to the available one [Herbst et al. 2013]. Therefore, elasticity can only be achieved in a horizontally scalable system. In the approach of this work, we do not aim for achieving elasticity. However, a scalable approach already lays the foundation for elasticity.

2.2 The Microservice Architectural Pattern

Microservice architectures [Newman 2015; Hasselbring and Steinacker 2017; Hasselbring 2018] are an approach to modularize software. As the name suggests, a software is divided into components, called microservices, that can be used largely independent of each other.

The separation into microservices is based on business capabilities. Each service maps to an own business area and provides a complete implementation of it. Microservices are developed by one cross-functional team with a good understanding of the respective business domain. For this domain, the corresponding team is responsible for recording the requirements, developing the service, and deploying it to production. The division into business areas should be so fine-grained that the teams do not become too large and the microservice can be reimplemented within a bearable time.

An important requirement for microservices is that they are isolated from each other. They run in separate processes and do share any state. Thus, they can independently be started, stopped, or replaced. In particular, microservices can be independently released to production so that a new version of one microservice does not require to update the others. If a failure occurs in one microservice, it should not effect other services. Therefore, they are deployed in individual environments. Those can be separate (virtual) servers or containers (see Section 3.6). This implies that microservices communicate exclusively via the network.

Microservices do not only share no state, but also no implementation. They communicate via clearly defined interfaces that hide their implementation. In contrast to several other approaches for modularization, microservices have also separate databases and do not share a common one. The complete separation of implementations facilitates an individual choice of programming language, database system and technology-stack for each service. That simplifies the exchange of those so that, for instance, a microservice can be migrated step by step to new versions of dependencies as well as to other technologies. In addition, that allows to use in each microservice the most suitable technology for its tasks. The loose coupling of microservices requires to define interfaces in a programming language independent way (e.g., REST [Fielding 2000]). However, even if services use similar technologies, they should not share each other's program code. If program code is required by multiple services, it should be shared in the form of an open source library.

For the development of prototypes for larger software systems, as presented in this work, using microservices offers some advantages. Due to the strong decoupling of components, it is easier to add further ones or remove components that are not longer used. As technologies can be selected individually per microservice, it is significantly simpler to evaluate and replace them.

2.2.1 Scalability

Unlike other approaches to modularize and decouple software components, microservices especially enable horizontal scalability [Hasselbring 2016]. Referring to the Scale Cube introduced in Section 2.1, microservices facilitate a scaling in the y-axis, i.e., scaling by functional decomposition. However, also a scaling in the other dimensions can be accomplished by using microservices.

After dividing a software into microservices, multiple instances of each of them can be deployed. These instances can be simple copies of each other (x-axis scaling) as well as separations that are only responsible for a data subset (z-axis scaling). In particular, this allows for a different number of instances per service so that services can be scaled independently of each other.

A microservice-based system can generally be considered as a distributed system. If it is scaled to a certain degree, it is also actually one. This makes it prone to errors, as described in Section 2.1.2. To ensure quality, errors must therefore be isolated in the services they occur in. That means, an error that arises in one microservice must not influence other services. For this reason, microservice architectures are structured as decentralized as possible. If a service makes a request to another, a possible failure of this request always has to be taken into consideration. This has to be respected in the implementation, for example, by automatically repeating failed requests after a certain period of time. This is particularly important when starting the entire system. If all services are started at once, it is impossible to determine which service is running at first in a distributed system. Therefore, it is likely that a particular microservice is available earlier than a service it depends on. A further option to overcome these issues is to use asynchronous messaging between services. Using an appropriate messaging bus, they can send messages without dealing with its delivery. We present such a messaging system in detail in Section 3.2.

If microservices depend on each other, it may be reasonable to cache required data. Thus, apart from performance reasons, a microservice may still be able to perform all or parts of its tasks even if the other service becomes unavailable. This implies that not all services always have consistent data. In many use cases, however, this is bearable.

Not only different microservices but also multiple instances of it need to be coordinated. Often this is done by a common database, which, however, can also comprise several instances. Also here, it may make sense to give up strong consistency in favor of scalability. Therefore, databases for microservices often only provide *eventual consistency*. A microservice becomes even more scalable if it is stateless. In that case, no coordinate at all is necessary and the load can be distributed to arbitrary many instances.

2.2.2 User Interfaces

Since the microservice pattern originates from web-based applications, a user interface (UI) for them is typically realized as a web application. Besides the part visible to the user in the web browser (*frontend*), most web applications also have a part running on the server (*backend*) to handle the user interactions. There exist two major approaches on how to realize UIs in microservice architectures [Newman 2015].

The first one is that all microservices provide their own UI. Often they do this in terms of individual components that can be composed in the latest possible step (e.g., when rendering the page in the web browser). Known as the *UI Fragment Composition* pattern, it offers the advantage that the separation of business functions can also be realized in the UI. Thus, frontend and backend can be developed by the same team and for new UI functionalities, no cross-team communication is required. However, this pattern also has its drawbacks. A user interface is not necessarily restricted to a web page. There can also be mobile apps or native clients that would require an own UI. UI components for such platforms are a lot harder to integrate. Another issue with this approach is that a good UI is consistent in its appearance and behavior. That is clearly harder to achieve if UI components come from different sources. Moreover, it is not always possible to directly map microservices to components. UI components that require data from different services are difficult to implement using this pattern.

To overcome these disadvantages, an alternative way to build UIs is as one single application that only uses the APIs of the other microservices. This approach is often combined with the *API Gateway* pattern to provide one single, central API and to abstract and hide the internal decomposition into microservices. Here, however, an issue is that the API Gateway can become rather large, especially if the required interfaces differ a lot for different frontends. To prevent this, a modification of this approach is the *Backends for Frontends* pattern. It defines a separate API Gateway (called backend) for each frontend. This way, each gateway can only provide those interfaces, which its frontend needs.

2.3 Big Data Processing

The term *big data* refers to data that cannot be captured, stored, processed, and perceived by traditional methods and technologies within a tolerable time [Chen et al. 2014]. They are of huge size, generated with high frequency, and of various types and sources. Furthermore, that data are considered to be of high value. Whereas big data in general covers divers types of data, we focus in the following on data that is acquired in a continuous temporal context.

Big data can be viewed from different perspectives and at different phases. To structure those phases, Chen and Zhang [2014] adopt the typical knowledge discovery process from the field of data mining. This process consists of 5 steps as shown Figure 2.2. Recording is the fundamental step of all forms of further data processing. In the context of industrial environments, this is the boundary to monitoring hardware. The next step is the filtering and cleaning of data as often the amount of recorded data is larger than those that need to be analyzed. This step covers also the integration and unifying of different data types to set them in relation to each other later. Afterwards, the data is analyzed in order to extract information out of it. Those analyses are more less complex algorithms or processing techniques. They generate new data from the input that either simplifies it or enriches it by setting it in relation with other data. In the next step, data is visualized to make it perceivable. This allows to interpret it so that in the last step based on that decisions can be made.

Different challenges arise in the individual phases. Solutions need to be found if the amount of data is so large that processing and storing is not possible anymore or too expensive. Many algorithms and techniques do not work in context of big data as their execution takes too long or they require too much memory. Also visualizing huge data sets poses difficulties as it may not be perceivable.

There are two typical approaches for processing big data. Both have advantages and disadvantages and it depends on the use case and application scenario which one is more appropriate.

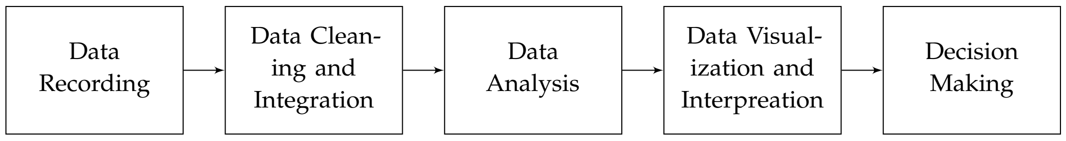

Data Recording
Data Cleaning and Integration
Data Analysis
Data Visualization and Interpreation
Decision Making

Figure 2.2. The typical knowledge discovery process [Chen and Zhang 2014].

The first possibility is to process data as real-time stream (also called *online*). Following this approach, data is analyzed immediately after recording. Hence, real-time analyses allow to quickly react to generated information. The issue with this approach is that it is often more difficult to carry out. Algorithms are often less efficient when working on real-time data or are harder to implement. Moreover, this approach makes high requirements on software and hardware as the data processing always has to be as fast as the recording.

In contrast, there are batch analyses (also called *offline*). Such analyses are executed if all or a large set of data is collected. In this approach, analyses are often more efficient but also performance is not that important. Another advantage is that batch analyses can be carried out when the hardware is idle anyway (e.g., at night). That can be a significantly cost saving. However, when it comes to the speed at which data is processed and results are available, batch analyses is not an option.

One solution to combine real-time stream and batch approaches are *Lambda architectures* [Marz and Warren 2015]. Architecture built upon that pattern encompass speed, batch, and serving layers. The batch layer receives recorded data and analyzes them. This could often take a long time for complex analyses. Therefore, the recorded data is additionally passed to the speed layer. It executes immediately the same or only part of the analyses. However, it focuses on a fast processing and inaccurate or incomplete results are not a problem. The serving layer merges the results of batch and speed layer, stores them, and provides access to it for the user.

The major drawback of Lambda architectures is that logic has to be implemented two times since different technologies are appropriate for batch and speed layer. To overcome this, *Kappa architectures* [Kreps 2014] are a simplification of Lambda architectures. They remove the batch layer and have solely a speed and a serving layer. Following this pattern, all data is considered to be an immutable stream of data. Thus, the speed layer is implemented in a way that supports the necessary queuing of input data and is able to repeat analyses on already processed data if additional data is received.

2.4 Cloud and Edge Computing

As in many other organizational structures, the architecture of computer systems is subject to the forces of centralization and the force of decentralization [Garcia Lopez et al. 2015]. In the early days of the Internet, it was designed as a decentralized network to exchange information between equal nodes. For the increasing role of the Internet as a platform for applications and services, however, its decentralized nature had several drawbacks.

Over the last decade, a trend towards *cloud computing* [Armbrust et al. 2010] has developed. Cloud computing describes a technique, where data storage and computations are performed in large, centralized data centers. The location of those as well as their underlying hardware is concealed in the “cloud”. Cloud computing providers offer various service models of different degrees of abstraction [Youseff et al. 2008]. Infrastructure as a Service (IaaS) abstracts the hardware, software will be deployed on. A user can specify the necessary infrastructure (e.g., computing power or memory) and, often by using virtualization techniques, the cloud provider allocates it. In addition to this, Platform as a Service (PaaS) also abstracts the necessary software environment, e.g., a programming language interpreter. Software as a Service (SaaS) describes the delivery of software via the Internet so that users do not have to install it locally.

A major benefit of cloud computing is the avoidance of hardware resource wastage. If an enterprise operates its own servers, it always has to keep as much capacity available as it is required at its maximum load. Therefore, it may also have to maintain hardware resources when they are not needed. Cloud computing, on the other hand, allows to always rent only the resources that are needed. This can be a significant cost benefit. Especially software with unpredictable, periodical, or expanding load can be operated cheaper.

Moreover, cloud computing can be used if setting up an own data center is too expensive or the necessary knowledge for it does not exist in a company. For instance, that is the case for many businesses that are addressed by Industrial DevOps. Another advantage in this industrial context is that software running in production plants is often not able to perform complex analyses. Cloud computing can be a solution to that.

Apart from services of external providers, called *public clouds*, such infrastructures can also be realized by the companies themselves in the form of *private clouds*. Mixed forms exist with *hybrid clouds*, where a private cloud is supplemented on high load with an external one, and *community clouds*, where partners that trust each other share a common infrastructure. The more public a cloud is, the less control it allows over the data. That is particularly important in the industrial context as many companies do not want their data to be processed on servers of external companies. However, the less public a cloud is, the smaller and, therefore, less scalable it is.

A major drawback of cloud computing in industrial contexts is that it can be too slow for many applications. If, for example, two sensors that are located next to each other communicate and data are first sent to the cloud before they are sent back to other sensor, this comes with high latencies and requires large bandwidths.

Edge computing, also referred to as *edge-centric* or *fog computing* [Bonomi et al. 2012; Garcia Lopez et al. 2015], is an alternative to cloud computing. Here, data are preferably processed where they are acquired: decentralized, at the “edges” of the networks. This way, the amount of transferred data can be reduced as well as the transmission distance, which saves costs. In production environments, edge computing is facilitated by increasing powerful devices, machines, and controllers. Thus, they are able to perform many computations and analyses by themselves. Moreover, this has also advantages in terms of privacy as data does not have to leave the internal network in the production.

A downside of this approach is that if computation is performed entirely decentralized, the overall system is not elastic anymore. However, using the edge and the cloud paradigm is not mutually exclusive. Instead, edge computing is usually indented as a supplement to cloud computing. In industrial production environments with high data volume, for instance, data can already be filtered and aggregated at the edges. Then, in a scalable cloud infrastructure, more precise and profound analyses can be executed. *Edge controllers* are physical devices performing the tasks of filtering and aggregating at the edges. Software components performing those tasks are called *edge components* hereinafter.

Chapter 3

Technologies

In this chapter, we introduce technologies that supports the development of a monitoring infrastructure for production environments. In Section 3.1, we introduce the Kieker monitoring framework. Apache Kafka (Section 3.2) and TeeTime (Section 3.3) are technologies for processing high data volume. A technology for its storage is the database Cassandra (Section 3.4). We present the web frontend framework Vue.js in Section 3.5, which supports the development of our visualization. Finally, for the deployment of our implementation, we present the containerization platform Docker in Section 3.6 and the corresponding orchestration software Kubernetes in Section 3.7.

3.1 The Software Monitoring Framework Kieker

Kieker [van Hoorn et al. 2009; 2012] is a framework that monitors and analyzes the runtime behavior of software applications and systems. It consists of two components: The monitoring component undertakes the task of gathering runtime information of an application, e.g., the execution time of specific methods. It focuses on a fast data capturing to affect the performance of the monitored application as little as possible. The analysis component reconstructs a system model of the software and assesses the collected data.

Kieker provides several ways to transmit monitoring records from the monitoring to the analysis component. This covers low-level techniques such as TCP or the file system, but also specific technologies such as Apache Kafka [Knoche 2017] or the database Cassandra [Moebius and Ulrich 2016]. In each case, they consist of a writer (used by the monitoring component) and reader (used by the analysis component).

Besides several build-in record types, Kieker provides the Instrumentation Record Language (IRL) [Jung and Wulf 2016] to define custom record types. Based on the IRL, code generators can construct data structures of those records for various programming languages. The generic nature of its reader-writer architecture is a great benefit of Kieker. It allows to process arbitrary (and therefore even custom) monitoring records.

Many requirements we have on our approach are already supported by Kieker in other contexts. We are able to use existing parts and adjust additional for our demands. Section 6.2 describes how we apply Kieker.

3.2 The Messaging System Apache Kafka

Apache Kafka [Kreps et al. 2011; Apache Software Foundation 2017] is a distributed platform to transmit and store messages with high throughput and low latency. Its architecture is designed to be highly scalable and fault-tolerant and, thus, Kafka is often used in event stream and big data processing.

Kafka can be deployed as a cluster of several nodes that are coordinated by *Apache ZooKeeper*[1]. Within a cluster of Kafka nodes, messages or records are grouped into categories called *topics*. Following the publish–subscribe pattern [Hohpe and Woolf 2003], *producers* write records into particular topics and *consumers* subscribe to particular topics to receive incoming records.

A topic can be separated into multiple partitions, which can be stored redundantly on multiple nodes to increase fault tolerance. Each partition is an ordered, immutable sequence of records and each record consists of a key, a value, and a timestamp. Producers can decide by themselves to which partition records are written. However, if using the default configurations, this decision is made automatically based on its key. This guarantees that records having the same key are always stored in the partition. A wise choice of the key allows therefore that records can be processed efficiently in parallel by multiple nodes or threads. For this purpose, each node or thread can subscribe to an individual partition and no coordination between them is necessary.

Kafka Streams is a stream processing framework built on top of Kafka that utilizes that. In a declarative way, one can specify a topology of processing steps and transformations. Kafka topics can serve as input or output of data. Based on this, Kafka Streams creates the necessary producers and consumers as well as immediate topics. Moreover, Kafka Streams is a framework to implement Kappa architectures.

3.3 The Pipe-and-Filter Framework TeeTime

Pipes-And-Filters [Shaw 1989] is an architectural pattern for software that processes streams of data. As its name suggests it consists of two type of components: pipes and filters. A filter represents a single processing step that receives and dispatches data. Filters can be connected by pipes that handle transfer of objects between filters. Using this, a network of filters connected by pipes can be created and data can be sent through this network.

TeeTime [Wulf et al. 2017; Henning 2016] is a Java framework for developing software systems that are based on the Pipes-and-Filters pattern. It contains the basic entities *stages*, *ports*, *pipes*, and *configurations*. Stages are equivalent to the filters in the pattern. The framework provides multiple abstract stage types that can be extended at will. At its execution, a stage reads an object from its *input ports*, processes it in a defined way, and sends it to its *output ports*. In addition, the framework provides a number of predefined stages. A pipe connects an output port of one stage with an input port of another stage. In

[1] `http://zookeeper.apache.org/`

a configuration, stages could be defined and their ports connected. The framework takes care of creating the right pipes between the ports.

3.4 The Column-Oriented Database Apache Cassandra

Apache Cassandra [Lakshman and Malik 2010] is one of the most common column-oriented databases. It is designed to run in a cluster of multiple nodes, where on each node one or more instances of Cassandra are running. Each node stores only a part of the total data. Moreover, to reduce the danger of data loss, data can be replicated to multiple nodes.

Similar to relational databases, Cassandra stores data in tables, consisting of rows and columns. Cassandra uses the concept of primary keys. A primary key is a set of multiple columns and can be divided into two parts: A partition key and clustering columns. The partition key is per default the first column of the primary key but can also include multiple columns. Using the partition key, Cassandra decides on which node data are stored. That means if a query is made to request data with a certain partition key, Cassandra knows on which node that data would be stored. Thus, it can decide that this node is responsible for handling the query. The second part of the primary key, the clustering columns, include the remaining columns. They determine in which order the rows should be stored. This way a simple narrowing of the search field can be achieved.

3.5 The Web Frontend Framework Vue.js

Vue.js [You 2018] (often referred to as Vue) is a JavaScript framework for building single-page applications (SPA). SPAs are applications that run in the user's web browser in the form of a single web page. User interaction is not realized by loading a new page but instead by directly processing it in the browser or by asynchronously communicating with a server. Vue provides a framework to structure such applications and offers tools to enhance their implementation.

Applications built with Vue consist of several, largely independent components, each representing an individual part of the user interface. Components are responsible for their own logic and appearance, i.e., layout and design. That facilitates encapsulation and separation of concerns. Components can be composed to other, larger components so that a Vue application can be considered as tree of components. Moreover, components are intended to be reusable, for example, if a single part of the user interface is displayed at multiple places in an application. For this purpose, they can also be configured if modifications of it should be reused.

Vue forces a loose coupling of components. Communication and data exchange is only allowed between a parent and a child component. It is prevented between siblings to reduce the risks of cyclic dependencies. To additionally eliminate cyclic dependencies between parent and child, also their communication is restricted. A child can only emit events to its parent, e.g., when a button is clicked. It is up to the parent how it handles this event. Hence, a child is not able to manipulate its parents' state. A parent can pass properties to a child at its creation. However, the child is not allowed to modify those if that would also change the parents state[2].

3.6 The Containerization Platform Docker

Containerization [Bernstein 2014] is a technique to virtualize the environment software it is executed in. *Containers* isolate and encapsulate a software with all its dependencies. In contrast to virtual machines, containers have a significantly lower overhead as they are able to use several parts of the operation system they are running in.

A common technology for containerization is the open-source platform Docker [Docker, Inc. 2017]. It allows to create templates for containers, called *images*. Those are text files that define everything that an application needs to run, for example, system tools, system libraries, or required resources. Thus, no development, testing, and maintenance for different systems is necessary. Single components or services of large software systems are often separated to individual images.

Containers can be created, started, and stopped as often as desired. Therefore, especially in the context of microservices, containers facilitate the reliability, scalability, and agility of software system [Hasselbring and Steinacker 2017].

3.7 The Container Orchestrator Kubernetes

Kubernetes [Cloud Native Computing Foundation 2017] is an open-source system for deploying and managing containerized applications on a cluster of physical hosts called *Nodes*. For this purpose, one or more containers can be grouped in so called *Pods*. Containers within a Pod share the same IP address and are always scheduled together on the same Node. Kubernetes automatically places Pods on Nodes considering required and available resources.

Furthermore, Kubernetes provides the concept of *Deployments*. These are objects describing the desired deployment of Pods. In particular, they allow to define the desired amount of Pods of a certain type. Kubernetes manages to create and run those Pods on Nodes such that Nodes are equally utilized and not all Pods are running on the same physical host. If more or less instances are desired to run, only the specified number in the Deployment needs to be adjusted. *StatefulSets* are similar to Deployment but intended to describe Pods that hold state such as databases.

Services serve for defining the interfaces of Pods. Only those Pods for which a Service is created are reachable from other Pods or from the outside of the cluster. In particular, they allow to restrict this individually. Using Services, Pods can discover each other.

[2]As this cannot be achieved on a language-level, Vue create warning messages if a component tries to do this.

Part I

A Scalable Monitoring Infrastructure for Industrial DevOps

Chapter 4

Approach

In this chapter, we give an overview of our developed approach for a monitoring infrastructure.

The application contexts of industrial DevOps are production environments such as factories. Such environments typically consist of a wide variety of devices and machines. Those devices and machines produce or process goods, consume resources and change their condition in the course of time. They influence each other and constitute altogether the overall state of the production environment. Therefore, in order to analyze the characteristics of the entire application context, it is necessary to know its individual parts as good as possible.

The design of our approach follows the general knowledge discovery process as described in Section 2.3. In Figure 4.1, we give an exemplary presentation of it. The individual steps are described in the following.

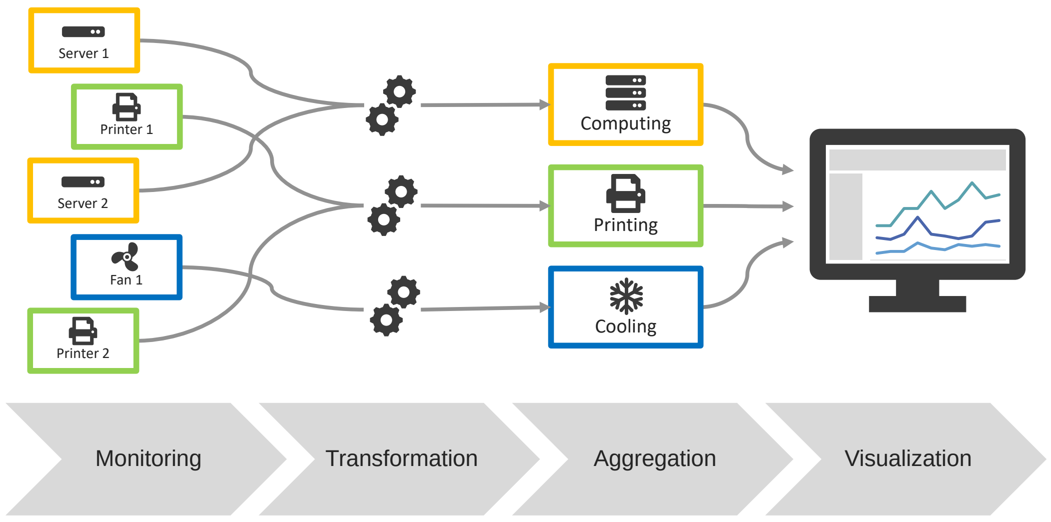


Figure 4.1. Our general approach for a monitoring infrastructure.

4.1 Data Recording

Referring to the knowledge discovery process, data recording is the fundamental step to gain knowledge. Caused by increasingly powerful hardware, more and more devices are equipped with sensors. Furthermore, following the trend towards the *Internet of Things*, they are increasingly equipped with network capabilities [Jeschke et al. 2017].

The data produced by such devices are manifold. They can comprise resources consumption data, data regrading the production speed or throughput of processed goods, but also the current state such as temperature. In the prototype present in this work, we limit ourselves to monitor electrical power consumption. This is a factor that companies particularly plan to optimize for (see Chapter 1). Power consumption can be measured in various ways. For our implementation, we decided to restrict our support to active power [International Electrotechnical Commission 2018].

Since not every device is able to monitor its consuming power itself, it is also possible to do this by using other devices. For instance, when considering electrical power, an appropriate power socket can be used that is able to measure the consumption. For this reason, we often use the term "sensors" in the following when referring to consumers.

4.2 Data Cleaning and Integration

Devices and machines in production environments mostly come from different manufactures located in different business domains. Furthermore, they are likely to differ in their ages and originate from different generations of technological evolution [Vogel-Heuser et al. 2014]. That leads to the fact that also the way they supply data varies widely. Most notably, this is due to the protocols and data formats they use but also to the way they measure. Parameters such as precision, sampling rate, or measurement units may vary from domain to domain.

In order to compare data of different sensors and to consider the data analysis from a higher level, data first have to be brought into a common format. This also includes converting measurement units or splitting up individual measurement that are send together. Moreover, it is possible that not all measurements are of interest and only specific values have to be selected. As the amount of data may be too large to be analyzed, it is often reasonable to aggregate measurements at first.

4.3 Data Analysis

The individual consumption values of devices are often too detailed to draw conclusions about the entire production environment. Instead, it may often be more reasonable to evaluate data for an entire group of devices. That is even more significant in cases, where

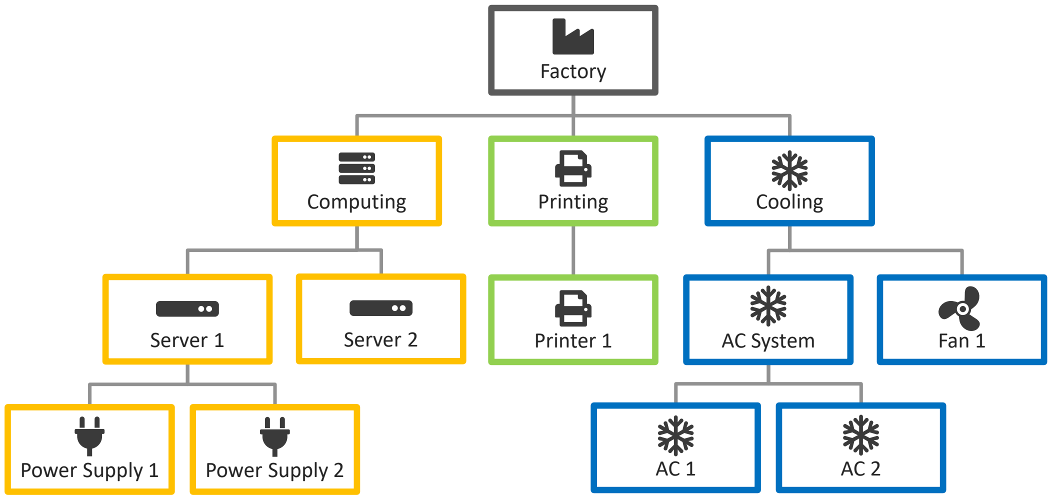


Figure 4.2. An example of a sensor registry. It has one top level node representing the set of all devices in a factory. Those devices are divided into three groups each containing one or more sensors. Moreover, groups can be nested to represented interrelated devices or device featuring multiple sensors.

devices have more than one power supply and those are monitored individually. It is likely that in such cases, only the summed data is of interest.

To aggregate arbitrary sensors in a generic manner, we developed an approach to hierarchically aggregate sensors. Therefore, we arrange all sensors in a tree data structure. The leafs of the tree correspond to real physical sensors that supply real monitored data. The inner nodes of the tree represent a group of sensors, which aggregate the sensors contained in it. Namely, this group corresponds to all sensors represented by the child nodes of a node. In the following, we call those groups *aggregated sensors*. Figure 4.2 gives an example of such a tree structure. We call it *sensor registry* in the following. A sensor registry has exactly one top level node, that is, a node without a parent.

Apart from the hierarchically aggregation, we perform further analyses to enable advanced visualizations. Therefore, the corresponding visualizations define which analyses these are.

4.4 Data Visualization to Enable Decision Making

Visualizing the monitored and analyzed data serves for enabling a user to draw conclusions about the current state of the overall production. Based on that, a user should be able to make decisions about the further operation of the production.

Together with the associated partners of the Titan project, we identified three key requirements that are of particular importance for meaningful visualizations. The first one is the ability to correlate the consumption of different devices. That is supposed to allow for analyzing the reasons of certain consumption values. Second, using simple symbols, similar to a traffic light, the user should be able to check at a glance whether the current state is normal or whether problems may have occurred. The third requirement is that the user should be able to look at consumption values in detail in order to analyze them closely.

Chapter 5

Architecture

In this chapter, we present an architecture design for a monitoring infrastructure of production environments as defined by the first goal of this thesis (see Section 1.2.1). Section 5.1 outlines the drivers that lead to our architectural decisions. The architecture itself can be seen from a software, implementation-oriented perspective as described in Section 5.2 and from an deployment perspective which is explained in Section 5.3.

5.1 Architectural Drivers

The decisions we have to make when designing an architecture are influenced by different factors. These are, firstly, requirements we set to ensure the final software works as desired and, secondly, constraints that need to be considered and may conflict with the requirements. In the following, we describe both requirements and constraints in detail.

5.1.1 Architectural Requirements

Horizontal Scalability Our approach should be designed for small-scale production environments as well as for arbitrary large ones. In particular, the amount of sensor data to be processed can vary strongly. That does not only differ from enterprise to enterprise but also within the same application scenario, for example, if after an initial test period additional company departments should be integrated.

If the amount of sensor data grows (per sensor as well as the total amount of sensors), more computing power is necessary. To a certain degree this can be achieved by providing accordingly more powerful hardware (vertical scaling). However, one quickly reaches a limit where additional power can only be accomplished by adding further machines.

Therefore, our architecture has to be designed in a way that facilitates an operation on multiple machines and, furthermore, utilizes them evenly. Additionally, increasing the amount of sensor data should also be possible during the ongoing operation to avoid downtimes in which the infrastructure would not longer have been monitored. Just like an increasing load, also a decreasing load should be able to handle.

Data Processing in Real-Time Concerning the processing of measurement data, we have soft real-time requirements on our approach. Following the definition of Martin [1965], that means that data need to be received, processed, and returned sufficiently fast so that the results of this processing can effect the environment. However, to differentiate this from hard real-time requirements, a delayed result does only effect the usefulness of that result but does not lead to catastrophic consequences [Shin and Ramanathan 1994].

The data transmission, analysis, and visualization in our approach should be performed as quickly as possible in order to allow for an insight into the current infrastructure's status at any time. This is the only way to react to unexpected events or to evaluate the current process scheduling. This requirement needs to be reflected in the architecture design so that, for example, using batch processing techniques is not an option for the majority of the analyses.

Adaptability The prototype we present in this work is not intended to be a finished product. Instead, it should serve as a basis for a continuous further development of our approach. For one thing, external factors can necessitate adjustments, for instance, if better technologies or approaches are available in the future. Secondly, also the project-internal demands are likely to change, especially as this prototype should also serve for identifying the precise needs for such a software system.

Hence, the architecture of our approach should be as adaptable as possible. For example, further metrics besides active power are going to be supported later. Moreover, an extension by further analyses and visualizations is conceivable. Apart from those extensibility requirements, we also require the possibility to replace or remove components due to changing demands.

Fault Tolerance Our architecture should provide a high degree of fault tolerance, as described by Avizienis et al. [2004]. Following their definition, faults are the cause of errors and errors may lead to a failure of system components or the whole system. It is very likely that faults occur during the operation of the software. In particular, in distributed systems that communicate over the network, faults are almost impossible to prevent.

We therefore require our architecture to tolerate single faults in components as far as possible so that such components do not fail entirely. Additionally, the system should be able to recover from a state that contains errors into a working one. This should be done as automatically as possible and preferably without restarting the whole system.

5.1.2 Architectural Constraints

Limited Resources The devices or sensors that we connect to our approach typically operate on limited hardware resources (e.g., CPU or memory). Usually, that is not sufficient to execute complex analyses directly on them. Moreover, the given resource capacities are not or only limited extendable and, thus, impede scaling of the software.

High Amount of Data The amount of data handled by our approach depends on the number of connected sensors as well as on how frequently they gather data. If further metrics besides active power are to be monitored, this is another influencing factor. The amount of devices and machines and their corresponding sensors typically depends on the size of the production environment. The frequency of measurement data depends on how accurate analyses should be. Therefore, the amount of data can get huge for large production environments.

When outlining our architecture, this has to be considered for processing the data as well as for their storage. Moreover, the transmission of data can become problematical.

5.2 Software Architecture

Considering the architectural requirements and constraints, we decided to design a microservice-based architecture for the desired monitoring infrastructure. An approach

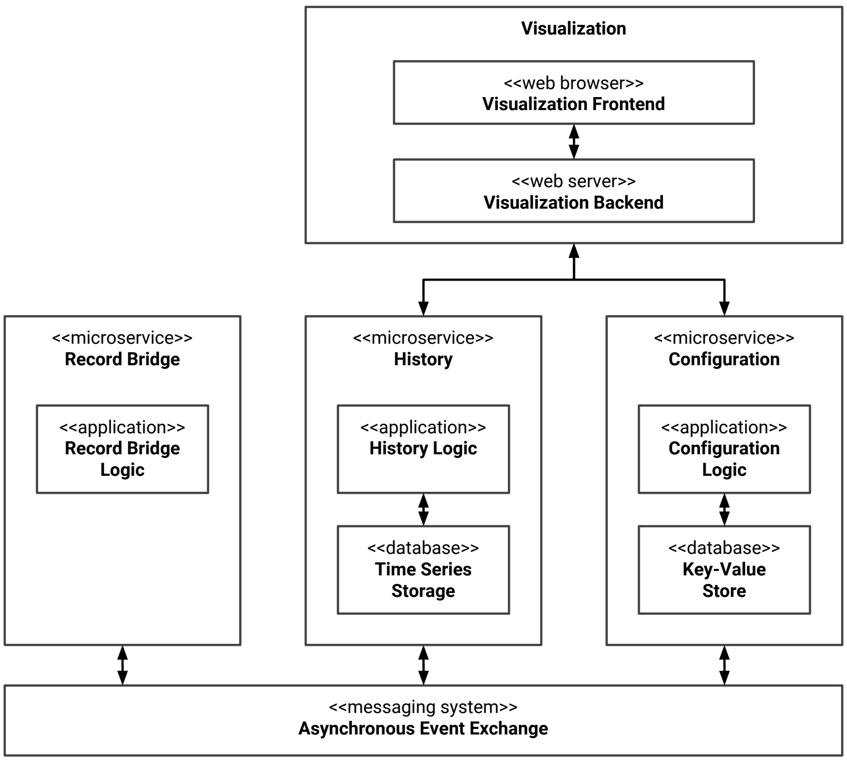


Figure 5.1. Microservice architecture for analyzing and visualizing metrics of a production environment in real-time.

based on a distributed system is already required by the demands for scalability. Also fault tolerance can be achieved significantly better in a distributed system. Following the microservice pattern involves us to encapsulate components according to business capabilities, which facilitates a better adaptability. In contrast to other distributed system patterns, microservices allow for an individual scaling of components that is independent of each other. That supports scaling the whole system more fine-grained and avoiding to waste resources as only those components can be scaled for which it is necessary. Since the individual services only require normal network connections between them, they can be deployed in totally different contexts. This offers a lot of freedom in the operation of the software (see Section 5.3).

Figure 5.1 shows a graphical representation of our architecture. It contains the three microservices *Record Bridge*, which integrates sensors, *History*, which aggregates and stores sensor data for the long term, and *Configuration*, which manages the system's state. Whereas the Record Bridge solely contains application logic, the services History and Configuration additionally contain a data storage subcomponent. The *Visualization* component is not a typical microservice as it does not represent an own business function but instead serves as an integration of different business functions. It consists of two parts, a backend and a frontend (see Section 5.2.4).

The services in our architecture communicate with each other in two different ways: firstly, synchronously using a request-reply API paradigm such as REST [Fielding 2000] to read or modify the other services' states; secondly, via a messaging bus or system to publish events that may be asynchronously consumed by other services. Using both communication approaches together is a common pattern when designing microservices [Hasselbring and Steinacker 2017].

As stated in Chapter 4, the major task of our approach is stream processing of sensor data. Figure 5.2 shows the data flow among components for this specific task. From a more functional than organizational oriented perspective, our architecture can also be considered as a Kappa architecture. It depends on the further development of our approach whether it should be considered more as an interacting microservice-based architecture or more as a stream processing approach with a speed, a serving, and perhaps also a batch layer.

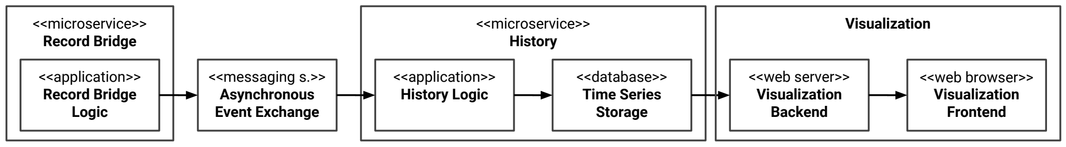


Figure 5.2. Data flow between the different components when processing new measurements. The Record Bridge receives monitoring data from physical sensors and queues them into a messaging system. From there, the History service aggregates the data and stores them into a database. The visualization's backend queries that database and forwards the monitoring data to the frontend.

5.2.1 Record Bridge

Sensors use different schemata, technologies, and transport mechanisms. Transportation can take place with high-level techniques such as HTTP but also on a low-level, for example, via serial data buses. Data can be encoded in text formats such as JSON or XML but also binary. And besides several standardized data schemata, there are also numerous proprietary ones. This requires to convert sensor measurements to a unique format that is used inside our whole approach, before they will be further processed and analyzed.

The Record Bridge fulfills this task. It receives the sensor data, transforms it, and then publishes it for other components by sending it to the messaging system. In our architecture design, it functions as a placeholder for arbitrary concrete Bridges, where each Record Bridge integrates a specific sensor type. As a sensor type, we consider a set of sensors that use the same schemata, formats, and transport mechanisms.

Record Bridge services are only supposed to convert data from one format into another. They do not need to have any knowledge about previous transformations and therefore can be designed stateless. This should also be respected in implementations since stateless components enable an arbitrary scaling.

An example of a Record Bridge for a concrete sensor is described in Section 6.3.2.

5.2.2 History

The History service manages past sensor data and provides access to them. That includes real sensor measurements as well as aggregated data for groups of sensors. Thus, one task this component has to fulfill is the hierarchically aggregation of data. That should be done in realtime, which means: Whenever a sensor supplies a new measurement, all aggregated sensors that contain this sensor should obtain an update as well. Like for real sensors, this component creates a new record with the aggregated values and publishes it via the common messaging system for other services.

In order to access past measurements, they first need to be permanently stored. Therefore, the History service has access to a database and when it receives new records (aggregated or not) it writes them to that. For other services, the History service provides access to the database in the form of an API, which has various endpoints that return records or statistics on them. Which these are exactly depends on the other services' needs.

The application logic is separated from the actual data storage. Thus both parts can be scaled independently. When choosing a database management system (DBMS), it should also be considered how well it can be scaled—both in terms of accessibility and storage. Even if the data retention is segregated into a DBMS, the application logic still cannot be considered entirely stateless. This is due to the fact that multiple instances need to coordinate themselves when consuming data from the messaging system or aggregating them.

The tasks "hierarchically aggregation" and "management of past data" could also be divided into two separate services. If this is reasonable highly depends on where the division into functionalities is made and if an independent scaling of both tasks is needed. Therefore, when implementing this service, both tasks should be encapsulated so that they can still be divided later.

5.2.3 Configuration

The Configuration service manages the system-wide settings of our approach. However, it does not serve as a central place for all configurations of individual services. Settings that clearly belong to a specific service should be configurable directly in that service. An essential requirement of this service is the ability to handle reconfigurations during the execution. In other words, no restart should be needed whenever the configuration changes and other services will receive notifications about those changes. Therefore, the Configuration service provides an API to update or request the current configuration and propagates updates via the messaging system.

Furthermore, this service contains a database to store the current configuration. It is the database's responsibility to store data in a reliable and perhaps redundant manner. Separating the database from the API logic also allows to scale both of them independently.

Currently, the only system-wide-managed resource is the *Sensor Registry*. It is a hierarchical tree structure specifying what sensors exist and how they could be aggregated.

5.2.4 Visualization

Besides monitoring and analysis, our approach also includes an interactive visualization. This visualization is a single-page application running in the web browser of the user. Our architecture contains a Visualization component following the *Backends for Frontends* pattern (see Section 2.2.2). It consists of two parts: The *frontend* is the actual single-page application. The *backend* is a simple web server whose sole function is to deliver the frontend via HTTP and to redirect requests to other services. From the backend's view, the frontend is only a set of static files that will be interpreted by the web browser. Therefore, it does not have to handle user events or manage any internal state. Instead, all interactions with the backend will be forwarded to the corresponding microservices and will be handled by them.

5.3 Deployment Architecture

The software architecture is designed to allow for an individual scaling of its components. We expect this to enable us to react to varying loads and requirements. Moreover, this scalability should make our approach feasible for different sized environments. However, it is not sufficient that the software architecture supports this, in the end the software must also be deployed accordingly.

For those reasons, large parts of our architecture are supposed to run in a computing center or cloud infrastructure instead of directly on hardware in the production. Besides scalability as mentioned above, this ensures that the data analyses do not conflict with the production process. Furthermore, it is likely that the hardware in the production is not powerful enough to run all components of the architecture.

The requirements of the deployment infrastructure vary depending on the production environment. For small environments or in the beginning, it may be sufficient to deploy it on one single server. When the amount of monitoring data increases, scalability can be achieved by adding more servers. For even larger production environments or when dynamic scaling is required, our approach is supposed to run on private or public clouds.

However, it may also be reasonable to run particular parts directly in the production environment. Applying the idea of edge computing, we can already reduce the monitoring data where it is recorded. That can be achieved by using appropriate filter or aggregate functions. In our architecture design, the Record Bridge can fulfill such tasks but there can also be an external edge controller doing this. Thus, the following four deployment combinations are conceivable:

1. There exists no separate edge component and the Record Bridge is deployed along with the other services in a computing center or cloud infrastructure. That is sensible if the Record Bridge needs to be scaled dynamically or if there is no appropriate hardware or software infrastructure available in the production environment. Also this approach is likely to be simpler to realize as its deployment would not differ from the deployment of the other services.

2. There is no separate edge component but the Record Bridge is deployed in the production environment. In this case, the Record Bridge acts as a kind of edge component that already filters and aggregates monitoring data.

3. There is a separate edge component and the Record Bridge is deployed in a computing center or cloud infrastructure. This option is reasonable for the same cases as the first one. However, it takes advantage of an edge component.

4. Both an edge component and the Record Bridge are deployed in the production environment. This is perhaps the most future-oriented alternative if hardware gets more powerful and data transmission becomes the limiting factor. Depending on the edge controller, it may even be possible to execute both components on the same machine. As data are usually aggregated by the edge component, the Record Bridge solely serves for converting the measurements into a more efficient data format. Only if the aggregation is not configurable enough and an additional filtering of data is necessary, it is reasonable to perform further aggregations by the Record Bridge.

These approaches can also be arbitrarily combined to adapt to the situation of the existing infrastructure, instead of adapting the production to our approach. Figure 5.3 presents all four approaches within a hypothetical deployment.

To realize this deployment architecture, all server-side running components need to be implemented as Docker images. If possible, subcomponents should be realized as separate

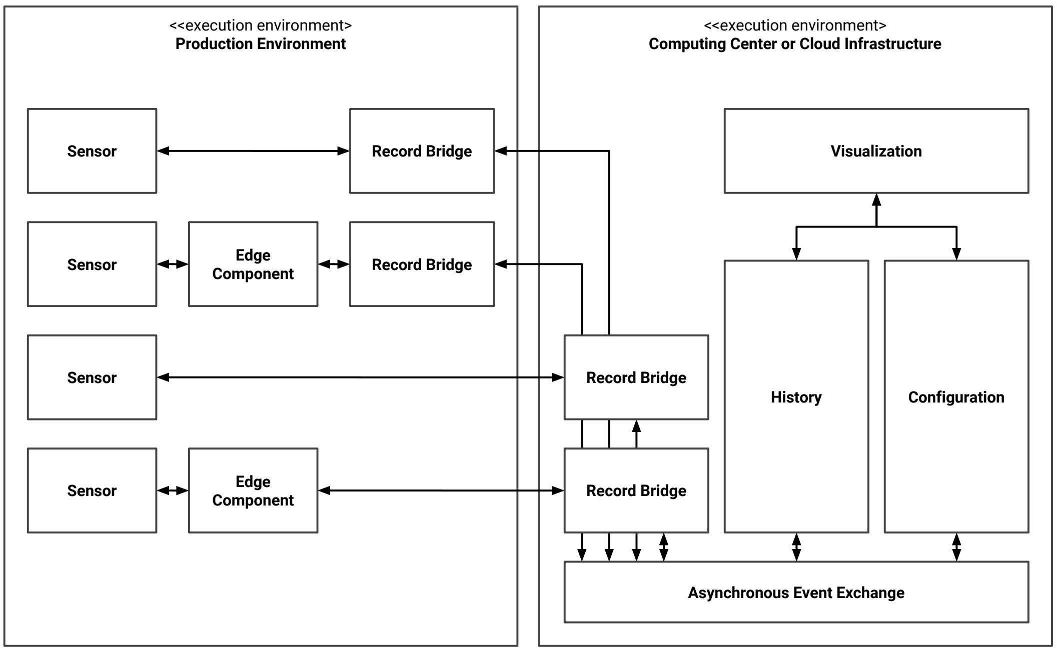


Figure 5.3. Deployment architecture showing all possibilities how the Record Bridge and an additional edge component can be deployed.

images. For example, the database and the implemented logic of a microservice should be deployed as individual containers to allow for an independent scaling of each other. Whereas there are ready-to-use images for most components from external sources (e.g., databases), we have to create Docker images for the self-implemented components.

Everything that is required for running our components is defined in these images. In particular, these are software dependencies as well as the necessary system configurations. Therefore, to deploy our approach, there is no particular configuration of the servers or the cloud infrastructure necessary. Instead, the execution environment merely has to provide a way to run Docker containers as well as appropriate network capabilities. This way, it is also possible to simply move components between servers. A further reason for using containers is the scalability they entail. An arbitrary amount of Docker containers can be created from images that can be started and stopped within seconds. Thus, it is possible to react on changing loads dynamically.

An implementation of the described architecture is intended to be operated by Kubernetes. Therefore, we realize all components as Deployments or StatefulSets. For self-implemented components, we map Deployments directly to their corresponding Docker images. Using these Deployments, we can create Pods as needed, whereby each Pod contains only exactly one Docker container of the corresponding component. The components that manage state across restarts such as databases have to be realized similar but as StatefulSets. Furthermore, we create Services to define the interfaces pods can communicate with each other and the outside. This approach enables a dynamical and flexible scaling of our deployment within a Kubernetes cluster. Whenever we need to run more or less new instances of particular components, we only notify Kubernetes about the desired amount of instances. Then, Kubernetes creates new Pods or terminates running ones as described in Section 3.7. In small or prototypical application scenarios, it may also be reasonable to deploy our approach without Kubernetes and to deploy containers manually.

How record bridges are deployed within the production heavily depends on the available infrastructure. However, there are also recent trends towards container-based architectures in production environments [Goldschmidt et al. 2018].

Chapter 6

Implementation

Based on the previously described architecture, we developed a prototypical realization of it. This chapter describes an implementation comprising all components required by the architecture as well as the selection of suitable technologies if necessary (e.g., databases). In the following, we first explain how we implemented the communication between services in Section 6.1. In this prototype, monitoring records are of a common format, which we realized using Kieker as described in Section 6.2. In Section 6.3, we present a generic framework to implement Record bridges and an exemplary realization of one service. Section 6.4 describes the implementation of the History microservice and Section 6.6 of the Configuration service. Finally, in Section 6.6, we present a visualization for our approach.

6.1 Communication between Services

In our developed architecture, we demanded the usage of a messaging system for a reliable and asynchronous communication between services. In our implementation, Apache Kafka undertakes this task. Event-based data, especially monitoring data, are exchanged via Kafka topics. In the default configuration, our implementation uses the following three topics: `records`, `aggregated-records`, and `configuration`. As their names suggest, they contain "plain" records, aggregated records, and notifications about a changed configuration. The number of partitions does not depend on our implementation but solely on the eventual deployment. In order to support an automatic partitioning, messages that are written to the topics need to include a key. For records and aggregated records this key is the identifier of their corresponding sensor or sensor group, respectively. Events of a changed configuration have the name of the changed property as their key.

Another way for services to communicate with each other is via REST [Fielding 2000]. Most services provide a REST interface that can be accessed by other services to request specific information.

6.2 Defining and Transmitting Records with Kieker

In order to enable different services working on the same monitoring data, there must be a common data format that is understood by every service. Even if our prototype only supports active power as a metric, there should be a format to exchange data that is as generic as possible.

The Kieker monitoring framework faces similar challenges, although in the context of software performance monitoring. Even if large parts are not relevant, we decided to adapt Kieker for our implementation. For example, in our approach data recording is done by external sensors while for software systems this is done by Kieker. Moreover, many analyses performed on software metrics are not transferable to other forms of monitoring data. Nevertheless, using Kieker affords some advantages, which are:

1. The Instrumentation Record Language (IRL) enables us to define record types in a generic way. Besides record types for active power, we can later define record types for other metrics which can then use our implemented infrastructure.

2. Using its writing and reading infrastructure, Kieker facilitates an abstraction of the record transmissions and storage process. Kieker already provides writers and readers for Kafka and the database Cassandra, which only had to be adapted for our needs. Moreover, we can easily expand our support to various other technologies supported by Kieker.

3. In a later step, our approach could also process software performance monitoring data such as response times and hardware utilization. Kieker features several means to monitor and analyze those data so that using Kieker for this prototype already provides the necessary infrastructure.

6.2.1 Kieker Records for Active Power

We define record types for active power measurements and aggregations of multiple active power measurements. An `ActivePowerRecord` consists of an identifier for the sensor it contains data for, a timestamp with millisecond precision, and the actual value measured by a sensor in Watt. An `AggregatedActivePowerRecord` also comprises an identifier, which is the name of the corresponding group of sensors, and a timestamp. Furthermore, it contains multiple aggregated values, namely the minimum, maximum, average, and sum of all records that are aggregated and, moreover, the count of them.

Declaring those types using the IRL, Kieker generates classes for the records and, additionally, corresponding factory classes that create records following the *abstract factory pattern* [Gamma et al. 1995]. Currently, we only need the generated Java classes but the IRL also generates classes for other programming languages. Figure 6.1 shows a UML class diagram describing the generated classes. Kieker provides the generic concept of serializers and deserializers that are used by the different reader and writer implementations.

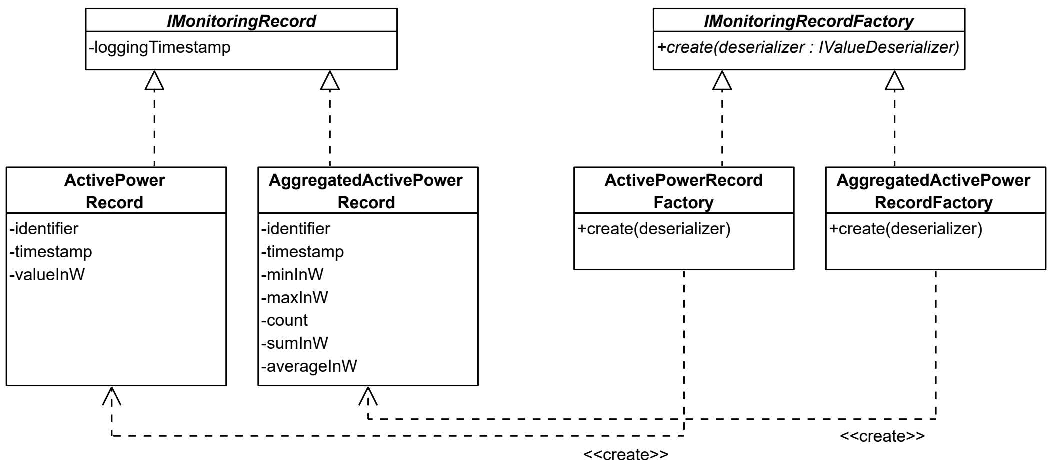


Figure 6.1. UML class diagram of the generated record and factory classes. Only the attributes and operations that are particularly important for our implementation are displayed.

6.2.2 Adapting Kieker's Support for Apache Kafka

Kieker already provides a pair of writer and reader to transmit records via Kafka. This technique uses a *chunking* mechanism, meaning that it combines a set of Kieker records into a larger Kafka message. Chunking records facilitates that Kafka messages are self-contained, i.e., they are processible without knowing about previous records [Knoche 2017]. In order to reduce the amount of data that are transferred, Kieker uses the concept of a string registry that maps each string to an integer value and vice versa. Thus, when the same string should be sent multiple times, only its corresponding integer value needs to be transferred. When using the chunking method, every chunk of Kieker records contains an own string registry.

In our desired approach, however, real-time data processing was a key requirement which we only wanted to omit if it would be absolutely necessary. A second flaw is that Kieker's current approach does not support specifying keys for messages, which, however, is essential for partitioning Kafka topics. In particular, there is no way to key records by their identifier as a chunk message can contain records with different identifiers.

Therefore, we decided to develop an alternative Kafka writer and also an alternative reader by adapting the current ones. Our approach directly maps one Kieker record to one Kafka message by binary serializing all its fields. By means of a configurable function, a key can be extracted from the record's properties. In our implementation, we use the sensor identifier as the key. In contrasts to all other Kieker writers and readers, our approach does not use string registries so far. Instead, we are writing strings directly to the records. For this reason, our implementation also provides a new registry-less serializer and deserializer. As we are using only a low amount of strings and we are also able to eliminate this if necessary, the direct serialization of strings is the simplest approach for us.

Even if this approach is working well for us, we should remark that in many use cases this option is not practicable and transmitting strings redundantly should be avoided. One strategy to solve this could be to transfer the string registry via a separate Kafka topic. Then, multiple readers can still distribute the record processing between them and do not need to coordinate themselves. Solely, all have to subscribe to the topic with registry updates. Since usually the number of monitoring records is significantly larger than the number of registry records, it is likely that this approach is feasible.

6.3 Integration of External Sensor Data

6.3.1 A Generic Framework to Integrate Sensor Data

As described in Section 5.2.1, there should be an individual *Record Bridge* service for every type of sensor that integrates the data of those sensors into our approach. However, the tasks that are fulfilled by those services are largely equal. They have to start the application, load configuration parameters, run continuously, and write records into Kafka topics. They only differ in the way how they receive or request data and how they convert those into Kieker records.

Due to the unqualified variety of different manufactures, devices, standards, and protocols it is very likely that, when applying our approach in a new production environment, there does not exist the necessary bridge and, therefore, a new one has to be developed. In order to make our approach extensible and adaptable, there should be a simple way to connect new sensors. Since it would be very cumbersome, however, to develop those bridge services always from scratch, we decided to design and provide a framework that eliminates repetitive tasks as much as possible.

This framework should take all the work such that only the sensor or protocol specific logic have to be implemented in order to create a new bridge. The typical procedure a bridge service has to manage corresponds to the following pipeline: Records are created, logged to measure the throughput, and, subsequently, written to a Kafka topic. Our framework implements this pipeline by using the Pipe-And-Filter framework TeeTime. The first step, creating records, depends on the sensor that should be integrated and, hence, has to be configured by the framework's user. In many cases, this first step can also be separated into multiple steps, for example, if external measurements are received in a first step and the conversion into Kieker records is handled in a second step. More generally we can say, the pipeline starts with exactly one *producer stage* followed by any number (including zero) of *transformer stages*, whereby the last stage has to emit Kieker records.

One can construct this pipeline in two different ways: Either via a TeeTime *Configuration* or via our Streaming DSL. While the TeeTime Configuration allows for more flexible set-ups (e.g., branching and concurrent execution), the Streaming DSL provides a way to define the pipeline in a declarative manner. This way, the user of the framework does not have to have any knowledge about TeeTime. The Streaming DSL is described in more detail in the following. Furthermore, the framework is highly configurable by parameters and action that are executed when starting or stopping the service.

TeeTime Streaming DSL

A stream is a data structure for a (often conceptual infinite) sequence of data that is lazy evaluated. Originating from functional programming (also known as infinite lists) [Bird and Wadler 1988], nowadays also classic object-oriented languages such as Java provide streaming APIs [Urma 2014].

In the following, we present a simple Java-based internal domain-specific language (DSL) [Fowler 2010] that allows for declaratively describing streams or, more precisely, their creation and operations on them. Streams constructed this way are typed, which means that all elements of a stream are of a specific type and, especially, this type is known at compile-time.

Generally, streams can be created in different ways. For the record bridge, we identified two typical use cases to obtain sensor data, which require different ways to create a stream. The first one is by actively querying a sensor and the second one is by passively listing for messages pushed by the sensor. We provide implementations for both options and describe them in the following.

querying A stream can be created by proving a function $\varnothing \rightarrow X$ where X is the type of the stream's elements. This function will than be constantly invoked and the elements it returns are appended to the stream. This method is reasonable if, for example, the current value from a sensor should be periodically requested.

listening A stream can also be created by providing a queue of elements that is filled by an arbitrary external source. Then, the elements that are read from this queue correspond to the elements of the resulting stream. This method can be used if receiving data and processing data should be handled asynchronously. This is the case, for instance, when a sensor pushes data. Listeners (e.g., HTTP server workers) can listen for new data and write them to a queue.

Similar to functional programming concepts or the Java Stream API, we provide a way to transform streams. We implemented the following operations that transform one stream into another one. Those operations will be executed *lazy*, meaning that all operations are executed on a single element before the next element is processed. The naming of operations conforms to the analogous operations in common functional programming languages such as Haskell.

map It takes a function that maps every element of this stream to an element of the new resulting stream. The elements of the new stream can be of a different type than the ones from the original stream.

flatMap It takes a function that maps every element of this stream to a multiset of new elements, again possibly of a different type. The newly created elements are appended to the resulting streaming. This means, the resulting stream corresponds to an ordered union of the returned multisets.

filter It takes a predicate (Boolean-valued function) and returns a new stream containing all elements of the initial stream which the predicate returns *true* for.

Based on a stream defined that way, our framework is able to create an appropriate TeeTime pipeline. For this purpose, it creates a *Producer Stage* from the stream-creating function and for every operation (*map*, *flatMap*, and *filter*) on this stream it creates a suitable *Transformer Stage*. Even though we designed the Streaming DSL to develop record bridges more simply, our framework is not restricted to this and can also be used in other contexts where TeeTime pipelines should be declared at a high abstraction layer.

6.3.2 An Exemplary Bridge for a Power Distribution Unit

In the following, we present an exemplary realization of a Record Bridge that integrates data from a power distribution unit (PDU) into our approach. A PDU is a device that distributed electrical power from one or more inputs to multiple outputs. In data centers it is used to power servers and similar devices. Many PDUs allow for requesting the current status and consumption values via network protocols.

An example for such a device is the *Rack PDU PX3-5911U* by the manufacture *Raritan*. It features 48 outputs and monitors metrics such as *active power* or *line frequency* for each of them. The device is connected to the network and, thus, provides interfaces to access those data. Besides a web-based user interface (Web UI) and a SNMP (Simple Network Management Protocol) interface, the PDU can be configured to push selected metrics via HTTP to a web sever. Using the Web UI, one can select for each outlet individually which data should be transmitted. As the PDU requests the sensor data every second, but perhaps records larger intervals, it aggregates the data and memorizes the minimum, maximum, and average value of the recorded interval. Depending on the actual configuration, the interval in which data will be send can be even larger than the recording interval so that one push action has to transfer multiple data sets. Summing up, it can be said that a push record of the Raritan PDU encompasses several metrics for each sensor and for each metric, it contains data for several timestamps. Figure 6.2 shows a simplified UML class diagram of this data structure. Record instances are encoded in the JavaScript Object Notation (JSON) and transferred via HTTP POST requests. To reduce redundancies in the HTTP messages, the actual encoded messages look a bit different. In order to consider a push event as successful, the web server has to return the HTTP status code 200.

The *Raritan Record Bridge* is an implementation for a Record Bridge as described in Section 5.2.1. It provides a web server that receives data pushed by the Raritan PDU and employs the framework described in Section 6.3.1 to further process these data and to set up a runnable application.

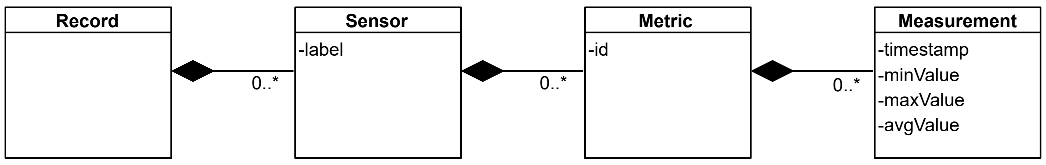


Figure 6.2. Conceptional UML class diagram of Raritan records showing only those attributes we actually use.

The web server is realized by the Java framework *Spark*[1] and provides exactly one route. It receives POST requests at the (configurable) URL `/raritan`, writes the received data into a queue, and confirms the request afterwards. The actual data processing is done asynchronously in a next step. As usual with a web server, it creates a thread pool [Mattson et al. 2004] so that multiple requests can be handled at the same time and each request is handled in a separate thread. Thus, multiple threads can write into the queue, however, only one needs to read from it (see below). To improve performance and throughput, we are using the lock-free *Multi Producer/Single Consumer* queue of *JCTools*[2].

The Record Bridge framework is now used to build a runnable application. By setting the configurable start and stop actions, the web server is started and stopped, respectively. The sensor data are processed by a pipeline that is declared using the Streaming DSL. The stream is created by a queue that supplies the raw data coming from the PDU. Those are all requests as strings received by the web server. In the next step, all JSON records are transformed to Kieker records by using the *flatMap* operation. The framework manages to create the TeeTime stages, appends the Kafka sender stage, and executes the resulting TeeTime configuration.

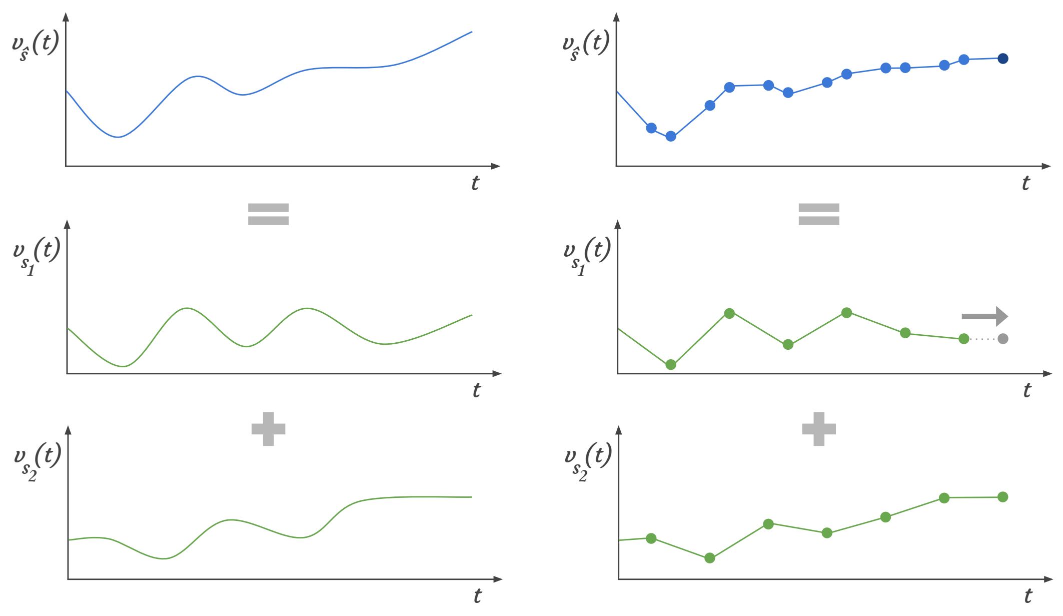
6.4 Hierarchical Aggregation and Archiving

The History microservice comprises three internal components that can be viewed mostly isolated of each other. In the following, we first describe all these parts separately and, afterwards, explain how they are connected.

6.4.1 Continuous Aggregation

In the following, we first present a method to calculate aggregated sensor data continuously and then how we realized a scalable implementation of this method.

[1]`http://sparkjava.com`
[2]`http://jctools.github.io/JCTools`



(a) The theoretical computation methodology assuming continuous values for s_1 and s_2.

(b) The practical computation with discrete values for s_1 and s_1. In order to compute the dark blue colored value for $\hat{s}$, the last present value of s_1 has to be shifted forwards.

Figure 6.3. Exemplary computation of the values for an aggregated sensor in the course of time. The aggregated sensor $\hat{s}$ has two child sensors, s_1 and s_2, and its value $v_{\hat{s}}(t)$ is defined by the sum of $v_{s_1}(t)$ and $v_{s_2}(t)$.

Calculation Methodology

For an aggregated sensor $\hat{s}$ that should aggregate the sensors $S = \{s_1, \ldots, s_n\}$, its value $v_{\hat{s}}(t)$ at time t is given by the sum of its child sensors' values at that time:

$$v_{\hat{s}}(t) = \sum_{s \in S} v_s(t)$$

However, since measured data are only present for discrete points in time, $v_s(t)$ for $s \in S$ is not known for many times. Furthermore, $v_s(t')$ with $t' > t$ is not known since the value should be computed in real-time and thus t' would be in the future. Therefore, it is not possible to perform a simple linear interpolation between the precedent and successive value. That means in effect, to compute $v_s(t)$ we can only rely on previous values.

For our approach, we simply equate $v_s(t)$ to the latest measured value. For the interpretation of those data, this means that the time series of the single sensors are shifted towards the future, whereby the shifting interval is at most the temporal distance between measurements. Figure 6.3 illustrates this.

If the data sources are measured frequently enough and the values do not fluctuate too much, this procedure should not influence the result notably. However, if that is not the case, our approach needs to be extended to consider more previous values and to calculate a potential value for $v_s(t)$ based on an appropriate forecasting algorithm.

Realization with Kafka Streams

In order to implement the calculation methodology described above, we designed a stream processing pipeline using Kafka Streams. Figure 6.4 illustrates this pipeline and pictures the individual steps, which we describe in detail below.

The initial data source is the Kafka Topic `records`. As described in Section 6.1, it contains key-value pairs with a normal active power record as value and its corresponding sensor identifier as key. This topic serves as an interface to the outside of this microservice since it gets its records form other services, namely the Record Bridge service.

Our Kafka Streams configuration consumes the elements of this topic and then forwards them to a **flatMap** processing step. In this step, every record is copied for each aggregated sensor that should consider values of this record's sensor. This means, if a new record is processed, the sensor registry is traversed bottom-up and all parents of the corresponding sensor (parents, grandparents etc.) are collected in a list. For each entry of this list, the flatMap step emits a new key-value pair with the according parent as key and the active power record as value.

Those key-value pairs are then forwarded to a **groupByKey** step, which groups records belonging together by serializing them to an internal Kafka topic. Thus, it ensures that all records with the same key are published to the same topic partition and, hence, processed by the same processing instance in a following step (see below).

The subsequent **aggregate** step maintains an internal *aggregation history* for each aggregated sensor that is processed in the course of time. An aggregation history is a map belonging to an aggregated sensor that holds the last monitoring value for each of its

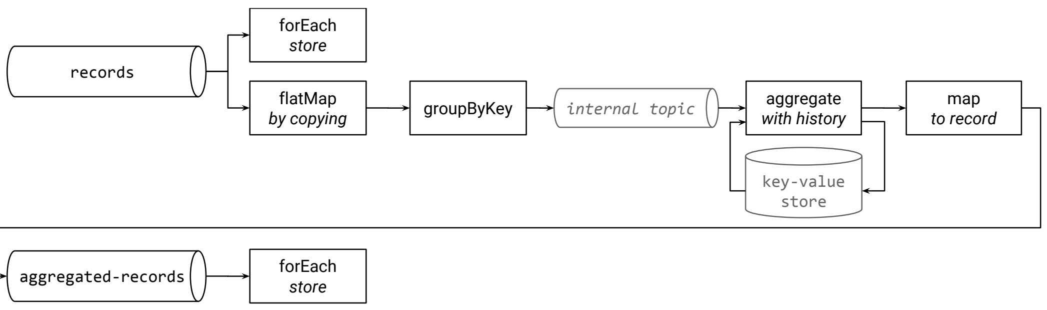


Figure 6.4. Graphical visualization of our implemented Kafka Streams topology. Horizontal cylinders represent Kafka topics, the vertical one a database. Grayed-out elements are implicitly created by the framework.

child sensors. It only stores the value for its real child sensors, not for the aggregated ones. Whenever a record arrives with a key for which no aggregation history exists so far, a new one is created. For all successive records the aggregation history is updated by either replacing the last value to this sensor or by adding it if no value for this sensor exist so far. Finally, it is, firstly, stored to an internal key-value store to be used in the next aggregation step and, secondly, forwarded to the next processing step.

Afterwards, the aggregation history is transformed to an aggregated active power record in a **map** step. This is done by calculating different statistics, such as average or sum, of the set of single monitoring values. These aggregated records are then written to the Kafka topic `aggregated-records`. This topic is again designed as an interface such that other services can consume those data, for instance, to perform data analyses on them.

Besides those steps for the hierarchically aggregation, the pipeline also contains two **forEach** steps that store the records from both topics `records` and `aggregated-records` to a database.

Whereas we declare the single steps of this data processing pipeline, the connection of the steps as well as the serialization to internal topics or databases is handled by Kafka Streams. If multiple instances of this application are started, Kafka Streams manages to balance the data processing subtasks appropriately. A fundamental principle of Kafka Streams is that partitions are always processed by the same instance since in this way no synchronization between reading instances is necessary. That means using this approach, we can create as many instances as there are partitions for the record and the aggregated record topics. As the number of partitions is bounded by the number of different keys and the keys correspond to the connected sensors, we can start as many instances as there are different sensors and aggregated sensors. However, this might only be a theoretical limit since the number of sensors will probably be much larger than the number of computers that are supposed to process this data.

6.4.2 Storage of Processed Records

Our architecture specifies a database to permanently store monitoring data. Monitoring records are flat, i.e., not nested, data structures that are not directly related to each other. Therefore, we do not need to support queries spanning multiple tables, e.g., using *joins* in relational databases, or nested data sets, e.g., as common in document-oriented databases. However, we require a high writing speed of the database in order to process monitoring data quickly. The reading speed is less important as external data queries are made via HTTP which is not designed for high-speed (see Section 6.4.3). Furthermore, those external requests only restrict the volume of records and, therefore, we do not need to support complex database queries such as groupings or sub queries. In consideration of these impacts, we decided to use the column-oriented database Apache Cassandra.

When writing monitoring records to Cassandra, we are using one table per record type. For our implementation that means, we need two tables: One for normal active power record and one for aggregated ones. When storing a record, it is mapped to a row so that for each record property there exists a corresponding column in the table. As the partition key, we use the sensor identifier of a record. This guarantees that records of a sensor are stored together on the same node and we can select multiple records of a specific sensor with only one query. The second part of the primary key, the clustering columns, is the timestamp. This ensures that records are stored in chronological order and we can poll all records of a specific time interval with only query.

Moebius and Ulrich [2016] presented an approach on how to store Kieker records in a Cassandra database. We build upon their approach and developed a generic serializer and deserializer for Cassandra. For this purpose, we brought it in accordance with Kieker's reworked writing architecture [Strubel and Wulf 2016] and designed a new Cassandra writer for Kieker. This writer aims for supporting different approaches on how to store data and, therefore, is highly configurable. Following the *strategy pattern* [Gamma et al. 1995], a strategy can be set to determine a table name based on a record. Our writer automatically creates the necessary tables if they do not exist already. Also the selection of the partition key and the clustering columns is configurable via a strategy. Furthermore, one is able to configure whether the record type and Kieker's internal logging timestamp should be stored. The serializer and deserializer perform a generic transformation from Kieker records into a cassandra row and vice versa. All in all, this approach is designed in a way that allows for storing arbitrary record types and that does not need any explicit configuration for individual record types per default. Nevertheless, it can be configured in order to store the records in the desired schema.

6.4.3 External Data Access

In order to provide access to the stored data for other components, the History microservice provides a REST API. It comprises various endpoints that either return a specific extract of the data or aggregations on them. In our prototype the required endpoints are determined by the needs of the visualization. A list of available endpoints is given below.

`GET /active-power/:identifier`
`GET /aggregated-active-power/:identifier`
Returns all records for the given sensor. If the optional parameter `after` is present, only those records are returned that have a timestamp greater than `after`.

`GET /active-power/:identifier/latest`
`GET /aggregated-active-power/:identifier/latest`
Returns the latest records, i.e., those having the largest timestamp, for the given sensor. The optional parameter `count` specifies the number of records that should be returned. If not defined, only the most recent one is returned.

`GET /active-power/:identifier/trend`
`GET /aggregated-active-power/:identifier/trend`
Returns a number indicating how the value (average value for aggregated records) of

records evolves over time, optionally only considering records newer than `after`. To calculate this number, the averages of the first record values and the last record values are computed. The number of those records is specified by the optional parameter `pointsToSmooth` (10 per default). Then, the end average value is divided by the start average value and, subsequently, returned.

`GET /active-power/:identifier/distribution`
`GET /aggregated-active-power/:identifier/distribution`
Returns a frequency distribution of records. For this purpose, each record is assigned to a bucket, where each bucket represents an equally-sized interval of values. The `buckets` parameter defines the number of buckets, which is 4 per default. Using the parameter `after`, the set of considered records can be restricted.

Our implementation starts a web server using the Java framework Spark. This web server accepts user requests on the specified routes and passes them to an internal class that creates corresponding database requests. The responses from the database are converted into the JSON format and returned to the requestor.

6.4.4 Connection of Internal Components

When starting up the History microservice, the following two tasks need to executed: Initial request of the sensor registry from the Configuration service and establishing a connection to the database. Both are calls to external services and there is no guarantee that these requests will succeed. This becomes especially important when starting up the whole deployment and the History microservice is earlier available than the other services. Thus, we implemented mechanisms to automatically repeat these tasks on failures. Furthermore, these tasks should be executed in parallel in order to reduce the overall start up time of the History service.

During the ongoing operation of this microservice, multiple internal components need to be run in parallel. These are the stream processing of monitoring records, the running of a web server to handle API requests, and listening for changes of the sensor registry that are propagated via the messaging system. Furthermore, the connections to the messaging system and the database need to be supervised. That is, in the case of faults or changing endpoints, the according connections must be reestablished. However, the corresponding frameworks we use handle that for our implementation.

Starting those continuous running tasks partly depends on the start of those tasks that are executed at the mircoservice's start-up. Figure 6.5 shows the individual tasks and how they depend on each other.

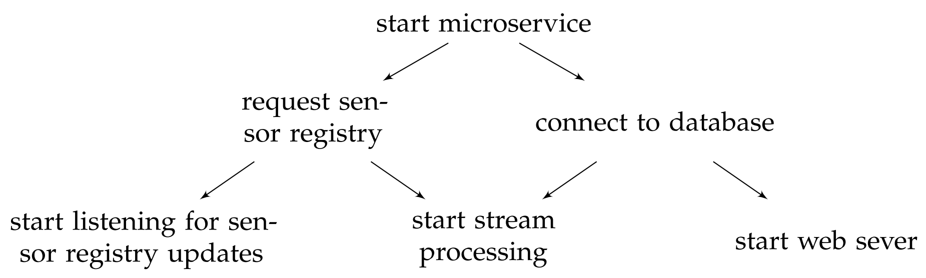


Figure 6.5. Dependency graph of internal tasks that need to be executed when starting up the History microservice. An edge from task A to task B indicates that A must be executed before B and, thus, B depends on A.

6.5 Configuration

The Configuration microservice manages system-wide properties as described in Section 5.2.3. It provides a REST interface that can be used by other components to read or update individual properties. We implemented the underlying web server for this interface with the Java framework Spark.

As also defined in the architecture, the properties should be storages via a database to ensure scalability and reliability of this service. The configuration properties are always key-value data structures. When the web server receives HTTP requests to read or update properties, it performs appropriate queries to the database to read or (over)write values for a given key. For our prototype, we decided to use the key-value database Redis[3].

On update requests, the web server also validates the new value and only stores it to the database if it is valid. Additionally, it publishes an *update event* via the messaging system. This way, other services can register for receiving notifications if certain properties change, instead of continuously requesting for updates. This allows other services to be informed earlier about changed properties and the network traffic is reduced.

In this prototype, the only property managed by the Configuration service is the sensor registry.

6.6 Visualization

Our architecture defines a visualizing user interface (UI) consisting of a frontend and a backend part. The backend solely forwards API requests to the responsible microservices and serves the frontend's source files (e.g., JavaScript source code, style sheets). The necessary logic for this is relatively simple and we provide an *NGINX*[4] Docker image

[3]`https://redis.io`
[4]`https://nginx.org`

implementing this. In the following, we therefore only describe the Visualization frontend in detail.

In Chapter 4 we list key reasons that motivate a visualization of the monitored data. There exists a wide variety of different ways to visualize data. Some are supposed to make the character of data comprehensible by providing as much detail as possible. Others aim to abstract the data as well as possible in order to display only the most essential information. The goal of all, however, is to enable the viewer to gain new insights of those data.

For the desired visualization of this prototype, we therefore provide various forms of visualizations that are supposed to serve different purposes. Conceptually, those visualizations are individual, self-contained UI components. That means, they are responsible for their own logic and appearance and solely depend on their environment via explicitly defined properties. This way facilitates an independent development of different visualizations and also enhances their reusability.

For the implementation, we use the JavaScript framework Vue.js, which allows for developing UI components in exact that manner. We wrote those components in the TypeScript programming language, which is compiled to JavaScript. In the following, we describe the visualization components we implemented. In addition to those, we also developed components for the layout (e.g., header bar), controlling (e.g., buttons) and user experience (e.g., loading spinner). In Section 6.6.2 we describe how we assemble all those components to create a single-page application.

6.6.1 Reusable Data Visualization Components

In the following, we describe four visualization components implemented in our prototype. Each of them uses the same data, i.e., the monitored power consumption. However, each of them provides a view on this data from a different perspective. Supported by according endpoints of the appropriate microservices, they use different degrees of data aggregation and abstraction.

The components themselves are responsible for communicating with the backend to receive the necessary data. Therefore, they load data asynchronously at runtime (*AJAX*) and may also periodically update their data. All components show data for a specific sensor. This can either be a normal or an aggregated one. When talking about the (monitored) value of a sensor, we refer to its actually measured value for normal sensors and to its computed sum value for aggregated sensors[5]. Figure 6.6 shows a screenshot of a web page with all four components.

Time Series Chart

This component features a time-value chart of sensors. It shows the course of the monitored active power in Watt (vertical axis) in relation to the point in time when that value was

[5]Since we only consider active power, we decided that using the sum is the most reasonable value. However, this is configurable and also another aggregation values, e.g., average, can be used.

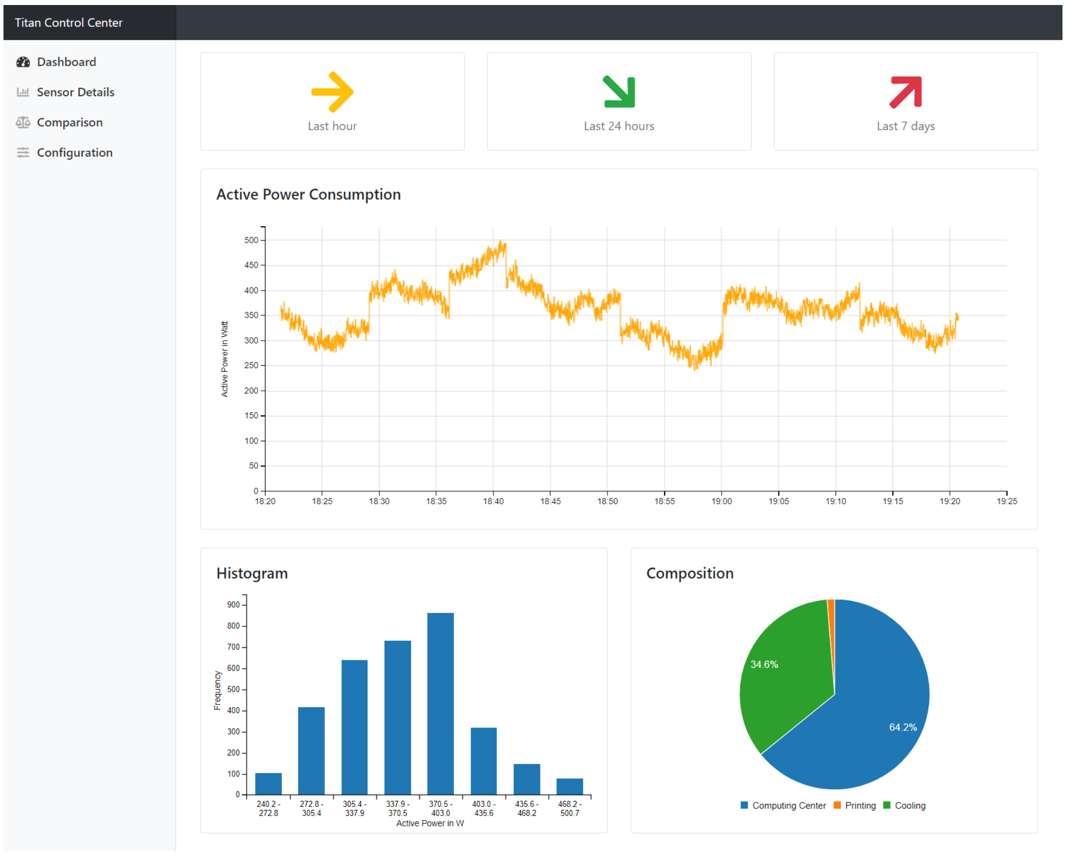


Figure 6.6. A screenshot of the dashboard view. As described in Section 6.6.2, it contains the trend arrow, time series chart, histogram and composition pie chart components.

measured (horizontal axis). The user can enlarge or reduce the displayed temporal interval or move forward and back by using the mouse or touch gestures. On a low zoom level the data points are only displayed as a connected line, whereas on a high zoom level the individual data points are shown. When hovering those data points with the mouse, tooltips appear and show the exact values.

At startup, this component loads monitored data for a specific interval. Afterwards, it continuously requests new data and adds them to the chart. If the new data do not longer fit into the current zoom level, the displayed interval automatically moves forward. Thus, it gives the impression of a chart that moves in time.

Research on how to efficiently visualize large data sets was conducted by Johanson et al. [2016]. In order to provide this visualization, we utilized their library CanvasPlot[6]. It is

[6]`https://github.com/a-johanson/canvas-plot`

based on the data-visualization framework D3 [Bostock et al. 2011]. CanvasPlot extends D3 and simplifies its usage by rendering the underlying elements (e.g., labels or the grid), handling user interactions, and especially visualizing the data in a resource-saving manner.

Composition Pie Chart

This component shows how the value of an aggregated sensor is composed of its child sensors. In a pie chart, it displays areas for each child, whereby the area's size corresponds to the proportional influence of that sensor to the aggregated value. The child sensors are descending sorted by their value. Then, in the pie chart they are displayed clockwise in that order, starting with the largest value top right. We realized this with D3 and a library built on top of it to simplify its usage, called C3[7].

This diagram is supposed to provide a simple overview of which devices or departments consume particularly much or particularly little. It does not serve to determine exactly how much the individual devices consume and also not to compare sensors with similar influences. Indeed, for such proposes, pie charts are not the appropriate tool [Spence and Lewandowsky 1991].

Histogram

This component provides a histogram to visualize the frequencies of certain measured values. Therefore, it sets up a configurable number of *buckets* and assigns each monitored value to one of these buckets. It divides the interval from the lowest to the largest monitored value into as many equally sized sub-intervals as there are buckets. Hence, each sub-interval can be assigned a bucket and to a bucket all monitored values are assigned that are within the corresponding interval. This component shows a bar for each bucket, whose hight corresponds to the number of elements within that bucket.

In order to reduce the amount of data transferred from the server to the client, the histogram component uses a suitable interface of the History microservice (see Section 6.4). Similar to the composition pie chart component, it uses the frameworks D3 and C3.

As described in Chapter 1, one objective of our approach is to provide opportunities to detect load peeks. This visualization allows for analyzing how evenly monitored values are distributed or for identifying whether there are single outliers (upwards or downwards). It serves as a supplement to the time series chart, which is supposed to be used for deeper analyses.

Trend Arrow

This visualization shows a large arrow that indicates how the values of a specific sensor evolved over a certain period of time. It uses the corresponding REST API of the History

[7] `https://c3js.org`

service (see Section 6.4) to get a numeric value representing this trend and translates the returned value into the direction of the arrow and a color.

The arrow is yellow and pointing to the right if the development is mostly stable (between 1.1 and 0.9). If the sensor's value increases over time by more than 1.1 it is displayed red and pointing up (larger than 1.5) or up right (between 1.1 and 1.5). If its trend is decreasing, it is displayed green and pointing down (less than 0.5) or down right (between 0.9 and 0.5).

The arrows along with the selected colors represent a kind of "traffic light". They are supposed to provide a simple assessment of the current consumption. Whether an increasing consumption is an actual problem or is expectable in the current situation depends on the operational context. Therefore, this visualization is only an addition to the time series plot that shows the trend in detail.

6.6.2 Assembling Components to a Single-Page Application

We realized the visualization frontend as a single-page application (SPA) using Vue.js. It does not only serve for the pure visualization of data, but is also a user interface for our entire approach.

This SPA consists of the described visualization components, self-defined auxiliary components, and components from external sources. We divided the application into four different views performing different tasks. The user can switch between the individual views via the navigation on the left site. A detailed description of the single views is given below.

Even if this is an SPA, i.e., a single HTML page serving all the different views, it enhances the user experience if it is possible to navigate back and forth in the browser nonetheless. Therefore, we applied Vue's routing extension, which fakes the browser to enter new pages when clicking on links. Each view has an individual route, i.e., a generic URI, assigned, via which that view can be called directly.

Dashboard

The dashboard is the start page of the SPA, which is shown per default if no other view is called via a specific route. It shows various visualizations that are all referring to the overall consumption, i.e., the top-level sensor of the sensor registry. A screenshot of this view is presented above in Figure 6.6.

In the upper area, it shows three instances of the *trend arrow* visualizations next to each other. They show the trend of the last hour, last 24 hours, and last 7 days. Below them, a large time series chart spans over the entire width. Per default, it shows the values for the last hours. Below that, a histogram and composition pie chart are placed with both taking half of the width.

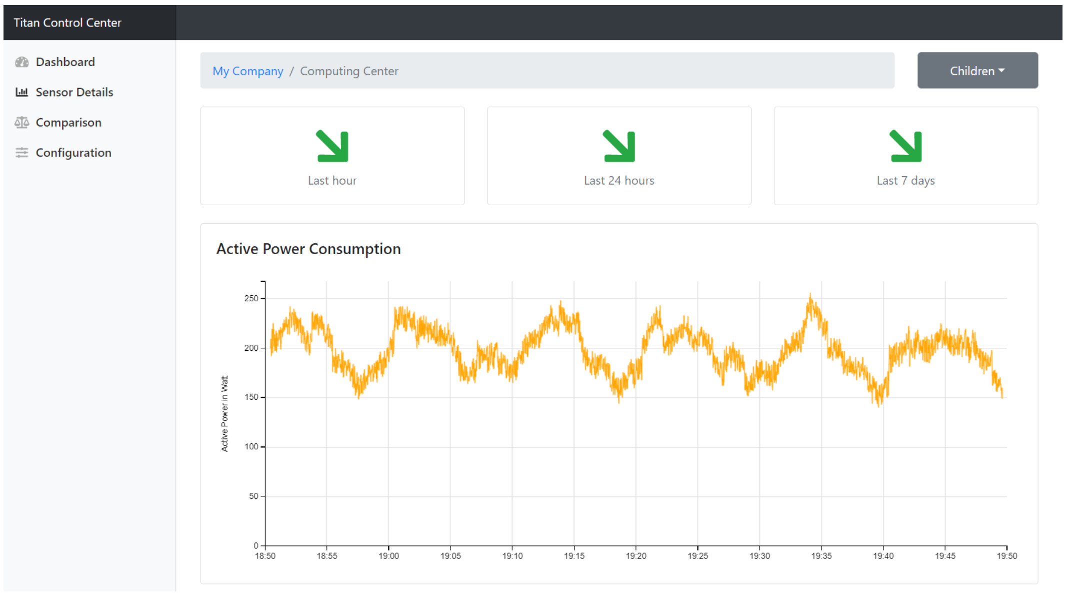


Figure 6.7. Screenshot of the upper part of the sensor details view featuring a navigation bar to select a sensor.

Sensor Details View

The sensor details view is largely the same as the dashboard (see Figure 6.7). However, while the dashboard only shows data for the total consumption, this view visualizes data for a certain, selected (aggregated) sensor. For that reason, it features an additional header bar to navigate through the sensor registry. On the left side, the header bar contains a list of its parents, representing the path within the sensor registry. On the right side, it has a drop-down button opening a list of links to the child sensors. When a non-aggregated sensor is displayed, the composition pie chart will not be displayed as such a sensor is not composed of multiple other sensors.

Comparison View

The comparison view serves to relate different sensors to each other and to compare their monitored values. Similar to the time series chart described above, it shows the monitoring data in a time-value chart. In contrast to that component however, this chart can display data for several different sensors. These are displayed as individual lines having different colors. Therefore, this chart features an additional legend showing the mapping of colors to sensors. A screenshot of it is given in Figure 6.8.

Apart from displaying various sensors in one chart, this view also allows for several charts above each other. That is often useful since this way data sets of different sizes can be

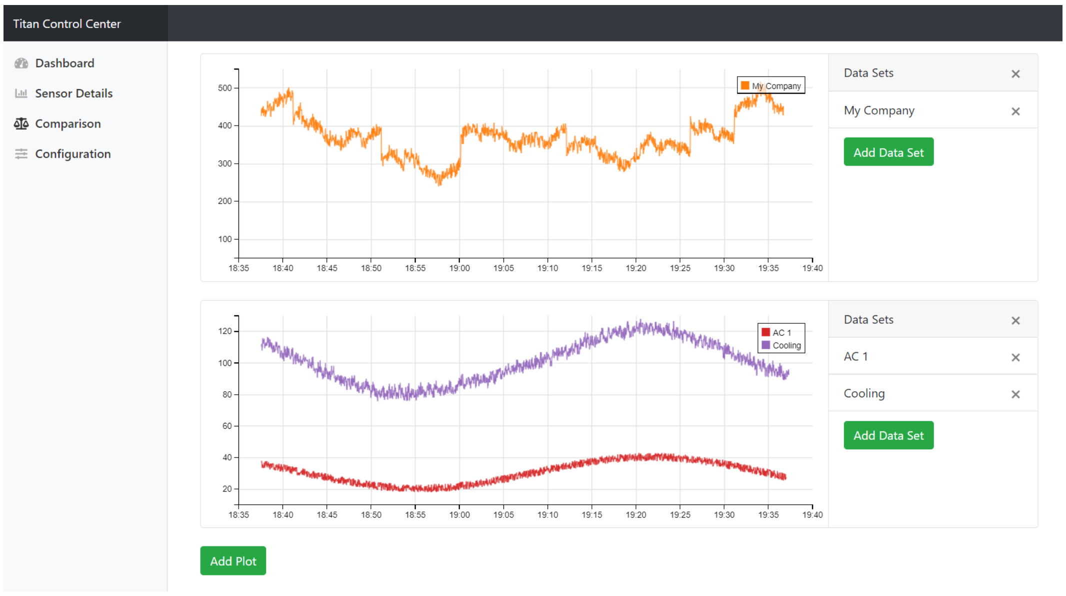


Figure 6.8. Screenshot of the comparison view showing two charts, one of which displays values for one sensor and the other for two.

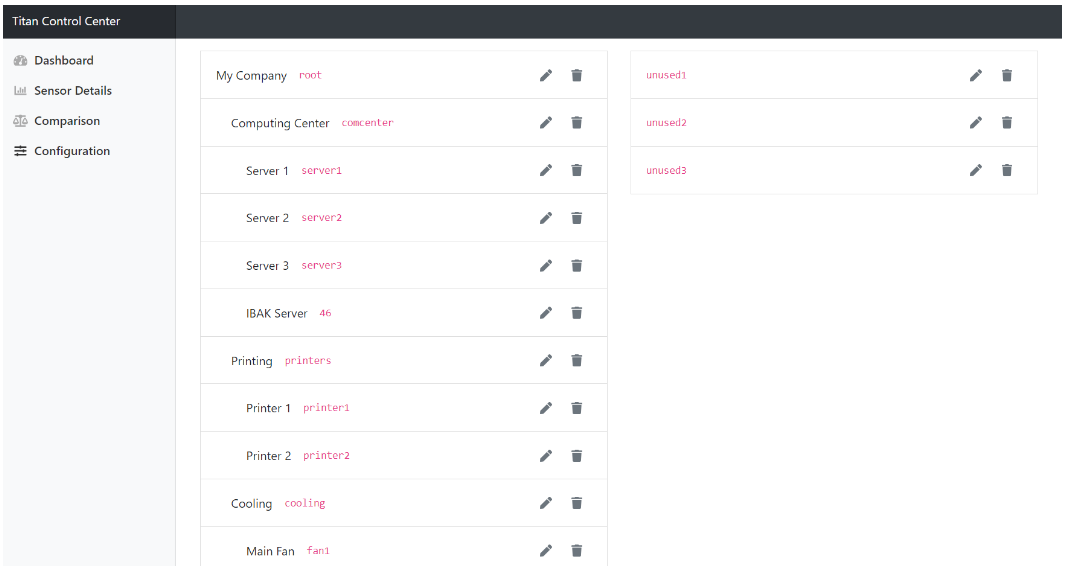
compared. For example, when comparing two sensors, one of which having a significantly higher value than the other: In that case, the scaling of the vertical axis is induced by the larger sensor in order to show all data points. However, that scaling would cause the other sensor to not showing details that clear anymore. Two charts with individual scaling would overcome this issue. In our implementation, this can be arbitrarily combined so that multiple charts can be compared, which itself can contain multiple data sets.

For this view, we implemented that multiple charts can be synchronized with each other. That means, if the displayed time interval in one chart is zoomed or shifted, the other charts are transformed as well. Whereas CanvasPlot enables this for multiple charts with each having only one dataset, we extend that in order to support synchronizing charts with multiple datasets.

Within this view, charts and sensor data sets can be dynamically added or removed. When adding a sensor, it can either be selected via a hierarchically drop-down list or via a search input field.

Configuration View

The configuration view functions as a graphical user interface for the Configuration microservice. It shows the sensor registry as it is currently configured. Figure 6.9 gives screenshot of this view. Sensors can be deleted in order to not longer be considered for



My Company `root`
Computing Center `comcenter`
Server 1 `server1`
Server 2 `server2`
Server 3 `server3`
IBAK Server `46`
Printing `printers`
Printer 1 `printer1`
Printer 2 `printer2`
Cooling `cooling`
Main Fan `fan1`

`unused1`
`unused2`
`unused3`

Figure 6.9. Screenshot of the configuration view, which allows to configure the sensor registry.

aggregation and visualization and the assigned name can be changed. Moreover, a sensor can be moved via drag-and-drop within the sensor registry to assign it as a child sensor of another sensor. When clicking the save button, the newly configured registry is transmitted to the configuration service and replaces the current one.

Part II

Evaluation

Chapter 7

Evaluation

The third goal of this work (see Section 1.2.3) defines a comprehensive evaluation of our approach. This covers its feasibility as well as its scalability. In the following sections, we first present a utility program to simulate sensors (Section 7.1) and describe the hardware and software environment we conduct the evaluation in (Section 7.2). Afterwards, we describe both evaluations in detail. The evaluated feasibility is presented in Section 7.3 and the scalability evaluation in Section 7.4. For each, the structure is as follows: First, we describe what we are going to evaluate and how we are going to achieve that. Then, we describe the results and assess them. Finally, we remark some threats to validity.

7.1 Sensor Simulation Tool

In the context of our evaluations, we developed a tool that artificially generates monitoring data. Besides monitoring real sensors, we therefore have another means to evaluate our approach. This tool allows for simulating a large number of sensors without actually operating devices or machines. That is faster, more flexible, and also cheaper as no devices have to be purchased, installed, or maintained. Furthermore, we can specifically simulate scenarios that would be hard to generate on real devices. For example, such a scenario could be a periodically increasing and decreasing load with precisely defined parameters.

This tool simulates a Raritan PDU as it is described in Section 6.3.2. In a flexible and generic manner, it creates power consumption measurements in real-time and pushes them to a configurable URI. In this context, real-time means, measurements are created and sent out at the time the measurement is supposed to occur, i.e., the time specified in the record. Simulated sensors generate measurements at a configurable, fixed time interval, for instance, every second. Their measured value can change in course of time and is defined by a function $f : \mathbb{N} \to \mathbb{N}$. This function gives the simulated value in Watt for the amount of milliseconds since simulation start. We designed a function builder (*builder pattern* [Gamma et al. 1995]) that enables creating and combining functions generically.

The simulation tool allows to create an arbitrary amount of sensors. For each simulated sensor, we can define its behavior individually. Internally, it creates a thread pool [Mattson et al. 2004] to dispatch records as simultaneously as possible.

Table 7.1. Hardware and software configuration of our evaluation cluster.

	Cloud Controller	Cloud Nodes
Number of CPUs	1	2
CPU	Intel Xeon E5-2609	Intel Xeon CPU E5-2650
Clock Frequency	2.40 GHz	2.80 GHz
Number of Cores	4	16
Number of Threads	4	32
RAM	32 GB	128 GB
OS	Debian GNU/Linux 9 (stretch)	
Kernel Version	4.9.0 – 6 – amd64	
Kubernetes Version	1.9.4	

7.2 Evaluation Environment

In our evaluations, we focus on a centralized, cloud computing deployment of our approach. However, we expect that the results can also be transferred to a decentralized deployment with edge components.

We deploy the components of our implementation that run on the server-side in the private cloud of *Kiel University's Software Performance Engineering Lab*[1]. The deployment corresponds to the deployment architecture described in Section 5.3 and it is run by a Kubernetes cluster. The private cloud consists of one controller and four worker nodes on which the individual components are deployed. All worker nodes are equipped with the same hardware and software configuration as described in Table 7.1.

The visualization runs in the web browser of the user. We evaluated it on a laptop featuring 16 GB of RAM and an *Intel Core i7-7500* CPU with 2 Cores, 4 Threads, and 2.70 GHz clock frequency. The system is operated by *Microsoft Windows 10 Pro* and as a web browser we use *Google Chrome Version 67*.

7.3 Feasibility Evaluation

For the realization of our approach, we implemented a prototype of our architecture. In the following, we evaluate whether our approach is feasible by evaluating its most significant parts. For this purpose, we present various scenarios, each of which should evaluate partial aspects of the implementation. Section 7.3.1 describes these scenarios and Section 7.3.2 presents their results.

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

7.3.1 Methodology

In the following, we describe how the individual parts of our approach are going to be evaluated. As the monitoring of devices' power consumption is performed by external sensors, we do not include this in the evaluation. However, we evaluate the data integration of those sensors using the Raritan Record Bridge microservice as an example. For the data analysis, we evaluate the implementations of aggregation, trend computing and histogram generation. Furthermore, the storage of data belongs to this category. To evaluate the visualization, we consider its four major components: time series chart, composition pie chart, trend arrow, and histogram.

In order to test all those components, we developed four scenarios that execute several different implementation parts. Together they cover the most essential parts of our implementation and, thus, each part of our approach is tested by one or more scenarios. In the following, we describe the individual scenarios in detail and explain, which implementations they test. Figure 7.1 visualizes that.

Scenario 1 This scenario serves for evaluating whether real sensors located in an industrial environment can be integrated into our approach. That encompasses the entire process: Integration of data via the Record Bridge, processing them in terms of writing into the database, and the visualization.

We conduct this using a Raritan PDU as described in Section 6.3.2 that is installed at IBAK. There, a computer is power supplied by one of the PDU outlets. We configured the PDU via its web user interface to record the active power of that outlet every second

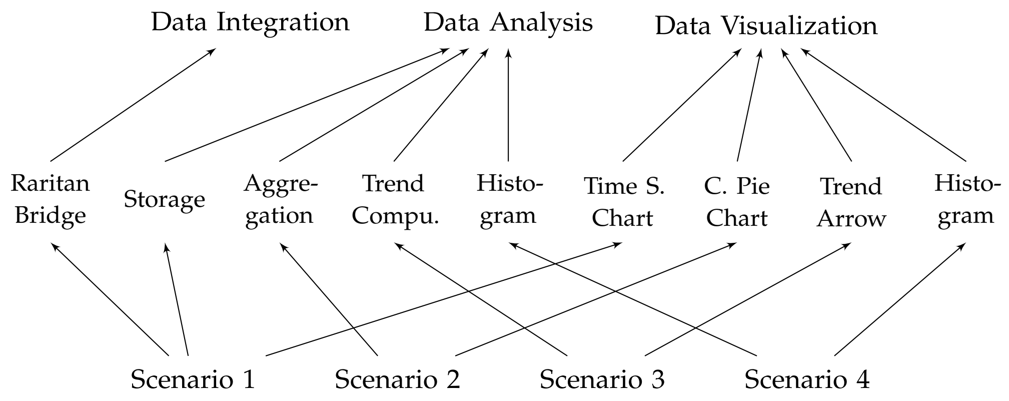


Figure 7.1. Feasibility evaluation plan. The elements in the upper row are individual parts of our approach. The middle row contains our essential implementations. In the bottom row, we list the scenarios we use for evaluation. Arrows between the bottom and the middle row denote which implementations are evaluated by which scenario. Arrows from the middle to the top row show which implementations belong to which part of our approach.

and push the measurements to an instance of the Raritan Record Bridge every minute. The Record Bridge is deployed at Kiel University as described above along with the other components.

As the evaluation infrastructure is protected by a firewall and can only be accessed within a local network, we installed a reverse proxy server within that local network. This reverse proxy is only accessible from devices located at IBAK and forwards sensor push messages to the corresponding bridge. Other messages are rejected. This way, we ensure, firstly, that only sensors from the allowed destination can push data and, secondly, devices from that destination cannot access other services. Apart from that, the reverse proxy also logs every incoming request in order to later verify the data integration process.

Scenario 2 In this scenario, we evaluate the aggregation within our approach. For this purpose, we simulate a simple environment consisting of four devices: a light, a fan, a server, and an air conditioning (AC) system. The simulation tool creates measurements of constant values for each sensor and pushes them to the Record Bridge. The measurement value as well as the interval in which the measurements are simulated is different for each sensor (listed in Table 7.2).

In the corresponding sensor registry for this scenario, the fan and the air conditioning system are grouped as an aggregated sensor, called *cooling*. Furthermore, this aggregated sensor is grouped together with the server and the light to a second aggregated sensor *environment*.

Table 7.2. Simulated sensors for Scenario 2 with their constant value and recording interval.

Device	Value	Interval
Light	10 W	10 000 ms
Fan	20 W	500 ms
Server	30 W	1 000 ms
AC system	40 W	30 000 ms

Scenario 3 In this scenario, we evaluate the trend arrow visualization as well as its calculation. We simulate one sensor that supplies data for slightly more than one hour. The simulation tool creates measurements based on the following function:

$$f(x) = \frac{x}{120\,000} + 100 + \text{noise}(5)$$

It generates values of about 100 W in the beginning and increases the values to about 130 W. The noise function creates random slight divergences of up to 5 W up or down. The simulation tool creates measurements every 10 seconds so that in total about 360 record are created. To evaluate the visualization, we use a smoothing value of 10. That means, the average of the first ten and the last ten values is determined to compute the trend. These are the first and the last 100 seconds.

Scenario 4 This scenario serves for evaluating the Histogram. This covers its computation in the History service as well as its visualization. For this purpose, we simulate again one sensor that creates measurements for one hour every three seconds. Thus, it creates 1200 values in total. The following function defines the values of measurements:

$$f(x) = -80 \cdot \sin\left(\frac{x\pi}{3\,600\,000}\right) + 100$$

When starting the simulation, the measurements have high values but they decrease sharply. The decline becomes weaker and after half an hour the value remains constant. Afterwards, it rises again. First only slowly but then increasingly faster. By the choice of this function, we expect a distribution with many counts for the low values but only a few for higher ones. For the evaluation of this scenario, we set the number of buckets for the histogram to eight.

7.3.2 Results and Discussion

In the following, we describe and discuss the results of each test scenario individually.

Scenario 1 We carried out the evaluation over a period of two weeks and were able to record data throughout the entire test period. The screenshot in Figure 7.2 shows an extract of the data of about one hour. The measured consumption is mostly constant with a value

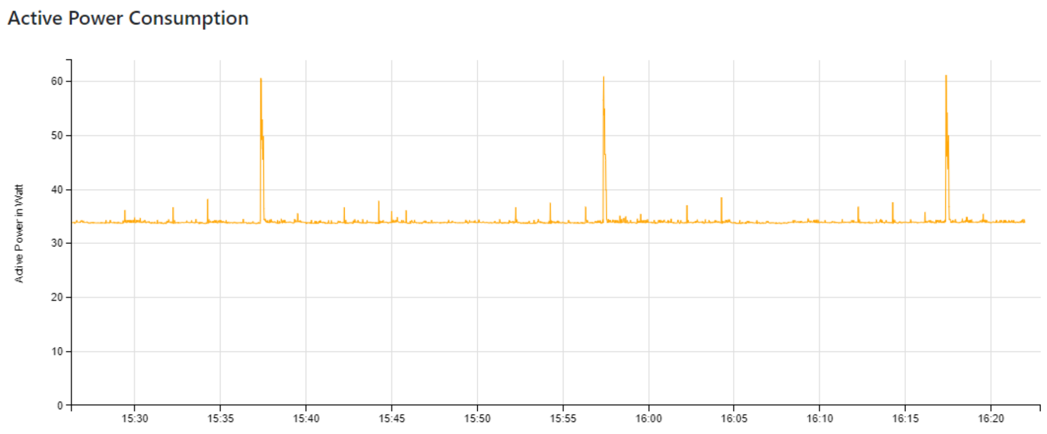


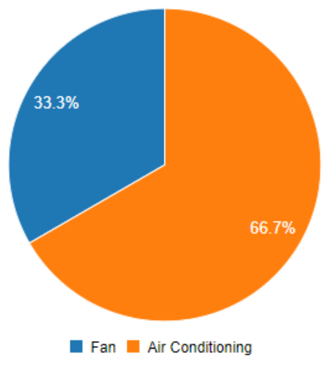
Figure 7.2. Screenshot of the time series chart component for evaluation scenario 1.

of about 33 Watt but shows significant outliers of about 60 Watt. Randomly selected log messages produced by the reverse proxy show also values of about 33 Watt.

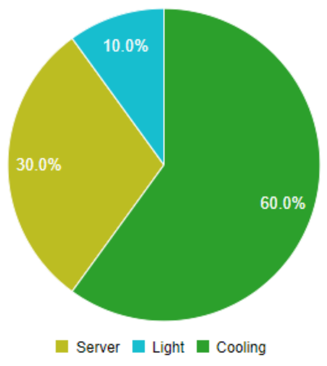
We conclude that the integration of the Raritan PDU is successfully implemented. In particular, we conclude that the transformation of data by the Raritan Record Bridge service works as desired. Also the storage of data was successful as we notice that records are stored with timestamps at intervals of one second. Furthermore, the visualization corresponds to the data returned by the History service's API.

Scenario 2 Figure 7.3 shows screenshots of the composition pie chart components of both aggregated sensors. For the cooling aggregated sensor, the chart shows that it consists of the two child sensors. The fan child sensor is responsible for 33.3% of the cooling consumption and the air conditioning for 66.7%. Furthermore, we request the REST API of the History microservice for the last value before finishing the evaluation. It provides a summed value of 60 Watt. The pie chart for the overall environment aggregated sensor shows the following split: 60% cooling, 30% server, and 10% light. A REST request to the History microservice returns 100 Watt as its last value.

The sum of the constant values for the cooling sensors corresponds to the summed value of the aggregated cooling sensor. Also the summed value of the aggregated environment sensor corresponds to the sum of all its four child sensors. The pie chart component displays the percentage shares displayed of both sensor groups correctly. In particular, the percentage share of the aggregated cooling sensors is displayed correctly in the chart for the overall environment. We conclude that computation of aggregated values is correct. Furthermore, we are able to verify that the aggregation also works if data measurements occur at different points in time.



(a) Aggregated sensor for cooling.



(b) Aggregated sensor for the overall environment.

Figure 7.3. Screenshots of the composition pie chart components of aggregated sensors for evaluation scenario 2.

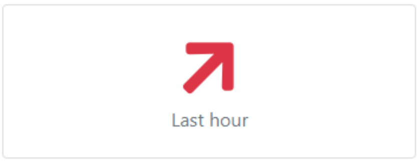


Figure 7.4. Screenshot of the trend arrow visualization component for evaluation scenario 3.

Scenario 3 After execution of this scenario, the trend arrow visualization component for the last hour shows a red arrow pointing up right. Figure 7.4 shows a screenshot of it. Furthermore, a request to the History service returns a trend value of rounded 1.3021.

The translation from the numeric trend value into the arrow graphic works as desired. Furthermore, the trend value calculation is correct. The generated consumption value started at about 100 Watt in the beginning of this evaluation scenario and increased by 30% to about 130 Watt. This matches the calculated value of 1.3021. Minor deviations result from the applied noise function.

Scenario 4 After carrying out this evaluation scenario, the histogram component shows the chart that is displayed in the screenshot of Figure 7.5. As described in Section 6.6.1, it splits the amount of records into buckets and shows the number of records per bucket. The diagram shows eight bars for groups that represent ascending ordered intervals of

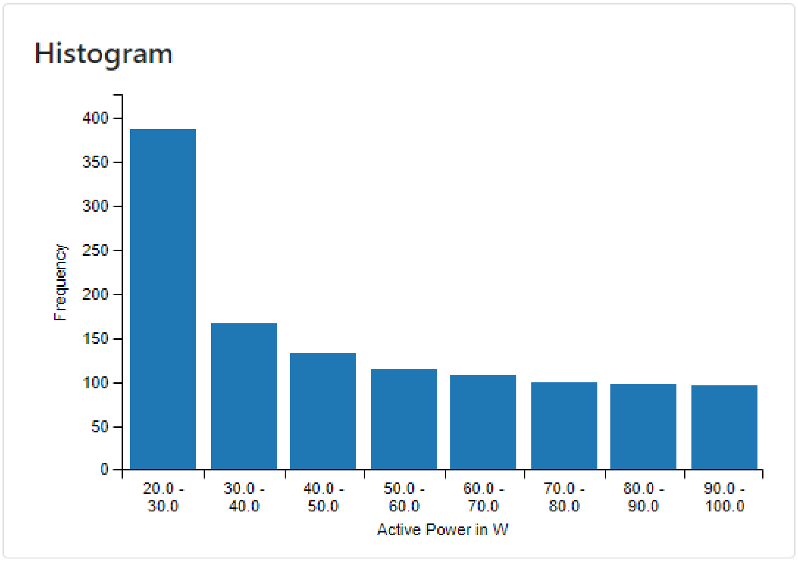


Figure 7.5. Screenshot of the histogram visualization component for evaluation scenario 4.

10 Watt. The first interval starts with a value of 20 Watt and last interval ends with a value of 100 Watt. The amount of records per interval decreases from the first to the last one. Thus, the significantly largest bar is the one for the interval 20–30 Watt showing a value between 350 and 400 records. The frequencies decrease for the following intervals but the differences between predecessor and successor also decrease. The last and smallest bar represents the interval 90–100 Watt and shows a frequency of about 100.

The number of records in the lower intervals is significantly higher than in the upper intervals. That corresponds to what was intended by the scenario and was expected. Furthermore, we request the History service's REST API and are able to verify that the displayed bars correspond to the calculate values. We conclude that the division of records into buckets as well as its visualization work as intended.

7.3.3 Threats to Validity

Our approach features several analyses and visualizations that may supply a variety of different results. As an evaluation of all parts each with various scenarios would exceed the scope of this work, we restricted the evaluation to the most significant ones.

We evaluated our implemented prototype only in one hardware and software environment. While we expect this infrastructure to be similar to a typical one used in practice, we cannot conclude that our approach behaves there the same. An evaluation on different systems would increase the validity.

For the scenarios 2, 3, and 4, we only evaluated simulated sensor data. To improve the validity of these evaluations, we need real devices to produce such data. Furthermore, we integrated only one sensor. In real production environments, however, a multitude of devices as machines is located.

7.4 Scalability Evaluation

The architecture of our approach as well as its implementation is designed in respect of scalability. In the following, we evaluate whether this was effective and our approach is able to monitor arbitrary large production environments. Large in this context means that it operates a high number of sensors that produce data with high frequency so that the total amount of data gets large.

7.4.1 Methodology

The general concept of this evaluation is to simulate a certain amount of sensors that create monitoring data with a certain frequency. For different deployment set-ups, we then evaluate whether they can handle that load.

The actual evaluation is conducted by generating monitoring data and sending it to Record Bridge instances. They forward it and the data passes through the single components of our architecture. We verify that by querying the database of the History microservice. For the whole evaluation, we omit the visualization as it is running in a web browser outside of the other deployment. Therefore, we assume that it does not have any influence on the entire system.

Deployment

As we expect the Record Bridge and the History microservice to be the most limiting parts of our architecture, we restrict the evaluation to explore their scalability. The Configuration service is only requested seldom and components from external sources such as Kafka or Cassandra are known to be scalable.

The evaluation's deployment consists of three Cassandra instances, three Kafka instances coordinated by one ZooKeeper instance, and one instance of the Configuration microservice along with one Redis instance as its database. For the History and Record Bridge, we evaluate different numbers of instances. We deploy no instance of the Visualization's backend as we do not evaluate the visualization for the reasons described above.

Although we have a huge amount of CPU cores available that are supposed to facilitate a high degree of parallelism, the limited number of machines could affect the validity of this evaluation. Kubernetes pods can allocate multiple cores and exploit parallelism already on application level. However, that would result in a situation where one instance is able to handle the same load as several instances sharing the same hardware. The consequence would be that no scaling effect could be seen, although one would actually be possible. Thus, we have to ensure that our approach achieves scalability on a deployment and not just on an implementation level. Kubernetes enables us to specify resource constraints for individual pods. We limit both the History and the Record Bridge service to only use one quarter of a core.

Data Acquiring

In addition to the components of our approach, we deploy the sensor simulation tool at our Kubernetes cluster and configure it for different loads. In each case, it simulates 100 sensors. These generate data at intervals of 500, 1000, 2000, and 4000 milliseconds. After three minutes, the simulation tool terminates.

The sensor simulator maintains an internal counter of dispatched data sets. We request this counter every second and store its value. That means, for discrete times at intervals of one second, we know the number of input records.

Moreover, we record the amount of processed measurement data. For this purpose, we request the History service every second for the currently stored quantity of records via its REST API. As for the input data, this gives us the number of processed records at discrete times.

Analysis

Based on the recorded values, we can calculate the throughput at discrete time intervals for the input data as well as for the stored records.

In order to obtain meaningful results, we consider the first minute of the execution as warm-up phase. Separating the warm-up phase is necessary as components are starting up and reconfiguring themselves. Especially the History service creates database tables when receiving the first records and the Kafka Streams parts are distributing tasks among each other. During that time, records cannot be processed that efficiently. Moreover, also low-level aspects can affect the performance on start-up. For example, the Java Virtual Machine which runs the microservices has a warm-up phase during which it performs many compiler optimizations [Horký et al. 2015]. For the following analysis, we only consider the throughput after the first minute.

For the input as well as for the stored data, we calculate the average throughput over all time intervals. We assume that the average is a reasonable measure since intervals with a low throughput compensate for intervals with a high throughput. This is due to the fact that Kafka queues records and if they are not processed in one interval, they can be processed in another.

Afterwards, we compare the average input throughput with the average processed throughput. Only if the output throughput is at least as large as the input throughput, the evaluated deployment is sufficient for the tested load. Otherwise this would mean, more records are created than that can be processed. Note: It may seem impossible at first, that the processing throughput is higher than the input throughput. However, if less records are processed in the warm-up phase, this delay may be caught up. Then, more records are processed than produced. However, it can also be the other way around: Single records may be processed delayed after finishing the evaluation. Therefore, we additionally compare the output throughput with 90% of the input throughput assuming that this delay can be tolerated.

Based on these data, we can determine the smallest possible deployment set-up that is able to handle that load. If the load and the number of necessary instance are proportional, we can consider our approach as scalable.

7.4.2 Results and Discussion

In most of the tested scenarios we are able to acquire data for the input as well as for the output throughput. However, we noticed by inspecting the log messages that sometimes the simulation tool fails when it tries to create too many threads. This only occurs occasionally in different scenarios so that this should not impair the evaluation.

Whereas the input throughput remains largely constant over multiple runs, the output throughput diverges heavily. The deviations are so wide that it is difficult to make general statements about the throughput. Analyzing the executions with a particularly low throughput, we found out that some instances of the History microservice sometimes do not process any data but instead remain idle. However, there would be enough data for processing and at some point they start working but then stop again. We are not able to obtain any kind of error messages, neither exceptions from Java nor log messages from the Kafka streams framework or the Kafka and ZooKeeper instances that allow to locate the error. Furthermore, we discovered that restarting the instances several times often solves the problem. When generating records with relatively high throughput in comparison to the number of processing Record Bridges, we are able to obtain error messages. They state the internal queues of the Record Bridges are full and no further messages can be stored. That corresponds to the expected behavior and does not explain the other issue. Apart from that, the Record Bridge services work as intended.

In the course of this work, we were not able to locate the reason of the diverging throughputs. To narrow the field of possible causes, we modified the evaluation set-up in various ways. One of those was to provide more resources per instance. Moreover, we tested a modified implementation that not executed the aggregation but solely the data storage. We also used different numbers of instances for Kafka and ZooKeeper. The number of Kafka topic partitions must be at least as large as the number of History service instances. Therefore, we tested different numbers of partitions ranging from 10 up to 100. Instead of 100 sensors that produce data, we also tested a set of 10 sensors that generate data with 10 times higher frequencies.

There are still several possibilities that could cause the issue. It is very likely that the problem is related to the coordination between Kafka Stream instances. We can currently not ensure that we configured Kafka or Kafka streams appropriately and, for instance, records are not buffered. If the configuration of our Kafka Streams instances is unsuitable, we have to adjust the implementation. If it is the configuration of Kafka or ZooKeeper itself, instead, the deployment has do be adjusted. We suspect an error in the aggregation logic to be unlikely as the results do not differ if the records are only forwarded to the database.

Based on the recorded evaluation results, we are not able to demonstrate that our approach is scalable. However, our results do not refute it either as it may still be possible that it scales just unreliably. To determine whether our approach is scalable apart from those reliability issues, we would have to compare only those executions we can assume for to behave as desired. Therefore, we compare the highest measured throughputs of all runs as those runs are most likely to be performed without the issue. We then evaluate if the maximum results show scalability characteristics. Note, comparing the averages of multiple runs is not reasonable as low values of unsuccessful executions influence surpassingly the average.

A selection of those results is presented in Table 7.3. In this table, the first four columns specify the configuration of a scenario. Δ_m corresponds to the measurement interval in milliseconds, $\#_B$ is the number of deployed Record Bridge instances and $\#_H$ the number of deployed History service instances. The column $\#_B + \#_H$ shows the sum of both. The next four columns list the results for that scenario. T_{in} is the average throughput for sending records, $\#_{in}$ is the amount of totally sent records, T_{out} is the average throughput

Table 7.3. Selected results of the scalability evaluation.

Δ_m	$\#_B$	$\#_H$	$\#_B + \#_H$	T_{in}	$\#_{in}$	T_{out}	$\#_{out}$	$T_{in} \leqslant T_{out}$	$0.9 \cdot T_{in} \leqslant T_{out}$
500	8	8	16	200.0	36099	188.9	20879	false	true
1000	2	8	10	100.0	18100	103.0	13755	true	true
1000	4	2	6	100.0	18100	92.9	10159	false	true
2000	1	2	3	50.4	9100	58.9	8311	true	true
2000	1	4	5	50.4	9100	60.0	7610	true	true
2000	1	8	9	50.4	9100	56.0	8393	true	true
2000	2	2	4	50.4	9100	56.2	8938	true	true
2000	2	4	6	50.4	9100	55.4	8896	true	true
2000	2	8	10	50.4	9100	56.8	8844	true	true
4000	1	1	2	25.3	4561	22.9	3439	false	true
4000	1	2	3	25.6	4600	27.7	4457	true	true
4000	1	4	5	25.6	4600	26.3	4484	true	true
4000	1	8	9	25.6	4600	24.9	4500	false	true
4000	2	1	3	25.6	4600	24.7	3902	false	true
4000	2	2	4	25.6	4600	26.0	4335	true	true
4000	2	4	6	25.6	4600	25.0	4500	false	true
4000	2	8	10	25.6	4600	25.3	4500	false	true

for processing records, and $\#_{out}$ is the amount of totally processed records. The last two columns show if $T_{in} \leqslant T_{out}$ and $0.9 \cdot T_{in} \leqslant T_{out}$ hold.

As a scenario configuration, we consider the triple consisting of Δ_m, $\#_B$, and $\#_H$. For the reasons described above, we list the execution results with the highest value for T_{out} for each configuration. Moreover, we only display those results for which $0.9 \cdot T_{in} \leqslant T_{out}$ is true. Additionally, we do not list scenarios where Δ_m is 2000 or 4000 and the number of instances is high. A complete table of all maximal results is given in Appendix B.

For each tested configuration, we determine the amount of produced measurement records per second. As we simulated 100 sensors in each configuration, the number of measurements per second equals $\frac{1000}{\Delta_m} \cdot 100$. Figure 7.6 shows the required number of instances for each evaluated input throughput. For this purpose, we determine the lowest number of Record Bridges, History services, and the sum of both for which $0.9 \cdot T_{in} \leqslant T_{out}$ is true. We assume that at least those numbers are necessary to handle that load.

We observe that the number of required instances growths with the increase of produced records per second. This applies to the Record Bridge and the History service but also for the sum of both. In particular, the latter is important to include mutual impacts. We are able to process even the highest tested throughput. Between 25 and 100 records per second, the number of required instances grows less than by a factor of 2. For throughputs higher than that, it grows faster than by a factor of 2. We conclude that our approach is able to scale with increasing load. Due to the issues discussed above, however, this does not always work reliably.

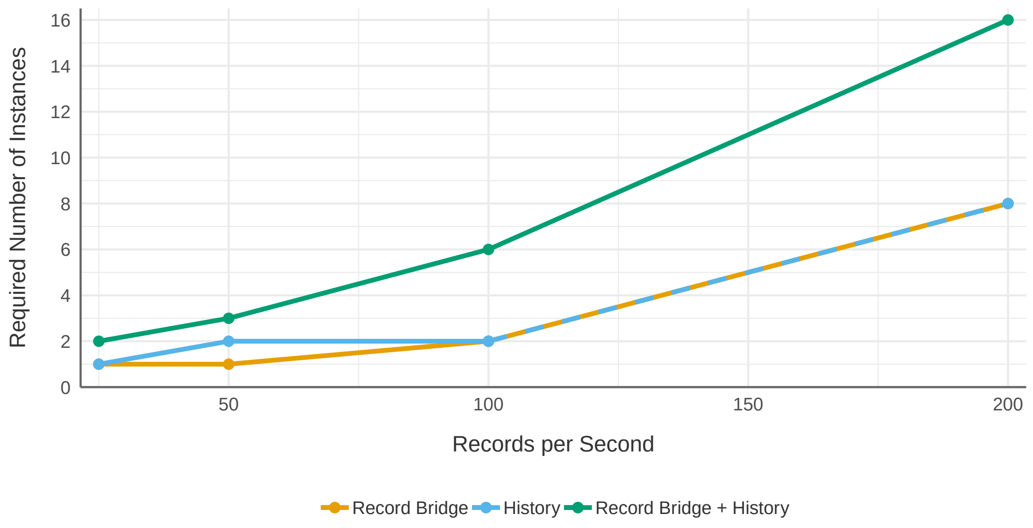


Figure 7.6. Lowest number of instances that is able to process a certain number of measurement records per seconds.

7.4.3 Threats to Validity

As described above, we use the maximum as function to determine an aggregated value of multiple runs. Similar to using the average, this also distorts the results since outliers upwards are particularly considered. We could obtain more conclusive results if we would calculate appropriate quantiles instead. However, since we only perform three runs, the maximum already corresponds to the $Q_{0.67}$ quantile. Executing the scenarios many more times would lead to more meaningful results.

We only evaluate the scalability in one single hardware environment. Moreover, we only use one the deployment set-up of all components apart the Record Bridge and the History service. For instance, we do not evaluate larger or smaller numbers of instances for Cassandra or Kafka and also we do not consider the aggregation part of the History service. Furthermore, we only evaluate certain scales of record sending intervals and amount of sensors. Significantly larger or smaller test scenarios, would increase the transferability to real production environments.

In this evaluation, only one instance simulates measurements. Due to its limited parallelism, the simulation tool does not necessarily behave like a real set of sensors, which measure and transfer data entirely independent of each other. Furthermore, the sensor simulation tool runs on the same hardware as the analyses. We can not necessarily assume that they do not influence each other.

Chapter 8

Related Work

During this work, we only found a few scientifically published approaches for industrial monitoring infrastructures that use scalable cloud computing patterns and technologies. Furthermore, we were not able to find any other approach based on microservices. We conclude that this is a rather new field of research. However, we notice that the field of monitoring production environments is being worked on from the perspective of integrating heterogeneous sensor types as well as from an analysis perspective. Moreover, we suspect that work in this field is also created in industry, which is not or at least not scientifically published. In the following, we present some related work and compare it with our approaches.

Gröger [2018] presents an *Industry 4.0* analytics platform developed at *Bosch*. It is designed to be deployed at more than 270 factories. The platform aims for integrating all types of data that occur in a factory and, thus, enabling optimization and predictive maintenance. Therefore, it integrates machine and sensor data as well as enterprise data. The analytics platform is designed as a Lambda architecture comprising a batch, a speed, and a serving layer. Based on the analyzed data, it provides a dashboard and reporting capabilities, but also more advanced techniques such as simulations or data mining and machine learning. Those are realized as analytical services, build on top of the Lambda architecture. This contrasts our architecture, which is entirely microservice-based. Moreover, in contrast to our approach, the Bosch platform is not exclusively build with open source tools but also with commercial ones. Similar to our approach, it allows for different deployment scenarios, adapted to the respective application environment. Unfortunately, the presented paper lacks a feasibility as well as a scalability evaluation.

A monitoring software for the *Industrial Internet of Things* is presented by Civerchia et al. [2017]. They monitor production environments in order to enable predictive maintenance. That is particularly interesting as predictive maintenance is supposed to be also addressed later in the Titan project and our approach can lay the foundation. In contrast to ours, their work focuses more on the data recording and the necessary hardware. The authors use battery powered sensor devices that communicate via wireless networks and open protocols. Sensors perform computations by themselves and are retrievable via REST interfaces. This requires more progressive technologies than assumed in our approach. The presented software was evaluated for two months in an electrical power plant. The evaluation focuses on the performance of this concrete set-up and on scalability. Whereas our approach is intended to integrate different existing data formats and protocols, their approach relies on standardized ones. Due to the intensive use of Internet of Things practices, it can be considered as distributed system. It is more edge-centric oriented (without explicitly using the term) than our approach.

Pramudianto et al. [2013] present an approach to integrate Internet of Things techniques such as sensors and devices of different types to overcome interoperability. Thus, this approach serves a similar purpose to our Record Bridge. It realizes a middleware as it is known from service-oriented architectures (SOA). Also their realization is similar to our Record Bridge. Whereas we deploy a microservice for each type of sensor, they create a *proxy*. Following the publish-subscribe pattern, this proxy publishes events if sensor data change and other components can subscribe to these events. Also in this respect, their approach corresponds to ours. Whereas we abstract from technologies and protocols of sensors and devices, the authors of this work focus on wireless network sensors. This approach was successfully evaluated by a prototypical test bed for a welding station. It was not evaluated regarding to scalability. However, as the described proxy components are stateless, we expect that at least this part of their approach is scalable.

A microservice-based Internet of Things platform for *Smart Cities* is presented by Krylovskiy et al. [2015]. This approach is similar to ours in that heterogeneous sensor technologies are integrated. As well as our prototype, it is an early-stage approach that serves for, inter alia, evaluating the microservice pattern in this context. The architecture presented in this paper resembles ours in many aspects. The integration of different sensors monitoring systems is realized by microservices. Similar to our Configuration service that manages the configuration of sensor devices, their platform comprises several *middleware* services that fulfill similar tasks. Moreover, their platform provides a service that stores historical data similar to our History service. Another concordance is that both implementations use REST as well as a message broker to let services communicate with each other. However, whereas we apply the open source messaging platform Apache Kafka, they use the open protocol MQTT. The authors choose the microservice pattern as components are developed by members of an interdisciplinary team. Thus, services can be organized independently around business capabilities and the team members can individually select their technologies. Unfortunately, their approach features no evaluation of scalability.

A scalable and furthermore elastic monitoring infrastructure in the domain of application-level software performance is proposed by Fittkau and Hasselbring [2015]. Similar to our approach, it faces the challenge of continuously aggregating data. Therefore, the authors apply a master-worker approach, in which data is aggregated in multiple levels. Monitoring records are sent to different *worker* instances that aggregate the data and push the results to a next level. There again, multiple workers aggregate it, forward it to next level and so on. A central instance, the *master*, is the last level that performs the final aggregation and provides the result to the user. In contrast to our approach, this one is not entirely decentralized, i.e., there is a master. A possible issue with this approach is that having a single master instance can result in a single point of failure. A solution to this could be to not specify an explicit master instance but instead define master and worker tasks. These tasks can then be assigned dynamically to instances. If the instance performing the master task fails, it can be assigned to another instance at runtime. Aggregating records in multiple levels may facilitate a better scalability. In contrast to our approach, it does not require that one instance obtains every single record. The issue with this idea is that statistics such as the median are impossible to be computed as the necessary data is already removed in lower aggregation levels. This approach excludes that already on the architectural level, whereas we only decided not to implement it but it can be added in future work. In an evaluation, the authors were able to analyze 20 million monitoring records per second. The evaluation environment they use is about twice as powerful as the one we use and they do not explicitly restrict the resources per instance. In contrast to their approach, in our evaluation we need to transfer monitoring data via HTTP. Therefore, we expect it to be unlikely to process similar amounts of records.

Part III

Conclusions and Future Work

Chapter 9

Conclusions

In this work, we presented an approach on how a scalable monitoring infrastructure for Industrial DevOps can be realized. Therefore, we first introduced the term Industrial DevOps as an approach how the ideas of DevOps can be transferred to the conventional manufacturing industry to face the challenges of digitalization. To the best of our knowledge, the approach presented in this work is new as we were not able to find similar works realizing such an infrastructure with open source and cloud technologies.

We developed a prototype for this infrastructure. To realize it, we first designed an approach on how data recording, integration, analysis, and visualization can be united. Our prototype focuses on monitoring electrical power consumption in the form of active power. The major function of the analysis is to group and aggregate similar sensor data.

Based on this approach, we designed an architecture following the microservice pattern. Originating in web-based applications, it provides solutions to the requirements for scalability, adaptability, and fault tolerance. We applied the principles of microservice architectures, combined them with techniques for real-time data processing and overcome the constraints of limited available resources in production environments and the high volume of data. The current state of the architecture comprises three microservices as well as a common visualization component.

The aspects we focused on in the architecture design, are also realized in the implementation. As web-based applications face similar challenges as we do, our implementation is also oriented towards them. For instance, we use the highly distributed databases and messaging systems Apache Cassandra and Apache Kafka. Our prototype integrates exemplarily data of a power distribution unit for server racks. It analyzes the data in terms of several aspects and visualizes it in different ways in a single-page application.

We evaluated our approach in a real production environment, where we integrated one sensor measuring the power consumption of a server. The evaluation shows that the data integration and processing functions reliably. We also evaluated the individual analyses and visualization separately and observed that they work as intended. Furthermore, we evaluated how scalable our approach is. We discovered that the potential for scalability exists but it is not always performed reliably. Detailed investigation have shown that this is more likely to be an implementation issue rather than one in the architecture design. We refer to future work, in which the exact reason for the unreliability should be found and solved.

In this work, we furthermore presented approaches and patterns to achieve a scalable processing of big data. In particular, we introduced the microservice architectural pattern and concepts to edge and cloud computing. Moreover, we presented established technologies that are frequently used in the context of scalable cloud applications.

We publish all software, which we developed for the presented prototype, as open source under the the Apache License 2.0 [Apache Software Foundation 2004]. The individual projects are located in public Git repositories. Furthermore, we provide ready-to-use Docker images for the individual software components and an exemplary deployment set-up for Kubernetes. The location for of the individual projects can be found in Appendix A.

Chapter 10

Future Work

The first objective that future work should address is to achieve scalability in a reliable manner. In Section 7.4.2, we already pointed out which causes we could exclude for the unreliability and which points need to be further investigated. Narrowing down the problem promises that it is realistic to find the cause.

In the next step, we can then work on making our approach elastically deployable. Our scalable architecture already lays the foundation for this. The great benefit of an elastic monitoring infrastructure would be that it allows to process the huge amount of sensor data cost-efficient in a cloud environment. In a production environment the volume of data is often subject to even higher fluctuations than, for example, in web applications. In factories, data may not be generated permanently but instead, for instance, only during daytime and weekdays as machines are idle for the rest of the time. An elastic analysis software running in the cloud would allow that only during the time data is recorded, hardware have to be payed.

The monitoring infrastructure presented in this work is only intended to be a prototype. For this reason, we only implemented a subset of the functionality that is intended to be provided later. Future work will address the extension of this approach in different ways.

Especially, this will focus on further analyses and visualizations. Enhanced visualization techniques are, for instance, provided by ExplorViz [Fittkau et al. 2015; 2017]. It is able to work with Kieker monitoring data and, therefore, we expect to adapt it for visualizing our monitoring data. Also enhanced analyses exist in the context of Kieker. For instance, a generic anomaly detection on Kieker data was presented by Henning [2016]. It is realized as a microservice and we expect that it is possible to integrate it into our approach.

Another possibility to extend our approach is to integrate other metrics and production or enterprise data. This would allow for correlating them with each other. This way one can investigate, for instance, how power consumption is related to the amount of produced units. Another promising field of research would be to correlate power consumption with software monitoring data. As this prototype is already able to measure power consumption of servers and Kieker monitors software systems anyway, only the necessary analyses have to be implemented.

Finally, our approach should be brought into accordance with other Titan projects. That covers, for instance, the attachment to the flow approach as described in Chapter 1. Using that, when defining the processes it should be automatically possible to directly integrate monitoring.

Bibliography

[Abbott and Fisher 2009] M. L. Abbott and M. T. Fisher. *The art of scalability: scalable web architecture, processes, and organizations for the modern enterprise*. 1st. Addison-Wesley Professional, 2009. (Cited on page 6)

[Apache Software Foundation 2004] Apache Software Foundation. *Apache license 2.0*. Accessed: 2018-07-29. 2004. URL: `https://www.apache.org/licenses`. (Cited on page 82)

[Apache Software Foundation 2017] Apache Software Foundation. *Kafka*. Accessed: 2018-07-29. 2017. URL: `https://kafka.apache.org`. (Cited on page 16)

[Armbrust et al. 2010] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM* 53.4 (Apr. 2010), pages 50–58. (Cited on page 11)

[Avizienis et al. 2004] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing* 1.1 (2004), pages 11–33. (Cited on page 28)

[Bernstein 2014] D. Bernstein. Containers and cloud: from lxc to docker to kubernetes. *IEEE Cloud Computing* 1.3 (Sept. 2014), pages 81–84. (Cited on page 18)

[Bird and Wadler 1988] R. Bird and P. Wadler. *An introduction to functional programming*. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1988. (Cited on page 41)

[Bonomi et al. 2012] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. Helsinki, Finland: ACM, 2012, pages 13–16. (Cited on page 12)

[Bostock et al. 2011] M. Bostock, V. Ogievetsky, and J. Heer. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17.12 (2011), pages 2301–2309. (Cited on page 52)

[Chen and Zhang 2014] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: a survey on big data. *Information Sciences* 275 (2014), pages 314–347. (Cited on page 10)

[Chen et al. 2014] M. Chen, S. Mao, and Y. Liu. Big data: a survey. *Mob. Netw. Appl.* 19.2 (Apr. 2014), pages 171–209. (Cited on page 10)

[Civerchia et al. 2017] F. Civerchia, S. Bocchino, C. Salvadori, E. Rossi, L. Maggiani, and M. Petracca. Industrial internet of things monitoring solution for advanced predictive maintenance applications. *Journal of Industrial Information Integration* 7 (2017). Enterprise modelling and system integration for smart manufacturing, pages 4–12. (Cited on page 75)

[Claps et al. 2015] G. G. Claps, R. B. Svensson, and A. Aurum. On the journey to continuous deployment: technical and social challenges along the way. *Information and Software Technology* 57 (2015), pages 21–31. (Cited on page 1)

[Cloud Native Computing Foundation 2017] Cloud Native Computing Foundation. *Kubernetes*. Accessed: 2018-07-29. 2017. URL: `https://kubernetes.io`. (Cited on page 18)

[Docker, Inc. 2017] Docker, Inc. *Docker*. Accessed: 2018-07-29. 2017. URL: `https://www.docker.com`. (Cited on page 18)

[Fielding 2000] R. T. Fielding. Architectural styles and the design of network-based software architectures. PhD thesis. 2000. (Cited on pages 8, 30, 37)

[Fittkau and Hasselbring 2015] F. Fittkau and W. Hasselbring. Elastic application-level monitoring for large software landscapes in the cloud. In: *Service Oriented and Cloud Computing*. Edited by S. Dustdar, F. Leymann, and M. Villari. Volume 9306. Lecture Notes in Computer Science. Springer-Verlag, Sept. 2015, pages 80–94. (Cited on page 76)

[Fittkau et al. 2017] F. Fittkau, A. Krause, and W. Hasselbring. Software landscape and application visualization for system comprehension with explorviz. *Information and Software Technology* 87 (July 2017), pages 259–277. (Cited on page 83)

[Fittkau et al. 2015] F. Fittkau, S. Roth, and W. Hasselbring. Explorviz: visual runtime behavior analysis of enterprise application landscapes. In: *23rd European Conference on Information Systems (ECIS 2015)*. Mai 2015. (Cited on page 83)

[Fowler 2010] M. Fowler. *Domain specific languages*. 1st. Addison-Wesley Professional, 2010. (Cited on page 41)

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. (Cited on pages 38, 47, 61)

[Garcia Lopez et al. 2015] P. Garcia Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. Edge-centric computing: vision and challenges. *SIGCOMM Comput. Commun. Rev.* 45.5 (Sept. 2015), pages 37–42. (Cited on pages 11, 12)

[Goldschmidt et al. 2018] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner. Container-based architecture for flexible industrial control applications. *Journal of Systems Architecture* 84 (2018), pages 28–36. (Cited on page 35)

[Gröger 2018] C. Gröger. Building an industry 4.0 analytics platform. *Datenbank-Spektrum* 18.1 (Mar. 2018), pages 5–14. (Cited on page 75)

[Hasselbring 2016] W. Hasselbring. Microservices for scalability: keynote talk abstract. In: *International Conference on Performance Engineering (ICPE 2016)*. Edited by A. Avritzer and A. Iosup. ACM, Mar. 2016, pages 133–134. (Cited on page 8)

[Hasselbring 2018] W. Hasselbring. Software architecture: past, present, future. In: *The Essence of Software Engineering*. Edited by V. Gruhn and R. Striemer. Cham: Springer International Publishing, 2018, pages 169–184. (Cited on page 7)

[Hasselbring and Steinacker 2017] W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In: *Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, Apr. 2017, pages 243–246. (Cited on pages 7, 18, 30)

[Henning 2016] S. Henning. Visualization of performance anomalies with kieker. Bachelor's Thesis. Kiel University, Sept. 2016. (Cited on pages 16, 83)

[Herbst et al. 2013] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: what it is, and what it is not. In: *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. San Jose, CA: USENIX, 2013, pages 23–27. (Cited on page 7)

[Hohpe and Woolf 2003] G. Hohpe and B. Woolf. *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. (Cited on page 16)

[Horký et al. 2015] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. Dos and don'ts of conducting performance measurements in java. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: ACM, 2015, pages 337–340. (Cited on page 70)

[International Electrotechnical Commission 2018] International Electrotechnical Commission. *IEC 60050*. Accessed: 2018-07-03. 2018. URL: `http://www.electropedia.org/iev/iev.nsf/display?openform&ievref=131-11-42`. (Cited on page 24)

[Jeschke et al. 2017] S. Jeschke, C. Brecher, T. Meisen, D. Özdemir, and T. Eschert. Industrial internet of things and cyber manufacturing systems. In: *Industrial Internet of Things: Cybermanufacturing Systems*. Edited by S. Jeschke, C. Brecher, H. Song, and D. B. Rawat. Cham: Springer International Publishing, 2017, pages 3–19. (Cited on page 24)

[Johanson et al. 2016] A. Johanson, S. Flögel, C. Dullo, and W. Hasselbring. Oceantea: exploring ocean-derived climate data using microservices. In: *Proceedings of the Sixth International Workshop on Climate Informatics (CI 2016)*. NCAR Technical Note NCAR/TN. Sept. 2016, pages 25–28. (Cited on page 51)

[Jung and Wulf 2016] R. Jung and C. Wulf. Advanced typing for the kieker instrumentation languages. In: *Symposium on Software Performance 2016*. Nov. 2016. (Cited on page 15)

[Knoche 2017] H. Knoche. Refactoring kieker's i/o infrastructure to improve scalability and extensibility. *Softwaretechnik-Trends* 37.3 (Nov. 2017), pages 17–19. (Cited on pages 15, 39)

[Kreps 2014] J. Kreps. *Questioning the lambda architecture*. Accessed: 2018-07-29. 2014. URL: `https://www.oreilly.com/ideas/questioning-the-lambda-architecture`. (Cited on page 11)

[Kreps et al. 2011] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: a distributed messaging system for log processing. In: *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece*. 2011. (Cited on page 16)

[Krylovskiy et al. 2015] A. Krylovskiy, M. Jahn, and E. Patti. Designing a smart city internet of things platform with microservice architecture. In: *2015 3rd International Conference on Future Internet of Things and Cloud*. Aug. 2015, pages 25–30. (Cited on page 76)

[Lakshman and Malik 2010] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pages 35–40. (Cited on page 17)

[Lwakatare et al. 2015] L. E. Lwakatare, P. Kuvaja, and M. Oivo. Dimensions of devops. In: *Agile Processes in Software Engineering and Extreme Programming*. Edited by C. Lassenius, T. Dingsøyr, and M. Paasivaara. Cham: Springer International Publishing, 2015, pages 212–217. (Cited on page 1)

[Martin 1965] J. Martin. *Programming real-time computer systems*. Prentice-Hall series in automatic computation. Prentice-Hall, 1965. (Cited on page 27)

[Marz and Warren 2015] N. Marz and J. Warren. *Big data: principles and best practices of scalable realtime data systems*. 1st. Greenwich, CT, USA: Manning Publications Co., 2015. (Cited on page 11)

[Mattson et al. 2004] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. First. Addison-Wesley Professional, 2004. (Cited on pages 43, 61)

[Moebius and Ulrich 2016] A. Moebius and S. Ulrich. Improving kieker's scalability by employing linked read-optimized and write-optimized nosql storage. In: *Symposium on Software Performance 2016 (SSP '16)*. Nov. 2016. (Cited on pages 15, 47)

[Morrison 2010] J. P. Morrison. *Flow-based programming, 2nd edition: a new approach to application development*. Paramount, CA: CreateSpace, 2010. (Cited on page 2)

[Newman 2015] S. Newman. *Building microservices*. 1st. O'Reilly Media, Inc., 2015. (Cited on pages 7, 9)

[Pramudianto et al. 2013] F. Pramudianto, J. Simon, M. Eisenhauer, H. Khaleel, C. Pastrone, and M. Spirito. Prototyping the internet of things for the future factory using a soa-based middleware and reliable wsns. In: *2013 IEEE 18th Conference on Emerging Technologies Factory Automation (ETFA)*. Sept. 2013, pages 1–4. (Cited on page 76)

[Richardson 2018] C. Richardson. *Microservice patterns*. Preprint: 2018-07-26. Manning Publications Company, 2018. (Cited on page 7)

[Shaw 1989] M. Shaw. Larger scale systems require higher-level abstractions. *SIGSOFT Softw. Eng. Notes* (Apr. 1989). (Cited on page 16)

[Shin and Ramanathan 1994] K. G. Shin and P. Ramanathan. Real-time computing: a new discipline of computer science and engineering. *Proceedings of the IEEE* 82.1 (Jan. 1994), pages 6–24. (Cited on page 28)

[Smith and Williams 2002] C. Smith and L. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2002. (Cited on page 5)

[Spence and Lewandowsky 1991] I. Spence and S. Lewandowsky. Displaying proportions and percentages. *Applied Cognitive Psychology* 5.1 (1991), pages 61–77. (Cited on page 52)

[Strubel and Wulf 2016] H. Strubel and C. Wulf. Refactoring kieker's monitoring component to further reduce the runtime overhead. In: *Symposium on Software Performance 2016 (SSP '16)*. Nov. 2016. (Cited on page 47)

[Tanenbaum and Steen 2006] A. S. Tanenbaum and M. v. Steen. *Distributed systems: principles and paradigms (2nd edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 2006. (Cited on page 6)

[Titan Project 2018] Titan Project. *The industrial devops platform for agile process integration and automatision*. Accessed: 2018-07-29. 2018. URL: `https://industrial-devops.org`. (Cited on page 1)

[Urma 2014] R.-G. Urma. Processing data with java se 8 streams, part 1. *Oracle (Java Mag.)* (2014). (Cited on page 41)

[Van Hoorn et al. 2009] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. *Continuous monitoring of software services: Design and application of the Kieker framework*. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009. (Cited on page 15)

[Van Hoorn et al. 2012] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: a framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012, pages 247–248. (Cited on page 15)

[Vogel-Heuser et al. 2014] B. Vogel-Heuser, C. Diedrich, A. Fay, S. Jeschke, S. Kowalewski, M. Wollschlaeger, and P. Göhner. Challenges for Software Engineering in Automation. *Journal of Software Engineering and Applications* 07.May (2014), pages 440–451. (Cited on page 24)

[Wulf et al. 2017] C. Wulf, W. Hasselbring, and J. Ohlemacher. Parallel and generic pipe-and-filter architectures with teetime. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Apr. 2017, pages 290–293. (Cited on page 16)

[You 2018] E. You. *Vue.js*. Accessed: 2018-07-29. 2018. URL: `https://vuejs.org`. (Cited on page 17)

[Youseff et al. 2008] L. Youseff, M. Butrico, and D. D. Silva. Toward a unified ontology of cloud computing. In: *2008 Grid Computing Environments Workshop*. Nov. 2008, pages 1–10. (Cited on page 11)

Appendix A

List of Implementations

History Microservice
Source code: `https://github.com/cau-se/titan-ccp-history`
Docker image: `https://hub.docker.com/r/industrialdevops/titan-ccp-aggregation`

Configuration Microservice
Source code: `https://github.com/cau-se/titan-ccp-configuration`
Docker image: `https://hub.docker.com/r/industrialdevops/titan-ccp-configuration`

Record Bridge Framework
Source code: `https://github.com/cau-se/titan-ccp-record-bridge`

Raritan Record Bridge Microservice
Source code: `https://github.com/cau-se/titan-ccp-record-bridge`
Docker image: `https://hub.docker.com/r/industrialdevops/titan-ccp-kieker-bridge-raritan`

Raritan Simulation Tool
Source code: `https://github.com/cau-se/titan-ccp-record-bridge`
Docker image: `https://hub.docker.com/r/industrialdevops/titan-ccp-raritan-simulator`

Frontend Component
Source code: `https://github.com/cau-se/titan-ccp-frontend`
Docker image: `https://hub.docker.com/r/industrialdevops/titan-ccp-frontend`

Common Libraries
Source code: `https://github.com/cau-se/titan-ccp-common`

Deployment Configuration
Source code: `https://github.com/cau-se/titan-ccp-deployment`

Appendix B

Scalability Evaluation Results

Table B.1. Results of the scalability evaluation showing the execution with the maximal achieved throughput T_{out} for each configuration of Δ_m, $\#_B$, and $\#_H$.

Δ_m	$\#_B$	$\#_H$	$\#_B + \#_H$	T_{in}	$\#_{in}$	T_{out}	$\#_{out}$	$T_{in} \leqslant T_{out}$	$0.9 \cdot T_{in} \leqslant T_{out}$
500	1	1	2	200.0	36100	1.8	264	false	false
500	1	2	3	200.2	36100	1.7	248	false	false
500	1	4	5	200.1	36100	1.9	278	false	false
500	1	8	9	200.1	36100	1.5	210	false	false
500	2	1	3	199.6	36024	23.2	2839	false	false
500	2	2	4	200.0	36100	53.3	6388	false	false
500	2	4	6	199.5	36018	78.3	7977	false	false
500	2	8	10	199.4	36032	112.8	13577	false	false
500	4	1	5	201.0	36100	22.6	3259	false	false
500	4	2	6	200.6	36100	65.7	9328	false	false
500	4	4	8	200.6	36100	82.8	12300	false	false
500	4	8	12	200.1	36100	118.6	15325	false	false
500	8	1	9	207.4	36100	29.7	4190	false	false
500	8	2	10	200.1	36100	80.3	12464	false	false
500	8	4	12	202.0	16300	67.1	4916	false	false
500	8	8	16	200.0	36099	188.9	20879	false	true
1000	1	1	2	100.1	18100	23.5	2843	false	false
1000	1	2	3	100.1	18100	42.8	5150	false	false
1000	1	4	5	100.1	18100	44.0	5305	false	false
1000	1	8	9	100.1	18100	87.8	4756	false	false
1000	2	1	3	100.0	18100	33.8	4791	false	false
1000	2	2	4	100.0	18100	81.4	11218	false	false
1000	2	4	6	100.1	18100	70.3	9398	false	false
1000	2	8	10	100.0	18100	103.0	13755	true	true
1000	4	1	5	100.0	18100	33.6	5153	false	false
1000	4	2	6	100.0	18100	92.9	10159	false	true
1000	4	4	8	100.0	18100	81.7	12549	false	false
1000	4	8	12	100.1	18100	71.1	11801	false	false

Table B.2. Results of the scalability evaluation showing the execution with the maximal achieved throughput T_{out} for each configuration of Δ_m, $\#_B$, and $\#_H$ (cont.).

Δ_m	$\#_B$	$\#_H$	$\#_B + \#_H$	T_{in}	$\#_{in}$	T_{out}	$\#_{out}$	$T_{in} \leqslant T_{out}$	$0.9 \cdot T_{in} \leqslant T_{out}$
1000	8	1	9	100.2	18081	51.0	6545	false	false
1000	8	2	10	100.7	18100	63.0	9987	false	false
1000	8	4	12	100.0	18100	85.6	13154	false	false
1000	8	8	16	100.1	18100	64.8	12544	false	false
2000	1	1	2	50.4	9100	38.8	5028	false	false
2000	1	2	3	50.4	9100	58.9	8311	true	true
2000	1	4	5	50.4	9100	60.0	7610	true	true
2000	1	8	9	50.4	9100	56.0	8393	true	true
2000	2	1	3	50.4	9100	31.1	4799	false	false
2000	2	2	4	50.4	9100	56.2	8938	true	true
2000	2	4	6	50.4	9100	55.4	8896	true	true
2000	2	8	10	50.4	9100	56.8	8844	true	true
2000	4	1	5	50.4	9100	36.5	5669	false	false
2000	4	2	6	50.4	9100	51.2	8513	true	true
2000	4	4	8	50.4	9100	53.5	8961	true	true
2000	4	8	12	50.4	9100	51.1	9000	true	true
2000	8	1	9	50.4	9100	35.1	5527	false	false
2000	8	2	10	50.4	9100	50.2	8791	false	true
2000	8	4	12	50.4	9100	51.8	8886	true	true
2000	8	8	16	50.4	9100	52.0	8994	true	true
4000	1	1	2	25.3	4561	22.9	3439	false	true
4000	1	2	3	25.6	4600	27.7	4457	true	true
4000	1	4	5	25.6	4600	26.3	4484	true	true
4000	1	8	9	25.6	4600	24.9	4500	false	true
4000	2	1	3	25.6	4600	24.7	3902	false	true
4000	2	2	4	25.6	4600	26.0	4335	true	true
4000	2	4	6	25.6	4600	25.0	4500	false	true
4000	2	8	10	25.6	4600	25.3	4500	false	true
4000	4	1	5	25.6	4600	23.0	3643	false	false
4000	4	2	6	25.6	4598	25.5	4499	false	true
4000	4	4	8	25.6	4600	25.0	4500	false	true
4000	4	8	12	25.6	4600	24.6	4500	false	true
4000	8	1	9	25.6	4600	22.8	3171	false	false
4000	8	2	10	25.6	4600	25.8	4496	true	true
4000	8	4	12	25.6	4600	24.8	4500	false	true
4000	8	8	16	25.6	4600	24.1	4356	false	true