



Universität Ulm | 89069 Ulm | Germany

Faculty of  
Engineering, Computer  
Science and Psychology  
Databases and Information  
Systems Department

# Design and Implementation of a Scalable Crowdsensing Platform for Geospatial Data

Master's thesis at Universität Ulm

**Submitted by:**

Ferdinand Birk  
ferdinand.birk@uni-ulm.de

**Reviewer:**

Prof. Dr. Manfred Reichert  
Dr. Rüdiger Pryss

**Supervisor:**

Robin Kraft

2018

Version from December 18, 2018

© 2018 Ferdinand Birk

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/de/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Composition: PDF- $\LaTeX$  2 $\epsilon$

## Abstract

In the recent years smart devices and small low-powered sensors are becoming ubiquitous and nowadays everything is connected altogether, which is a promising foundation for crowdsensing of data related to various environmental and societal phenomena. Very often, such data is especially meaningful when related to time and location, which is possible by already equipped GPS capabilities of modern smart devices. However, in order to gain knowledge from high-volume crowd-sensed data, it has to be collected and stored in a central platform, where it can be processed and transformed for various use cases. Conventional approaches built around classical relational databases and monolithic backends, that load and process the geospatial data on a per-request basis are not suitable for supporting the data requests of a large crowd willing to visualize phenomena. The possibly millions of data points introduce challenges for calculation, data-transfer and visualization on smartphones with limited graphics performance.

We have created an architectural design, which combines a cloud-native approach with Big Data concepts used in the Internet of Things. The architectural design can be used as a generic foundation to implement a scalable backend for a platform, that covers aspects important for crowdsensing, such as social- and incentive features, as well as a sophisticated stream processing concept to calculate incoming measurement data and store pre-aggregated results. The calculation is based on a global grid system to index geospatial data for efficient aggregation and building a hierarchical geospatial relationship of averaged values, that can be directly used to rapidly and efficiently provide data on requests for visualization. We introduce the Noisemap project as an exemplary use case of such a platform and elaborate on certain requirements and challenges also related to frontend implementations. The goal of the project is to collect crowd-sensed noise measurements via smartphones and provide users information and a visualization of noise levels in their environment, which requires storing and processing in a central platform. A prototypic implementation for the measurement context of the Noisemap project is showing that the architectural design is indeed feasible to realize.



## Acknowledgment

First of all I like to thank my supervisor Robin Kraft for all the support and the freedom to lead the thesis into a more general direction. I am grateful for revising different drafts and for helpful discussions to refine my ideas.

I also thank the Institute of Databases and Information Systems (DBIS) and my reviewer Prof. Dr. Manfred Reichert for making this thesis possible. A special word of thanks goes to my second reviewer Dr. Rüdiger Pryss for all his support in the past years and the general advice in future-related questions.

Furthermore, I like to mention Michael Stach and my fellow Students Johannes, Tobias, Andreas and Lukas, who were involved in vital discussions about concepts and technology related to software architecture, which helped my understanding a lot. I also like to thank them for making the university a place for fun and ideas.

I like to express my gratitude to my beloved girlfriend Dorothee for motivating me when I had a bad patch and her appreciation for being neglected way too often because of my studies.

A last thank goes to my family for the possibility to pursue my own interests and the *Erwin Hymer Group* as well as the company *Karl Storz Endoskope* for the financial support in the context of the scholarship "Deutschlandstipendium".



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Problem statement . . . . .	2
1.3	Objective . . . . .	3
1.4	Structure of the thesis . . . . .	3
<b>2</b>	<b>Fundamentals</b>	<b>5</b>
2.1	Crowdsensing . . . . .	5
2.2	Internet of Things . . . . .	9
2.3	Geospatial Data . . . . .	13
2.4	Cloud-Native . . . . .	18
<b>3</b>	<b>Project Context: Noise Sensing</b>	<b>29</b>
3.1	Introduction of the Noisemap Project . . . . .	29
3.2	Related Work on Noise-Sensing Platforms . . . . .	31
3.3	Noise and Sound . . . . .	32
3.4	Mobile Application . . . . .	35
3.5	Incentives . . . . .	36
3.6	Requirements of the System . . . . .	39
3.7	Challenges . . . . .	45
<b>4</b>	<b>Backend Design</b>	<b>49</b>
4.1	Concept . . . . .	49
4.2	Overall Architectural Design . . . . .	50
4.3	Fundamental Infrastructure and Automation Principles . . . . .	54
4.4	Apache Kafka - A Distributed Streaming Platform . . . . .	61
4.5	Bounded Context: Measurements . . . . .	65
4.6	Bounded Context: User Identity . . . . .	79
4.7	Bounded Context: Social . . . . .	84
4.8	Bounded Context: Incentive . . . . .	85

*Contents*

4.9 Bounded Context: Communication . . . . .	87
<b>5 Implementation</b>	<b>91</b>
5.1 Infrastructure . . . . .	91
5.2 Deployment . . . . .	95
5.3 Measurement-Context . . . . .	101
<b>6 Discussion</b>	<b>115</b>
6.1 Feasibility of the Approach . . . . .	115
6.2 Reasoning the Scalability . . . . .	117
6.3 Requirements Comparison . . . . .	121
6.4 Food for Thoughts . . . . .	128
<b>7 Conclusion</b>	<b>131</b>
7.1 Summary . . . . .	131
7.2 Outlook . . . . .	132
<b>A Appendix</b>	<b>147</b>



# 1

## Introduction

This Chapter motivates the subject of the thesis and introduces related problems. Further we briefly describe the objective and give an overview about the structure of the following Chapters.

### 1.1 Motivation

The importance of visualizing geospatially related information on a map to gain awareness of facts hidden in the information is known since 1854, as Dr. John Snow supposedly used a map containing points of Cholera outbreaks in the city of London, to identify a water pump as the root cause with the help of spatial distance patterns. Although certain facts are criticized as myths [50], this is the first known example of how to use a combination of visualization and spatial analysis on geospatial data to get evidence for a phenomenon. Technical advancements, such as the Internet or the creation of Geographic Information Systems (GIS), have drastically enhanced and extended the subject area of geospatial analysis since then.

Further developments in information technology will help to find more and more use cases for geospatial data, their visualization and analysis. The vision of an *Internet of Things (IoT)* is one of the most popular trends in computer science and one of the driver's in Big Data. A large section of the IoT-picture has to do with data collection in which context the "Things" are essentially small sensors attached to generic low-power communication technology, that is somehow connected to each other and the Internet. Smart mobile devices, such as smartphones, smartwatches and fitness tracker, are

## 1 Introduction

not different from that point of view but have become increasingly more pervasive in the recent years, now often being a ubiquitous companion for humans. There have been over 2 Billion smartphone users worldwide in 2016 and the number is predicted to grow beyond 3 Billion by 2021 [16]. Because smartphones, or smart devices in general, include various sensors, they offer great potential to build a “mobile sensor network” for crowdsensing geo-distributed environmental and societal phenomena. The fact, that smartphones are already distributed around the globe, is an enabler for producing large-scale data that is referenced with geo-locations determined by smartphone’s internal GPS capabilities. To visualize and analyze the data all the different user-owned devices produce, it has to be brought together and processed in a common platform.

### 1.2 Problem statement

In order to create valuable datasets, a certain density and distribution of users is needed, demanding specifically implemented features in relation to crowdsensing, such as sufficiently provided incentives for users. When growing awareness of crowdsensing and additionally deployed IoT sensors lead to an invisible comprehensive sensing utility, the result will be a “generation of enormous amounts of data which have to be stored, processed and presented in a seamless, efficient, and easily interpretable form [32].” Cloud computing is a promising foundation providing enough resources to handle and process that data. However, traditional monolithic designs have drawbacks in utilizing the flexibility of the cloud for faster and more modular development as well as scalable and elastic operation. Additionally, crowd-sensed data about environmental or societal phenomena often have a geospatial context, that must be considered when processing the data. Computations on geospatial data are more complex because of costly operations on fine-grained coordinates for aggregating the right data together. Therefore choosing an efficient concept for indexing and visualizing geospatial data and the resulting aggregates is important. A simple request-based aggregation for hundreds of thousands of users will either overwhelm badly designed systems and databases or at least result in high latency for data requests on large-scale datasets.

## 1.3 Objective

We pursue the development of a conceptual architectural design, that is scalable and general in a sense, that it is useful as a starting-point to develop specific platforms for crowdsensing of diverse geospatial data. The backend should be able to collect, process and efficiently return data for visualization via a state-of-the-art interface. We focus on a cloud-native approach with an architecture around bounded contexts, microservices and specific underlying infrastructure principles. We will combine this approach stream processing used in the Internet of Things for decoupled processing of incoming geospatial data. We precisely describe a use case in form of a project for crowdsensing of geospatial noise data and use its requirements as a guideline to define the capabilities for our generic design. Further we implement interesting parts of the design in the context of this project. The cloud-native approach is supposed to offer already some inherent scalability as well as a decoupled modular system design beneficial for distributed development in research groups with different focuses.

## 1.4 Structure of the thesis

We are going to introduce certain fundamental concepts useful for the understanding of the work in Chapter 2. After that, Chapter 3 is introducing the Noisemap project for crowdsensing of geospatial noise data, which we use as an example to identify generic capabilities that a platform for crowdsensing of geospatial data must have. Chapter 4 is describing the fundamental principles and technology we use to conceptually design the architecture of such a platform and subsequently Chapter 5 is highlighting interesting aspects of the design in form of a prototypic implementation. We will discuss the feasibility and scalability aspects of the design and compare our platform to the identified requirements in Chapter 6. The finishing Chapter 7 is summarizing the thesis and giving an outlook on further ideas and possible refinements, that someone implementing a precise use case must make.



# 2

## Fundamentals

This Chapter introduces terms and concepts that are critical fundamentals for this work. The common understanding of these terms is helpful for the other Chapters.

### 2.1 Crowdsensing

Crowdsensing is a concept for collectively bringing together sensor-data from a large group of individuals. With the advent of integrating sensors in ever getting smarter mobile phones, such data can be directly collected by users with their smartphone, which we more precisely call *Mobile Crowdsensing* because mobile devices are used. To understand the origin of Crowdsensing, we are going to introduce the more mature term *Crowdsourcing* as a generalization of Crowdsensing.

#### 2.1.1 Crowdsourcing

Crowdsourcing as a term has been made popular by Howe in 2006. It describes a combination of *crowd* and *outsourcing* [37], where it is seen “*as the act of taking a task traditionally performed by a designated agent (such as an employee or a contractor) and outsourcing it by making an open call to an undefined but large group of people [38]*”. A refreshed scientific definition is an integration of multiple Crowdsourcing initiatives and narrows the term to online activities:

**Definition 1.** *Crowdsourcing is a type of participative online activity in which an individual, organization, or company with enough means proposes to a group of individuals*

## 2 Fundamentals

*of varying knowledge, heterogeneity, and number, via a flexible open call, the voluntary undertaking of a task. The undertaking of the task, of variable complexity and modularity, and in which the crowd should participate bringing their work, money, knowledge and/or experience, always entails mutual benefit. The user will receive the satisfaction of a given type of need, be it economic, social recognition, self-esteem, or the development of individual skills, while the Crowdsourcer will obtain and utilize to their advantage that what the user has brought to the venture, whose form will depend on the type of activity undertaken [24].*

The specific clarification of the mutual benefit is an important aspect of every Crowdsourcing initiative to attract and retain a meaningful user-base. The task-receiver (from now on called user) must understand the benefit he gains in exchange for undertaking the task. Without proper clarification of incentives, users are per se reluctant to participate in online Crowdsourcing activities. Taking part in sensing-activities often includes users' time, battery of mobile devices and consumption of quota from the available data plan. Additionally, the long ongoing social discussion about privacy issues of online-activities in general, is an inhibition for users to take part in such activities.

### 2.1.2 Incentive Management

Well-understood benefits fill the role of user incentives. The motivation of users with well-chosen incentives is one of the fundamental problems of every application that relies on the crowd. For Mobile Crowdsourcing Zhang et al. reviewed three types [84]:

- *Entertainment*: Using gamification or location-based mobile game concepts involving sensors of users' mobile devices in order to motivate them to use the application. This incentive category must be well-aligned to the actual sensing-activities and the phenomena of interest. Therefore, its applicability is limited and requires considerable effort in gamification of the original problem.
- *Service*: A user has to fill out both roles (Contributor and Consumer) in order to use the system. This incentive category is universally applicable to many environmental problems. Features, such as visualization of the monitored values,

can be restricted with fine-grained access-control based on the degree a user is contributing to the overall application. This can motivate a user to contribute and in exchange get access to more enhanced information. However, there must be an intuitive and attracting user experience in order to gain users in the first place and users should not be annoyed with bad resource usage, such as battery, data-quota and private-data consumption.

- *Money*: Contributing users are paid money for their provided data. These incentives are the most general type, because it is independent of the application and can even reimburse the user for leak of private data. It can also compensate bad user experience to some extent.

### 2.1.3 Mobile Crowdsensing

The term *Mobile Crowdsensing (MCS)* has been coined by Ganti et al. and is the manifestation of general Crowdsensing, that involves smart mobile devices with the capability of sensing the environment to gather information, that can lead to valuable knowledge of an objective. We call that objective a *phenomenon*. Although, we do not limit MCS to the actual usage of the various sensors of smart devices, but also include the collection of subjective feedback about a phenomenon from the crowds' mobile devices in the scope of Mobile Crowdsensing.

We define a *sensing-action* as the actual task of gathering information about a specific phenomenon. If that phenomenon pertains to the individual that has sensed the information, the sensing-action is categorized as ***personal sensing*** and is often a personal record-keeping of a phenomenon directly related to the individual. Movement patterns or health attributes of an individual could be examples for personal sensing. More interesting, because harder to monitor, are large-scale phenomena that require the involvement of many individuals to gain some knowledge about it. Such sensing actions, called ***community sensing***, require many individuals acting as a community. Instead of merely keeping a personal history, records of the community are combined for analyzation for common patterns that might show further insights about the phenomenon [28].

## 2 Fundamentals

The user-involvement in the sensing-action can be differentiated between high and low. Some actions require the user to become explicitly active and initiate the sensing-action, this is called **participatory sensing**. Actions carried out autonomously in a continuous manner and therefore require almost no user-involvement, are called **opportunistic sensing** [46]. However, a user always has to participatory opt-in (in the Crowdsourcing sense) by downloading software, actively registering or activating a service at first to enable opportunistic sensing.

### 2.1.4 MCS-Spectrum

For Ganti et al., MCS is a range of *community sensing* paradigms and therefore excluding *personal sensing* from the concept of Crowdsensing. According to them, Community sensing can be described a spectrum of the user involvement from participatory to opportunistic sensing [28]. However, *personal sensing* is still strongly related to Crowdsensing, because the record that is taken by sensing the large-scale phenomena, is usually of interest to the individual in some degree anyway, since it can be also used for keeping track of the personal record history. Vice versa we state, that a comparison of many personal phenomena, for example to find common patterns in a social group, is an action regarding a large-scale phenomenon and therefore can be seen in a Crowdsensing context.

This consideration allows to view MCS as a two-dimensional **MCS-spectrum** as shown in Figure 2.1, with manifestations in user-involvement and the kind of processing of the sensed record. Crowdsensing starts **when at least two personal records of two independent individuals are processed together in order to gather further insights**. If an application can perform multiple sensing-actions, each can be placed individually within the

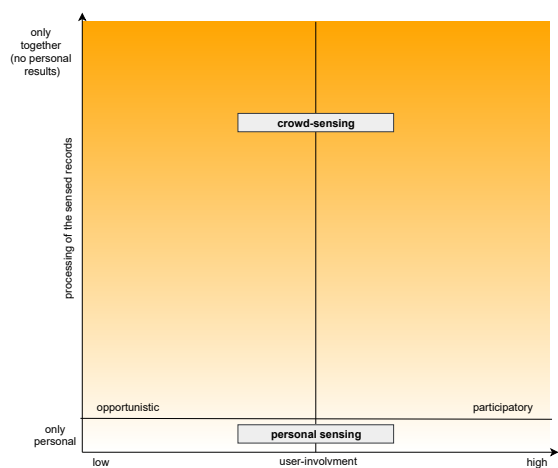


Figure 2.1: MCS-Spectrum for Crowdsensing and personal-sensing represented as a 2D-spectrum.



spectrum as a point. The connected points describe the Crowdsensing characteristic of an application. If for example, the dots are mostly placed on the upper left, the application can be described as opportunistic for large-scale phenomena. Otherwise, when a mobile application senses information and only performs processing locally without exchanging results their characteristic is placed at the bottom of the spectrum and we consider the application to only perform *personal sensing*. As soon as an application is exchanging results (either with a central backend or decentralized) **and** is taking these results into account while processing, it is effectively using results coming from the crowd and therefore is considered a *Crowdsensing* application.

## 2.2 Internet of Things

Alongside Crowdsensing, technical advancements are an enabler for large-scale environmental sensing. Enhanced micro-electromechanical systems (MEMS) and new low-powered communication technology have enabled cheap miniature sensors able to communicate wirelessly with each other or a gateway. Such *wireless sensor networks* (WSN) are already used in environmental monitoring applications. These small devices are one part of what is popularly called the *Internet of Things* (IoT). Definition 2 is provided by Gubbi et al. to clarify what is meant by the IoT [32].

**Definition 2.** *Interconnection of sensing and actuating devices providing the ability to share information across platforms through a unified framework, developing a common operating picture for enabling innovative applications. This is achieved by seamless large scale sensing, data analytics and information representation using cutting edge ubiquitous sensing and cloud computing.*

While the IoT is also about actuating devices and machine to machine (M2M) communication in general, for environmental monitoring the WSN-part is of special interest. Other than an ever moving human crowd, such devices can be mounted at specifically chosen urban locations and perform their sensing actions without requiring certain incentives. Additionally, their data can be assumed to be of higher quality in general, because once mounted, the context of the device is not changing considerably anymore, as opposed

## 2 Fundamentals

to mobile smartphones carried around by humans. However, even with the availability of cheap sensors, building a large-scale sensor-network with a certain density requires considerable planning, monetary resources and effort for maintenance. But there is new emerging communication technology, such as *Long Range Wide Area Network (LoRaWan)*, that can be a reliable link for direct communication between sensors and gateways using a long-range, low-power and unlicensed communication channel [80]. This technology has the ability to cover a large urban area with only a few well-placed antennas and is currently being deployed in an ever-increasing number of urban areas.

### 2.2.1 Crowdsensing, the IoT and the Maker Movement

The Maker Movement is currently one of the users of free communication technology, such as the mentioned LoRaWan. The movement has emerged from a culture focusing on *“Do it yourself” (DiY)* in the non-digital world. It is lately gaining momentum in digitalization and content production for online-platforms. Cheap micro-controllers and sensors enable the Maker Movement to take DiY one step-further and let them create applications for smart environments themselves. This has been resulting in a *“manifesto for diy internet of things creation [22]”* together with countless online available tutorials to create own applications for hobby or educational purposes. For Silvia Lindtner, who has conducted research about Hackerspaces<sup>1</sup> in China, *“this contemporary maker culture is concerned not only with open Internet technology and digital things, but also with physical things such as hardware designs, sensors, and networking devices that bridge the digital and physical worlds. While the earlier movement was concerned with the workings of software code and the workings of the Internet, this contemporary maker movement is concerned with hardware designs and the workings of the Internet of Things [48].”*

Despite they are in general independent technological and societal concepts, a combination of Crowdsensing, the IoT and the Maker Movement has not only the potential to create new ideas for sensors or how certain sensors could be used in different ways

---

<sup>1</sup>“Hackerspaces are community-operated physical places, where people share their interest in tinkering with technology, meet and work on their projects, and learn from each other [33].”

and at never thought of locations. The combination also entails a potential to create community-based sensor networks by using the motivation and eagerness of the Maker Movement to build and deploy custom devices. This will form a new type of Crowdsensing, that is not necessarily mobile like ubiquitous smart mobile devices, but still sufficiently distributed to cover large areas for environmental sensing. Much of the sensor data coming from such community-based sensor networks could also easily be integrated in existing Crowdsensing initiatives to enhance their data-quality and extend their use-cases into fields, where Mobile Crowdsensing is lacking continuity, because the crowd is moving around all the time.

One example where the Maker Movement created their own Crowdsensing application is the “Luftdaten-Projekt“. They introduced a platform called *luftdaten.info*, where everyone interested in technology and environmental transparency can build their own air quality sensor. The sensor, shown in Figure 2.2, can be mounted outside of the house to measure the mass concentration in the air and provide the platform with sensor data about air pollution [30]. Although often not professionally certified, such sensors can deliver reasonable accuracy to unveil a relative spatial distribution of certain phenomena [7].



Figure 2.2: Typical self-made sensor for measuring mass concentration in the air. Image by [41] ©.

### 2.2.2 Big Data

When we perform Crowdsensing worldwide and do not constrain the collection to a certain geospatial area, problems related to scale in multiple aspects can arise. When the number of active users grows above a certain size, the data volume exceeds processing capabilities of conventional approaches. We call such data volumes *Big Data*, when they are of high-volume, high-velocity, high-variety and/or high-veracity information assets,

## 2 Fundamentals

which are not easily processable with conventional information handling. New and innovative approaches for cost-effective processing are needed to enable further insights into the data [29].

### 2.2.3 Stream Processing

This paradigm is related to BigData and the IoT, because it is an enabler to continuously process data coming from all kinds of sensors in large volume. The term “refers to the ongoing processing of one or more event streams” [56]. The foundation is a *data stream* (or *event stream*), which is an abstraction for infinite and ever-growing datasets, where over time new data is arriving constantly. Examples are datasets like credit card transactions, stock trades or in our case simple measurements, that are constantly produced by the crowd. Additional attributes of such streams are ordering, immutability and ability to replay, because we know which event occurred before or after another event and events that happened, can by definition never be modified again at a later point, making the stream deterministically replayable.

Narkhede et al. place the *stream processing* paradigm between *Request-Response*, as the low latency programming paradigm and *Batch Processing*, as the high-latency paradigm. In Request-Response, processing only happens on request, making the paradigm suitable for problems where processing only requires a few milliseconds. Batch-Processing on the other hand, happens based on a certain schedule, for example once per day, and is mostly used with data analytics frameworks like Apache Hadoop or conventional data-warehousing, where the processing of large data often takes several minutes or even hours. Stream processing is filling the gap between those two paradigms by continuously processing new results in an asynchronous manner. This is helpful for problems where request-based processing would take too long to calculate the response, but scheduled processing is not reactive enough [56].

We can describe stream processing as a pipeline. Data is injected on one side as soon as it enters the system at time  $t$  and is then piped through multiple processing steps where it is possibly separated and joined together with other data several times. On the other side of the pipeline there is a continuous output of different results, that are based

on the most recent available data, which is the data from time  $t$  and earlier. As we ingest new data into the system at time  $t+1$ , it is piped through the same processing steps and ultimately updating the former outputted result, which is now based on data from time  $t+1$  and earlier.

## 2.3 Geospatial Data

In our physical world we often want to know where events happened or where objects are. Even when we put a signature under a formal contract, we usually have to provide the name of the location where the signing happened. The same premise is effectively relevant for Crowdsensing of environmental data. Certain phenomena are measured at a particular geographic location either by humans or statically deployed sensors. When we collect such events or measurements, we produce data with a **geographic relation**. These relations can be made in a direct or indirect manner:

- *directly*: The mapping is provided by coordinates in a spatial referencing system.
- *indirectly*: The mapping is provided by a reference to an entity, which itself has a geographic relation that can be indirect or direct. However, at some point the chain of indirect relations has to end in a direct relation. In this way every indirect relation can resolve the direct mapping by following the path of indirect relations.

Figure 2.3 shows an example of indirect and direct relation types for certain entities. For example, a noise measurement performed with a regular smartphone using GPS capabilities, is geographically related directly by the attached latitude and longitude coordinates the GPS-sensor provides at the time of the measurement. A sport or music event however, is often only related to a specific venue, such as an arena or a stadium, which usually can be found at a regular address with zip-code, street and number. The reference of the event to the venue and the reference of the venue to the address both are indirect geographic relations. However, the address is directly related to geographic space, because for example in Germany the “*Zentrale Stelle Hauskoordinaten und Hausumringe*” is maintaining a register with exact coordinates for every building [45].

## 2 Fundamentals

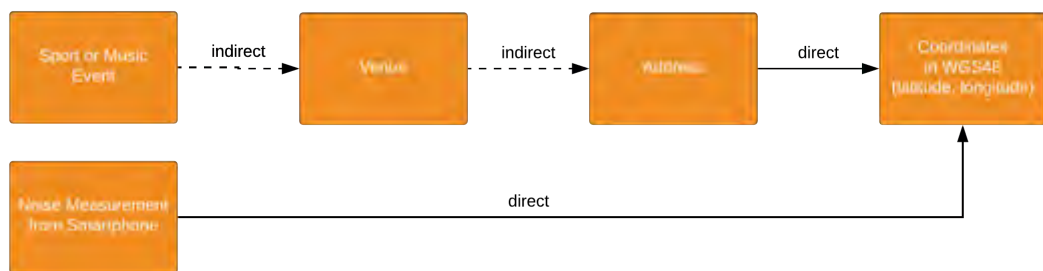


Figure 2.3: Example for different types of geographic relations.

The example shows, that both relation types can be used to find out where something has happened and therefore we provide the following definition:

**Definition 3.** *Geospatial Data (geodata) describes data, that can be directly related to geographic space. In order to directly relate initially indirect data, we demand that all related data forming the chain to the direct relation is contained in the same data document, for example by nesting the related data in the root entity.*

This definition allows a distinction between geospatial and non-geospatial data. For example, a data-document containing a Sport Event and only the Name and Address of the Venue where the Event is happening is not geospatial data, because it cannot be directly related to geographic space without querying for further data. The same document would be geospatial data, if the address is already resolved to geographic coordinates and therefore already directly related to geographic space.

### 2.3.1 Coordinate Referencing System

The relation of data into geographic space is usually something like a position or a few positions forming an area or more rarely positions forming a corpus. A coordinate referencing system (CRS) or sometimes spatial referencing system (SRS) allows us to describe the distribution of positions in the space. It has the purpose of unambiguously identify any point in the geographical space. We can however express each point in

different formats because there is not just one CRS. A point is usually described by latitude<sup>2</sup> and longitude<sup>3</sup>, but the representation of the angles is different for each CRS and some systems do not even use latitude and longitude to uniquely define a point in the space, especially systems that are constrained to a particular region, like the *Ordnance Survey National Grid* reference system used for surveys in Great Britain [4]. Additionally, the CRS defines a *Geodetic Datum* as a set of points to reference places on the Earth.

**The World Geodetic System (WGS)** is one of the most commonly used datums, particularly the revision *WGS84* (often also known under the code *EPSG:4326*[2]), because it is used by the well-known *Global Positioning System (GPS)* that is used in navigation and many other location-dependent applications by millions of people every day. It approximates the sea-level of the earth by using a defined ellipsoid. For practical computational use, we can express the latitude and longitude values in decimal degrees representing the respective angles. Depending on the number of digits, points can be expressed with a certain precision, that is again dependent on the distance to the meridian and the equator. In *WGS84* the equator has a length of about 40075 kilometers and is separated into 360°. With no decimal digits for longitude values, this results in a precision of about 111.320km for points on the equator. Similarly, this is roughly the precision for the latitude value of any point, because the length of the arcs from north-pole to south-pole are about 20004km long but only separated into 180°. Table 2.1 shows calculated precisions for different numbers of decimal digits. As the Table shows, for longitudes the precision error decreases the further away a point is from the equator, because the length of the parallel to the equator is less for points closer to the poles. Knowledge about geospatial precision is useful for efficient storage and calculation of geospatial data, because if a device is only capable of determining the geodetic position of a point with an accuracy of a few meters, like smartphones are able to, it is useless to store and perform calculations with coordinate values containing more than 5 decimals.

---

<sup>2</sup>Latitude: angle describing the position in north-south direction ranging from 90° at the north pole, 0° at the equator and -90° at the south pole

<sup>3</sup>Longitude: angle describing the position in east-west direction. It is defined by 0° at the meridian in London and the anti-meridian at ±180°, crossing the Fiji-Islands.

## 2 Fundamentals

Table 2.1: Precision values for numbers of decimal digits at latitude and longitude values for coordinates.

Decimal digits	Precision of Latitude or Longitude at 0° (equator)	Precision of Longitude at 45° (circumference ~28385km)
0	111 320 m	78 847 m
1	11 132 m	7 884.7 m
2	1 113.2 m	788.5 m
3	111.32 m	78.85 m
4	11.132 m	7.885 m
5	1.1132 m	788.5 mm
6	111.32 mm	78.85 mm
7	11.132 mm	7.885 mm
8	1.1132 mm	0.7885 mm
9	0.11132 mm	0.07885 mm

### 2.3.2 Representation

While indirectly related geospatial information is a sole reference, direct geospatial data can be of various types. The data structure must be able to represent points, lines, areas, areas with holes or even volumes. To ensure interoperability between libraries, tools and services the Open Geospatial Consortium (OGC) published the OGC Simple Feature Access Specification [40], that defines how to deal with geographical structures. The Simple Feature Specification defines a feature as an “abstraction of real-world phenomena”, which usually has a geometric object assigned to it, in order to provide a relation to a CRS. The GeoJSON Format is one of many geospatial data interchange formats, such as its alternatives Geographic Markup Language (GML), Well-Known Text (WKT) and Well-Known Binary (WKB). They were all developed to represent geodata based on Simple Features. GeoJSON must use coordinates based on a WGS84 datum with longitude and latitude expressed in decimal degrees. Section 4.5.4.1 provides further details about GeoJSON.



### 2.3.3 Data Binning

In order to analyze geospatial data, it is helpful to partition individual data records into buckets, often specifically chosen polygons or circles on the earth's surface. These buckets can be declared manually on purpose, to utilize human knowledge for getting results in a specific area of interest. Such human defined buckets could be boundaries of cities and states or at finer granularity boundaries of areas divided by postal-codes to aggregate and average values for those areas. However, getting access, creating and storing such information for worldwide analysis requires a lot of effort and may only be suitable for certain areas of interest.

**Discrete Global Grid Systems (DGGSs)** are a more universally usable alternative to human defined buckets. The OGC published a specification for a DGGS in 2017, where they define it as [66]:

**Definition 4.** *A DGGS is a spatial reference system that uses a hierarchical tessellation of cells to partition and address the globe. Sahr et al. characterize a DGGS by the properties of its cell structure, geo-encoding, quantization strategy and associated mathematical functions and is providing the following definition.*

The system represents a series of discrete global grids, where each grid is having an increasing number of cells with respect to its predecessor grid and therefore having a finer resolution. DGGSs can be created in various ways. Sahr et al. describe five design choices to consider when specifying a DGGS [67]:

1. A base regular polyhedron, such as an octahedron or an icosahedron as shown in Figure 2.4 on the left.
2. A fixed orientation of the base regular polyhedron relative to the Earth, such as the R. Buckminster Fuller's Dymaxion Orientation for an icosahedron.
3. A hierarchical spatial partitioning method defined symmetrically on a face (or set of faces) of the base regular polyhedron. Depending on the used polyhedron different polygonal shapes such as triangles, squares, diamonds and hexagons are used. An example is shown in Figure 2.4 in the center.

## 2 Fundamentals

4. A method for transforming that planar partition to the corresponding spherical/ellipsoidal surface like shown in Figure 2.4 on the right.
5. A method for assigning points to grid cells.

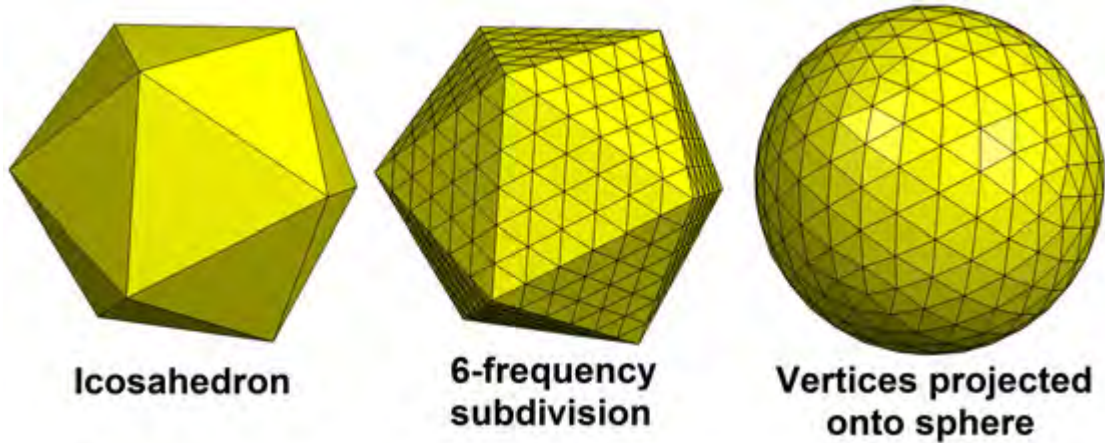



Figure 2.4: Example DGGs based on partitioning of an icosahedron.  
Image by [73] 

DGGs, such as the one shown in Figure 2.4 have the advantage, that they cover the whole spherical surface of the earth and can therefore be used generically to partition data collected at every place on the earth. Additionally, a system with different resolutions, like presented in [14], allow to represent the same data efficiently in differently sized buckets which is important to visualize and aggregate data on different scales. Because the grid is calculated deterministically with the help of the system-internal functions, we do not have to store the boundary of the cells to place points accordingly. We can use the system's method described in design choice 5 to calculate, in which bucket a specific geospatial location has to be placed and inversely, can calculate the boundary of a bucket if we know the index of a specific bucket.

## 2.4 Cloud-Native

The term *cloud-native* has already been used, to vaguely describe applications that have been developed specifically for the cloud to leverage its advantages. It has been the

research community that introduced the term, but since 2015 the industry is more and more using it to describe a concept, that focuses on modular application design around containerized microservices that are operated on cloud resources by the help of tools and platforms for dynamic scheduling [43].

We will clarify our understanding of the cloud and introduce a few important terms before providing our definition of *cloud-native*.

### 2.4.1 The Cloud

In information technology, “*the Cloud*” is a well-known general term. Originally a cloud-graphic has been used in network-diagrams to represent the Internet, which was someone else’s concern [76]. In part, that notion is still valid for our general understanding of the Cloud today. Without further specification the term actually cannot describe more than “*something is usable via network access*”. Vaquero et al. have proposed a definition for Clouds by studying more than 20 other definitions and integrate them:

**Definition 5.** *Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs. [75]*

The National Institute of Standards and Technology (NIST) belonging to the U.S Department of Commerce, thinks of the Cloud as a *model* or *paradigm* for the access and usage of computing resources as well as application resources and formed one of the most popular definitions around the Cloud concept:

**Definition 6.** *Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is com-*

## 2 Fundamentals

*posed of five essential characteristics, three service models, and four deployment models. [51]*

Both, Definition 5 and 6 highlight an *easy and flexible access to an adjustable pool of resources*. This notion is expressed in the essential characteristics of the cloud model, namely *On-demand self-service, Broad network access, Resource pooling, Rapid elasticity and Measured service*. The flexibility of self-service provisioning enables a cloud consumer to get access to resources without contacting other human business partners and without conducting time consuming negotiations of contracts. Instead, the consumer would often use an application programming interface (API) to perform resource provisioning and adjustment. The placement of the resource, for example in which data-center rack, is transparently handled by the provider, the only important aspect for the consumer is getting network access to that resource. Because there are often no long-living contracts for resources, the consumer can elastically provision and shutdown resources as needed and can therefore better align the computing power to the load of a system, which is one of the large benefits of the Cloud.

**Cloud Service Models** allow consumers to decide how much control they want to have about the resources in terms of configuration and maintenance. The different categories are important for early decisions in software development, when the resulting application should leverage cloud computing. This decision often conforms to *Make or Buy* decisions at, for example, producing a car. The NIST has defined the following categories for these offerings [51]:

- *Infrastructure as a Service (IaaS)*: The consumer can provision fundamental computing resources and is able to deploy and run arbitrary operating systems and applications. He does not manage the underlying physical infrastructure but has full access to configure operating systems and control storage or deployed applications.
- *Platform as a Service (PaaS)*: The consumer can deploy applications using programming languages, libraries, services and tools supported by the provider. Operating systems, storage and underlying infrastructure are managed by the

provider. He has control over deployed applications and configurations regarding application hosting.

- *Software as a Service (SaaS)*: The consumer uses applications of a provider, that are running on cloud infrastructure. He does not manage anything regarding application- or infrastructure configuration and only uses an application's capabilities through an interface.

*IaaS* in the example of producing a car would be equal to a car maker getting raw sheets of iron and all of the following steps are subject to the value chain of that particular car maker, giving him full control of the production process and quality outcome. *PaaS* however, would deliver the car maker not raw sheets of iron, but already bended doors or tailgates with certain holes already drilled. The car maker is no longer performing all parts of the value chain on its own but only some special steps like drilling additional holes for optional components, apply specific coloring and integrating specifically engineered components. *SaaS* would mean, a car maker is only focusing on the assembly part of the value chain when building a car. He would buy finished components and only assemble them together using defined screw holes and wires. In the software development context, these holes and wires would represent certain APIs that we can leverage to integrate their service into our platform product, rather than developing the service on our own.

### 2.4.2 Scalability and Elasticity

One of the main advantages of using the Cloud is easier elasticity and inherent scalability of resources, such that provisioning and de-provisioning of computing resources can happen automatically. If the expected load is unknown and volatile, the Cloud is a good choice to minimize the effort for adapting the available resources to the current load. This benefit is strongly related to elasticity, that in order to be defined, needs scalability clarified as a precondition. Both scalability and elasticity have been described and differentiated by Herbst et al [36].

- **Scalability** is described as the ability to sustain increasing workloads by using additional resources. It does not consider temporal aspects of how fast, how often,

## 2 Fundamentals

at what granularity scaling actions can be performed and is not related to how well the provisioned resources match the demand at any point in time.

- **Elasticity** is described as the degree to which a system is able to adapt to workload changes by scaling in an autonomous manner, such that resources match the demand as closely as possible at any point in time.

Accordingly, we first need a system that is able to scale, in order to achieve true elasticity. Scaling can happen either *vertically*, by increasing the power of the resource we currently have, or *horizontally*, by adding new resources. Cloud computing offers both types. Most providers have differently sized computing types that a user can easily change if more power is needed. At some point however, the largest computing type represents the upper limit for the vertical scalability. For truly scalable systems in a more and more connected society, it is inevitable to be horizontally scalable to overcome the limited size with an increased number of resources and therefore having the capability of theoretically scaling endlessly. Making a system scalable by replicating services or sharding<sup>4</sup> data into partitions can often be achieved easily. Achieving true elasticity however, requires a well-designed architecture and software platforms capable of orchestrating resources according to metrics describing the load of the system.

### 2.4.3 Microservices

Microservices became a popular alternative to monolithic software architectures after companies of the “Internet-Economy”, with Amazon and Netflix leading the way, published learnings of their internal software architecture. In 2014 Fowler and Lewis provided one of the most popular articles about the, as they define it, “Microservice-Architectural Style” [27]:

**Definition 7.** *In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These*

---

<sup>4</sup>Describes a horizontal partitioning of a database to distribute data between multiple smaller database instances instead of maintaining one large database.

*services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

In earlier research, we concluded, that the term “Microservices” is not only describing the component and the resulting architecture, but also a style, that is manifesting a culture and methodology of a software-developing organization approaching a specific business field. If a correct decomposition of such a business problem into microservices is missing, even a by definition decoupled Microservices-Architecture, will eventually become tightly coupled. However, it is undefined how to make such a decomposition making the result depending on the favored degree of modularity [11].

### 2.4.3.1 Bounded Context

A *Bounded Context* is a popular idea for decomposition of business fields having its roots in *Domain Driven Design* [25] and is often used as the modularization concept for defining certain microservices along business capabilities.

The term is best described by Evans himself as “a description of a boundary (typically a subsystem, or the work of a particular team) within which a specific model is defined and applicable [26].” The contexts serve as inner boundaries for a global domain, like Crowd-sensing of geospatially related data is, and are the result of strategic decomposition of a large domain into smaller subject-specific parts with a strong inner cohesion. They correspond to distinct *business capabilities* as mentioned in Definition 7 and represent autonomous business domains. If the boundaries are set correctly, each context-model can be very specific for the intended business capability, resulting in an *Ubiquitous Language* for the specific domain context. This language is the common understanding of domain experts, developers, business and project management. By avoiding a global canonical domain model, which would require dealing with unnecessary model attributes not relevant for a specific business context, we can prevent misunderstandings and maintain a smaller context model that is easier to overview and thus easier to change and enhance [55].

### 2.4.3.2 Microservices-Style

Microservices are a concept to modularize software, that emphasizes continuous delivery, automation through applying DevOps principles and organizations composed of small autonomous teams focused on a specific business capability of a larger software system. The style aims at solving shortcomings of other styles, that focused on teams aligned with their technical competences like Frontend-Development, Backend-Development or Database Administration. These other styles, often seen in monolithic architectures or early enterprise-SOA initiatives, require complex coordination between teams, because almost every change requires cross-team coordination, since changes often concern the whole stack from frontend to the database and most certainly operations, because someone has to deploy that change.

The Microservices-Style is forcing organizational structures composed of teams being aligned with bounded contexts and having full-stack responsibility for frontend, backend, database and sometimes even operation of their services. The assumption is that, if the system is well-separated into bounded contexts with services being aligned to those contexts, most changes are only context-internal and do not require cross-team coordination and therefore allow faster deployment cycles of changes, which ultimately results in faster feature development.

### 2.4.3.3 Microservices-Architecture

A Microservices Architecture is usually the result when an organization applies the Microservices-Style. On the macro-level the architecture is a distributed application composed of microservices that communicate over lightweight mechanisms such as Representational State Transfer (REST) or lightweight messaging and together provide the overall functionality of the application system.



#### 2.4.3.4 The Microservice

One Microservice can be seen as an independent application that has its own lifecycle and might use a different technology stack than other services in the system. Each service should perform a single task in the sense of the *Single Responsibility Principle*<sup>5</sup> to maintain a modular architecture and prevent side effects on features that actually are not changed at all. To allow independent development and deployment, each service is only allowed to be maintained by exactly one Team in order to have fast and unbureaucratic decision making and prevent conflicts in schema and model changes.

For the same reason services should have its own database and services in other contexts should only be able to retrieve the data via defined APIs by lightweight communication technologies. The consequence is that direct access to the database of a service in another context or integration of two services should be avoided when using microservices, because it contradicts the modularization of logic and data. An exception to this principle may be two services residing in the same functional context, which is usually managed by one functional team that can therefore still develop and deploy independently of others.

#### 2.4.4 Self-contained Deployment Unit

Building an application out of many independent services can be challenging in terms of deployment and packaging. Just deploying every service onto the same machine leads to many problems and conflicts in library versions, port numbers, configuration settings and not to forget great effort of managing dependencies, when moving some services onto other hosts. This is especially cumbersome to scale, because the exact environment has to be rebuild on every additional machine.

In a cloud-native environment we want to have services baked into a “self-contained deployment unit”. According to the Open Container Initiative (OCI) it should encapsulate the software-component, which is implementing the logic of the service, and all of its dependencies “in a format that is self-describing and portable, so that any compliant

---

<sup>5</sup>One component should deal with only one responsibility, so that there is only one reason to change the component, which is when the requirement for the responsibility changes [49].

## 2 Fundamentals

runtime can run it without extra dependencies [59].” In the cloud-native sense such a deployment unit is a standard container implementing the OCI runtime specification. Other than Virtual Machines, that have their own user and kernel space, containers share the underlying operating system of the machine and actually are just isolated processes using certain kernel features. The most prominent container-technology currently used in the cloud-native context is Docker<sup>6</sup>, because it has been the first technology to provide certain concepts and tools, that help managing the lifecycle of such standard containers. Some of the core notions of Docker described as follows:

**Dockerfile** Is a structured text file containing instructions telling Docker how to build an image. Every Dockerfile must describe a Base-Image which is used as a start state. Further, it describes certain commands that change the state of the image by copying certain files at specific directories or installing additional software.

**Image** Is a portable representation of a containers state. Images are named in the format of *Name:Tag*, where the tag usually represents the version of the application contained in the image. They have intermediate layers corresponding to one command in the Dockerfile, that can be reused in other images and thus saving disk usage.

**Registry** Is a central place to store and host Images. The container runtime can then pull those centrally stored images onto a specific computing node and run a container of it.

**Container** Is a concrete instance of an image. The container can be started, stopped, paused and snapshotted into other images.

### 2.4.5 Cloud-Native Definition

As it is the case with many previous hypes in information technology, there has not been an exact definition of what “cloud-native” means. The scientific community has used the term from about 2012, as Kratzke & Quint identified in their survey of research-publications containing the term. They worked out some aspects, that show a “common

---

<sup>6</sup><https://www.docker.com/what-container>

but unconscious understanding” across several of the studies and provided Definition 8 [43]:

**Definition 8.** *A cloud-native application (CNA) is a distributed, elastic and horizontal scalable system composed of (micro)services which isolates state in a minimum of stateful components. The application and each self-contained deployment unit of that application is designed according to cloud-focused design patterns and operated on a self-service elastic platform.*

From an industry’s point of view the hype about this term has multiple reasons. On the one hand, it emerged from Google’s approach to manage its production workload in several giant clusters managed by an orchestration system called *Borg*, which has been open-sourced under the name *Kubernetes*<sup>7</sup> in 2014 [77]. On the other hand, a rising interest in Docker-Containers is equally relevant for the cloud-native hype [52]. A further major reason is the preceding hype of developing everything distributed and scalable in form of microservices, which is now slowly becoming mature and state-of-the-art. The need of special tooling to use Microservices efficiently, gave birth to an ecosystem of numerous tools, technology and services for managing all the hard parts of developing a complex distributed system in the cloud context (see Appendix A.1).

**The Cloud Native Computing Foundation (CNCf)** has been established in 2015 as a subgroup by the *Linux Foundation* to foster collaboration between cloud-native participants, mainly cloud-vendors and creators of the tooling mentioned in the former paragraph. It is currently in possession of the strongest interpretational sovereignty within the software industry of what “cloud-native” means. Their Technical Oversight Committee currently uses the following definition [18]:

**Definition 9.** *Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled*

---

<sup>7</sup><https://kubernetes.io/>

## 2 Fundamentals

*systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal effort.*

Cloud-Native is nowadays a term with a broader meaning than just describing some sort of application architecture, thus we use the following definition:

**Definition 10.** (Being) *Cloud-Native is an organizational commitment to use cloud-native technologies (as defined in 9) as an enabler to build cloud-native applications (as defined in 8) in order to operate their software system reliable while at the same time constantly having developers applying changes to the software.*

### 2.4.5.1 Summarization

A cloud-native design can be summarized by emphasizing the three fundamental pillars:

- **Microservices oriented:** Application-Systems are separated into small independent components which increases the overall agility and maintainability of applications.
- **Self-contained Deployment Units:** Every service component of the application-system is a self-contained deployment unit, that has already packed libraries and environment dependencies in the correct versions within a container technology.
- **Dynamically orchestrated:** The self-contained deployment units are actively scheduled by cluster management tools to optimize resource utilization and automatic recovery of failed components.

We will use these pillars in Chapter 4 and 5 as a basis for the design and implementation of our platform backend.

# 3

## Project Context: Noise Sensing

This Chapter introduces the Noisemap project we use as an exemplary guideline for the design and implementation of our platform proposal. We are going to introduce its contextual terminology and is presenting incentives, requirements and challenges regarding that project, which are considered as a guideline in the following chapters.

### 3.1 Introduction of the Noisemap Project

Today's cities represent noisy environments because of various sources such as traffic, crowded streets with pubs, restaurants, nightclubs and possibly lots of additional construction work going on. For people with chronic disorders like tinnitus, this represents a challenging environment and has a direct effect on their condition in day-to-day life. Obviously, additional noise exposure supports hearing loss resulting in more severe tinnitus, therefore people already having hearing disorders should avoid noisy environments.

#### 3.1.1 Tinnitus

Tinnitus is a subjective phenomenon difficult to assess and for which it is hard to gather data that is useful for researchers. *TrackYourTinnius* is one Crowdsensing approach that follows the *ecologically momentary assessment* using mobile smartphone applications for assessing symptom severity by collecting self-reports about the perception of tinnitus combined with objective measurements [68]. They showed that Crowdsensing is a promising approach and users are actually motivated to use mobile Crowdsensing applications to provide patient feedback for tinnitus assessment [63]. They also started to

### 3 Project Context: Noise Sensing

use the data provided by the crowd in several scientific follow-up studies in [61], [65] and [62], which is an example for how Crowdsensing can help gathering valuable statistical data. Although applications like *TrackYourTinnitus* can help patients in understanding their tinnitus better, it is still hard for them to control noise exposure in public environments, because of missing information about noisy areas.

#### 3.1.2 The Noisemap-Project

We like to introduce this Crowdsensing approach to help tinnitus patients, but also people caring about their hearing ability, to get information about noise levels in certain areas to avoid noisy places as a preventive measure. The project can collect and visualize noise levels via Crowdsensing, by leveraging the ability of recording noise with mobile smartphones and correlating the measurements to the current geographic coordinates of the device. The focus of the project is on environmental (or recreational) noise exposure of users. As opposed to occupational noise exposure, there are almost no noise limits because of insufficient policies or lack of enforcement. In fact, each person is self-responsible, for how much it exposes itself to noise in their recreational time. Popular activities, such as sport events, music events or hanging around in a bar with music, can be threatening regarding loss of hearing or tinnitus in the long term. The problem is that most people are not aware of the amount of noise they are exposed to until they notice first symptoms of tinnitus. Cranston et al. show, that Fans in an indoor Hockey arena are exposed to sound levels with an average of  $97\text{dBA}$  and a threatening peak of  $124\text{dBA}$  sound pressure level [21].

Even if they care about their hearing ability, it is hard for them to track their consumption of noise and avoid noisy areas proactively. The problem is even worse considering the fact, that many workers are already exposed to a noise dose (see Section 3.3.3) of 100% during their work shifts. All non-occupational activities, such as pub visits, add to their total noise exposure, which is exceeding healthy limits as a result. Regular exceeding of noise dose limits in a worker's career is hazardous and a substantial risk factor for hearing damage [47].

### 3.2 Related Work on Noise-Sensing Platforms

**The goal** is to provide people, that care about their noise expose, with an option to regularly measure the noise around them with their mobile phone and use those measurements locally, to calculate an estimation of the daily noise dose for each user. Additionally, we want to use all measurements collectively in the Crowdsensing sense, to give users a geospatial visualization with best effort live data on a map. Users that care about their hearing, can use that information to be aware of current and past noise levels at certain places or events and perhaps decide to avoid them or take further measures like earplugs before visiting. The collected data produced by this Crowdsensing initiative is also very valuable for additional purposes such as city-planning, assessments in the real-estate market or scientific research of noise pollution by applying statistical methods, as the *TrackYourTinnitus* project did on their data.

### 3.2 Related Work on Noise-Sensing Platforms

Using Mobile Phones for measuring noise is not a new approach. Schweizer et al. have developed a participatory sensing application and an urban sensing platform to enhance the sparse dataset of noise pollution in cities. Their application is based on the Android operating system and leverages the smartphone's microphone and GPS-sensors to perform location-related noise-measurements. However, the measurements only focus on raw SPL measurements, rather than measuring frequency weighted samples like an *LAeq*. The measurements are submitted into an open urban sensing platform, called *da\_sense*, where it is stored linked to a user-account, which can control the public visibility of the data for privacy reasons. The platform allows access to the data via a visualization website using a heatmap overlay or through APIs [70]. The City Soundscape project for noise monitoring and acoustic urban planning published by Zappatore et al., is able to produce acoustic measurements with smartphones and collect them in a cloud-based platform "in order to help city managers in improving the life quality of their cities" [83]. A web-based visualization application is able to suggest certain noise reduction interventions to city planners and helping them meeting the European laws and regulations.

### 3 Project Context: Noise Sensing

Both platforms however, offer no visualization of the measured noise levels directly within the mobile application and only focus on web-based approaches, where a focus on data-quota and energy consumption is no priority.

## 3.3 Noise and Sound

In order to understand *Noise* we have to introduce *Sound* as a physical phenomenon of small pressure variations in form of vibrations that propagate as audible waves. The human ear or technical means like microphones can notice these audible waves and direct them to signal processing. From a linguistic point, the term *Sound* has no negative or positive association and therefore solely describes the physical phenomenon that can be measured.

Noise however, refers to “a sound, especially one that is loud or unpleasant or that causes disturbance” [60], and is therefore negatively associated and often not desirable for human persons. The disturbance can happen subliminal, when the noise has the form of a constant background noise, for example from distant traffic. Although, there are background sounds, which might be natural, like rustling leaves, a waterfall or a water-stream, where it is not clear whether the sound is unwanted or not, we can classify them as background noise, too [74]. Additionally, we classify the exposure to sound from activities a human deliberately undertakes as “noise exposure”, when the effect of that sound is unwanted, like for example hearing loss. Examples are music events that a human visits even if it is common knowledge that their sound levels are harmful.

While noise is omnipresent, the exposure to noise can be classified according to the setting. The World Health Organization (WHO) uses *occupational noise* for every noise that corresponds to a workplace setting and *environmental noise* for every other setting of human activities, which might be noise from traffic or playgrounds when residing in their homes or noise from nearby sport- and music events or bars. Environmental noise is also often called recreational noise [19].




### 3.3.1 Sound Measurement

In an occupational setting sound measurements are typically performed with a professional device, like depicted in Figure 3.1 and called *Sound Level Meter (SLM)*.

According to the Occupational Safety and Health Administration (OSHA), which is responsible for enforcing safety regulations at workplaces in the United States, these devices must be calibrated and meet the ANSI Standard S1.4-1971 (R1976) or S1.4-1983 [6]. A SLM measures variations of pressure in the air with a diaphragm in a microphone. These variations result in the output of a *sound pressure level (SPL)* in decibels, which is defined as an effective sound pressure in relation to a reference value, often  $20\mu Pa$ , the threshold of human hearing. The logarithmic scale of decibels seems to correlate to the human perception of sound and the interpretation of loudness. Therefore  $0dB$  for a human is usually silence and  $120dB$ , which is  $1Pa$ , can be the threshold of pain for human ear [20].



Figure 3.1: The functionality of a typical professional Sound Level Meter (SLM). Image by [3] .

**Recreational Noise Measurement** is hardly possible for a large part of the population, because these SLMs are expensive and only meant for occupational use. To close this gap and provide the population with an easy and uncomplicated way of performing recreational noise measurements is exactly the target of this project. It tries to use a large crowd to perform geospatial mapping of average noise levels. However, because usually contributors to our platform do not have professional SLMs, we only can rely on results of built-in microphones of smartphones.

### 3.3.2 Weighting Filters

The SPL is a pure physical value that is not perfectly suitable for depicting the human perception of sound. Therefore, SLMs often apply frequency-selective weighting filters, that are better aligned to human hearing.

The most popular weighting in human noise exposure research is the A-weighting curve measured in *dBA*. It accounts for the less sensitive perception of the human ear on low frequencies and therefore has a better description of the relative perceived loudness. However, it is not a useful approximation of the human ear's response to frequencies, but rather "an approximation of equal loudness perception characteristics of human hearing for pure tones relative to a reference of 40*dB* SPL at 1*KHz*" [10] and therefore actually only valid for quiet sounds with a relatively small SPL and pure tones. Although, by comparing different weighting filters empirically, it has been concluded that A-weighting gives a better estimation on the impact of given waveforms on human hearing than other weightings.

Other filters such as C-weighting can be useful in comparison with A-weighted values to determine if a sound has significantly low-frequency components, where its value would be higher than the A-weighted values [10].

### 3.3.3 Noise Dose

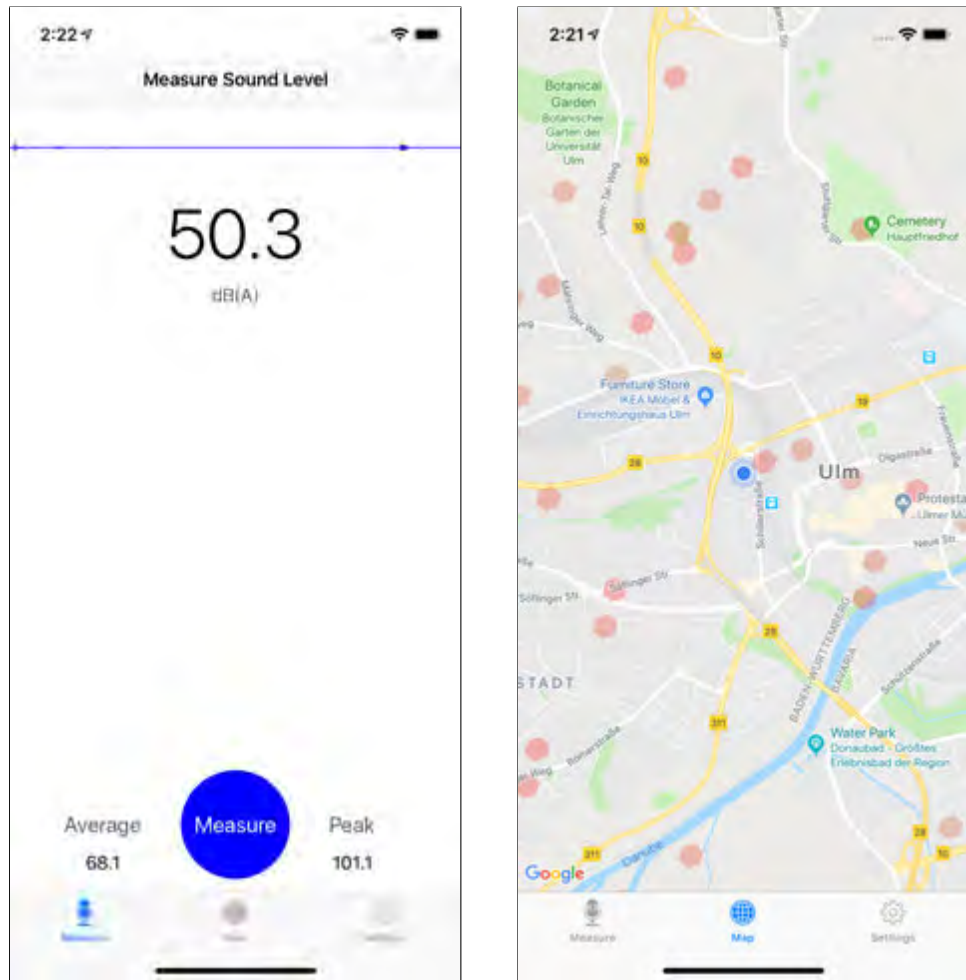
This abstraction is a personal value representing a total sound exposure normalized to eight-hours. However, as Tingay and Robinson showed, the Noise Dose value is defined differently by various regulations including different terminology and ultimately showing significantly different results in the end [72]. Together with the ISO-Standard, that is used within the EU, the NIOSH regulations show the highest results, because they define an exchange rate of 3*dB*, which means that an increase by 3*dB* is doubling the value outcome. The NIOSH Noise Dose is measured at a criterion level of 85*dBA* for a criterion time of 8hours, which would represent a Noise Dose of 100%. When for example, the individual is exposed to 88*dBA* continuously, it reaches the 100% mark after only 4 hours [17].

Although the Noise Dose is a mean to regulate noise exposure in occupational settings, the noise collected in recreational activities is contributing to the overall Noise Dose. A visit in an average discotheque or of a sport event, such as an NFL game, is likely reaching or even exceeding the limits of the daily Noise Dose alone [47].

## 3.4 Mobile Application

The project includes the mobile application shown in Figure 3.2, which is developed independently of this work by Robin Kraft at the Databases and Information Systems Department of the University of Ulm. The IOS application is capable of measuring the SPL for a certain time-window of a few seconds and subsequently, aside from a regular *Time-Weighted-Average (TWA)*, deriving different weighted values, such as *A-weighted (LAeq)* and *C-weighted (LCpeak)*. Since the purpose of microphones in smartphones is recording voice and sounds for multimedia applications, their accuracy for measuring the SPL is usually not comparable to professional SLMs. To at least provide some measurement accuracy, the application is using correcting factors for the measured SPL, which are retrieved through calibration. Each microphone behaves differently and finding out the correct factor for calibration requires laboratory equipment, which is the reason why the mobile application is developed for IOS based devices in the first place, because the operating system is limited to a smaller group of device types than the Android operating system. However, even after calibration the values cannot reliably be used for exact testimonies of noise levels, but they should still be good enough to highlight significant differences between quiet, moderate and noisy areas [54].

### 3 Project Context: Noise Sensing



(a) Screen to measure noise.

(b) Screen to visualize aggregated measurements on a map.

Figure 3.2: Example mobile application for measuring and viewing noise data by [42].

## 3.5 Incentives

Like other crowd sensing initiatives this project has some requirements in order to be successful. It is critical for the project to attract many users and motivate them to actively measure noise in different areas to maintain a steady input stream of noise data. Considering that users running the frontend application on their mobile phone have drawbacks like battery drainage, they have to be provided with sufficient incentives. Noisemap is currently a small scientific project with a limited financial budget, therefore

incentive types, that involve paying money to users are not considered. We favor a combination of the *Service* and *Entertainment* categories, as they are defined in Section 2.1.2.

### 3.5.1 Service Incentives

One option of the platform is providing information as an incentive. Sharing the collected data with users is an obvious first step to motivate users to take part in sensing-actions. They provide measurements and the platform offers them information about collected measurements, either by visualizing them or showing a list of conducted measurements by other users nearby. This way we compensate a user's effort with information he would not be able to get without using our platform.

As a next step, we can create fine-grained access control for different types of information based on a user's activity. This will privilege actively contributing users. A simple classification of users into *Member Levels* can help to build such an access-control:

**Guest** An unregistered individual independent of whether it is contributing or not, because we cannot distinguish guest measurements. They only have access to pre-aggregated values of the past day and only within a certain radius, for example 1 km, of their current location.

**Newcomer** A registered person until it has mastered a certain task, like provided at least one measurement a day for five days. Like *Guests* they have only access to information within a certain radius of their current location, however *Newcomers* can also show pre-aggregated values which are more recent than 1 day.

**Member** A registered person when it has mastered the *Newcomer* task. Once a person reaches this class, we value the effort of mastering the *Newcomer* entrance and never downgrade it the person further, even if it is no longer actively contributing. This should keep the hurdle small to become an active contributor again. *Members* can visualize pre-aggregated values without any constraint on current location and additionally access single measurements within, for example a 1km radius, of their current location.

### 3 Project Context: Noise Sensing

**Active Member** A *Member* that has either submitted at least one measurement in the last hour or more than 168, which is the number of hours in one week, measurements in the last seven days, not including the current day. Being active has the benefit, that all features of the platform are accessible without constraints for the current location. An *Active Member* can not only request single measurements or pre-aggregated values for any area on the world, but also request live-aggregations to show an aggregated visualization of the measurements collected in the most recent 15-minute window.

**Premium** This is a reserved class for possible payed features that can be introduced in such a platform in the future.

An additional information incentive can be the personal noise exposure calculation in form of a Noise Dose, that is calculated if a user is constantly performing automatically scheduled measurements in the background. Such information can be very valuable for any user that is caring for their hearing ability.

#### 3.5.2 Entertainment Incentives

Integrating certain social aspects into the platform can additionally enable the possibility of using entertainment incentives. Most of the entertainment incentives result in social reward and by that try to motivate users for actively contributing to the platform. The first and obvious step is to collect user statistics that each user can share with other users on a custom user profile. The next step would be integration of Gamification. The creation of a concept for Gamification would exceed the scope of this work, because it is strongly related to specific use-cases and involves aspects of human psychology. Still, we want to provide exemplary approaches, that show how such a Gamification could look like.

We would use a concept of Challenges and corresponding Awards.

**Challenges** Describe specific tasks, which can be consistently or temporarily available and can be additionally geographically constrained and reoccurring randomly. Users can view these Challenges in the application, or even on a map in the

case of geographically constrained Challenges. Further, a user might receive notifications advertising Challenges concerning its interest.

**Awards** Describe medals or pins that can be shown in a user profile once achieved a specific Challenge. There could also be a publicly available board, where other users can view recently awarded users.

Examples could be Challenges being inviting to measure continuously, like at least one measurement every quarter of an hour for a certain number of hours, or inviting to measure geographically distributed, like providing a measurement in at least 5 geographical cells within a certain timespan. Additionally, certain areas without good coverage could be used in Challenges as special areas with bonus rewards to foster a balanced geographical coverage.

## 3.6 Requirements of the System

Pryss et al. have developed a set of requirements useful in the context of Crowdsensing and mHealth that are based on RESTful principles. Some of them have been used as a rough guideline to develop the specific set of requirements in the Noisemap project [64]. We do not constrain the term *system* to the backend system of the platform, but also take frontend requirements into account, too. Also system and application might be used interchangeably in this section. We identified and listed the requirements divided in two categories:

- *Functional*: The functionality the system should have and which represents the business functions that an implementation must provide is listed in Section 3.6.1.
- *Non-Functional*: The constraints and quality criteria not directly related to functionality, but should also be fulfilled is listed in Section 3.6.2.

### 3.6.1 Functional Requirements

The following requirements in Table 3.1 present the functionalities, the system should provide to fulfill the purpose of the project. We categorized them in Noise Level Measurements, Visualization, Users, Notifications, Incentives and Privacy.

Table 3.1: Functional Requirements for the Noisemap-Project.

Nr.	Requirement	Description
1	Noise Level Measurements	The system should offer a functionality to measure the objective noise level (LAeq, LCPeak, TWA) at a certain area utilizing sensors of mobile devices.
1.1	Noise-Dose	The system should be able to continuously measure the noise around a user and provide the user with information about his current noise dose.
1.2	Triggers	The system should differentiate between triggers (User-Initiated, certain Locations, Time, Notification) for measurements that can be configured by users.
1.3	Subjective noise Level	Additional to objective measurements, the system should be able to measure the subjective noise level of users.
1.4	Offline capability	The system should be able to let the user measure within areas with bad/none cellular connection and re-sync when the connection is established again.
2	Visualization	The system should offer a visualization of the noise level in a specific area as a map overlay. The map should be adjustable by the user, for example to scroll and zoom in the map.
2.1	Heatmap	The system should offer a Heatmap of single measurements, filterable by time and measurement types until the user zooms out of a certain detailed zoom level.
2.2	Aggregations	The system should offer a visualization of aggregated measurements within certain polygons in a defined resolution for different zoom levels.
3	Users	The system should allow users to register and login.
3.1	User Profile	The system should allow users to maintain a profile.
3.2	Groups	The system should allow users to join groups and discuss about noisy hotspots.

continued . . .



Table 3.1 continued: Functional Requirements for the Noisemap-Project.

Nr.	Requirement	Description
4	Notifications	The system should be able to notify the user when something interesting happens (group related activity, awards and promotions) or when the system notices that the user resides in an area of interest (requesting a measurement, notifying about critical noise levels).
5	Incentives	The system should motivate users to register and contribute measurements.
5.1	Restrictions / Rewards	The system should restrict certain functionality to registered users and reward their contributions with additional functionality.
5.2	Gamification	The system should offer different gamification features in order to animate the user to submit data more frequently. Features may include but are not limited to statistics, submission streaks, awards or comparisons with other users in the user's area.
6	Privacy	The system should maintain user-privacy interests as far as possible and only share information about the user with others, which that respective user has explicitly configured to be shareable.
7	Fraud-Protection	The system should take measures to notice and prevent ingestion of false data. Questionable data should be logged somewhere and users with a high degree of false data can be notified, penalized or blocked by the system.

### 3.6.2 Non-functional Requirements

Beatty and Wieggers worked out the following 16 qualities of non-functional requirements divided into internal and external (refer to Table 3.2). A quality is external, if it is mostly important to users of the software, such as Usability, Performance and Availability. A quality is internal if it is mostly important for development and operations, such as Modifiability, Efficiency and Scalability.

### 3 Project Context: Noise Sensing

Table 3.2: The qualities types of Non-Functional Requirements after [79].

Nr.	Name	Type	Description
1	Efficiency	Int.	How efficiently the system uses computer resources.
2	Modifiability	Int.	How easy it is to maintain, change, enhance, and restructure the system.
3	Portability	Int.	How easily the system can be made to work in other operating environments.
4	Reusability	Int.	To what extent components can be used in other systems.
5	Scalability	Int.	How easily the system can grow to handle more users, transactions, servers, or other extensions.
6	Verifiability	Int.	How easy it is to confirm that the software was implemented correctly.
7	Availability	Ext.	The extent to which the systems services are available at any given time.
8	Installability	Ext.	How easy it is to correctly install, uninstall, and reinstall the application.
9	Integrity	Ext.	The extent to which the system protects against data inaccuracy and loss.
10	Interoperability	Ext.	How easily the system can interconnect and exchange data with other systems or components.
11	Performance	Ext.	How quickly and predictably the system responds to user inputs or other events.
12	Reliability	Ext.	How long the system runs before experiencing a failure.
13	Robustness	Ext.	How well the system responds to unexpected operating conditions.
14	Safety	Ext.	How well the system protects against injury or damage.
15	Security	Ext.	How well the system protects against unauthorized access to the application and its data.
16	Usability	Ext.	How easy it is for people to learn, remember and use the system.

We specified the project's internal requirements in Table 3.3 and the external requirements in Table 3.4.

Table 3.3: Internal Non-Functional Requirements for the Noisemap-Project.

Nr.	Quality	Description
1	Efficiency	The system should use computing resources in the back-end in a balanced manner between storage and computing efficiency. Often required expensive computations should be based on pre-computed persistently stored aggregates. Additionally, the system should minimize the size of the required bandwidth, especially to at public facing APIs, to prevent too much data consumption of users' quota.
2	Modifiability	The system should be designed in a modular manner with defined interfaces. A change in one module without changing the interface should not affect other modules at all. It is desirable that even an extension of functionality in one module should not affect other modules.
3	Portability	The system should be composed of self-contained modules that can be transparently located across computing resources. The package of each module should contain the required environment configuration and dependencies and abstract from the underlying infrastructure.
4	Reusability	The modules should be implemented with generic interfaces to be reused for other contexts than noise-measuring with a minimal change effort.
5	Scalability	The system must be highly scalable on module level. Only the module under load should be required to scale by just adding more instances of the module. The system must support horizontal scalability to avoid limitations of vertical scalability.
6	Verifiability	The system can implement certain measures to test and verify the correctness of interfaces and used algorithms. Correctness can be confirmed manually within an integration-environment before releasing a module to production.

Table 3.4: External Non-Functional Requirements for the Noisemap-Project.

Nr.	Quality	Description
7	Availability	As a platform that is dependent on user satisfaction, we should avoid outages longer than a few seconds and if not possible, at partial failures silently hide faults in certain modules and give the user at least a partial response of the working modules.
8	Installability	The user-facing application must be easy to install and accessible through an application store native to the underlying operating system.
9	Integrity	The system should use only technologies that are able to replicate and possibly allow backups of stored data. However occasional loss of a measurement before or within the processing pipeline is not critical.
10	Interoperability	The system must have an API that is accessible through the Hypertext Transfer Protocol (HTTP). The API should mostly follow the REST style.
11	Performance	The application must be able to react on user requests within a few-hundred milliseconds. If not feasible, the user-facing application should tell the user that this is a sophisticated operation and might take a few seconds.
12	Reliability	Regular outages should be avoided to maintain a good user satisfaction.
13	Robustness	The system should deal with partial failure of certain modules and automatically try to restore the module within the cluster. Healthchecks should help noticing failed components and recreate them before actual failures propagate to the user.
14	Safety	No special requirement
15	Security	The system must secure certain API-endpoints and authorize every request. Sensitive user data like credentials must only be stored in form of a hash.
16	Usability	The user-facing application must provide a natural look and feel that is native to the device-platform and therefore help the user to quickly understand the functionalities. Settings should provide hints or tooltips that explain the effect of the setting in detail, especially for privacy settings.

## 3.7 Challenges

Using a smartphone application for Crowdsensing unleashes multiple challenges. Especially using opportunistic Crowdsensing to measure autonomously without manual user actions offer problems regarding the smartphone context while performing the measurement. Automatically detecting some contextual situations is a challenging task for the Crowdsensing application.

This Section lists some problems and explains, why they are critical especially for a crowd-based noise measurement system, that partly uses opportunistic activity.

### 3.7.1 Public-Private Detection

The platform is supposed to collect measurements of environmental noise levels, but also be a tool for the user to measure his personal noise dose over the course of his recreational time. Assuming that users have activated opportunistic features to perform measurements in the background, the system has problems to recognize which measurements are relevant for the crowd. While noise measurements performed in private space are perfectly suitable for personal calculations, they do not provide any value or even distort values intended for the crowd.

For example, a user living next to a noisy place like the Time Square in New York City might have his smartphone lying around next to a window that is well shielded to prevent infiltration of outside noise. In that case an automatically performed measurement would measure the inside-noise of the privately owned flat and relate that measurement to a GPS-location almost exactly placed at the Time Square. Using this measurement for aggregations will result in distortion of the environmental value at the Time Square.

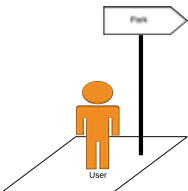
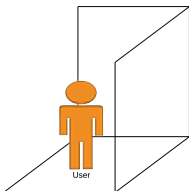
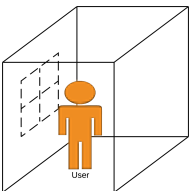
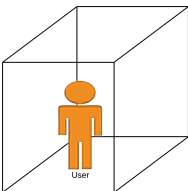
Indoor-Outdoor Detection could mitigate the problem by some degree with the cost of losing measurements, that are performed at public indoor places like restaurants or other event locations. Assuming that GPS-sensors in smartphones work reliably, we could use the signal strength of received satellites. Deep inside a building with no windows the GPS is only reaching a few satellites or the signal even might not be available. The

### 3 Project Context: Noise Sensing

inverse effect should be observable at an open space somewhere outside, because the GPS-receiver can receive the signal of multiple satellites in probably good shape and determine the position very accurately. Outer walls of buildings are problematic because they harm the GPS-signal even from outside when approached closely and inversely windows are beneficial for receiving signals indoor. There might be certain accuracy values for positions determined by the GPS, that cannot be reliably assigned to indoor or outdoor scenarios.

**Related Work on Indoor-Outdoor Detection** is conducted by Wang et al.[78], who are using a machine learning approach leveraging signal strengths of GSM base stations to reliably determine one of four scenarios. They describe the problem by splitting the context into four environments, a phone-carrying person can be in. Table 3.5 shows the types including their definition.

Table 3.5: Four environments related to indoor-outdoor contexts based on [78]

Environment	Open Outdoors	Semi-Outdoors	Light Indoors	Deep Indoors
Definition	Outside a building	Close to a building	In a room close to windows	In a room without windows
Example				

They employ different algorithms to learn patterns of available base station and corresponding signal strengths. They tested the results in experimental areas different from the training area and showed, that the Random Forest algorithm can determine the scenario with an accuracy of 95.3%.

Zhou et al. introduce a system called *IODetector* using multiple sensors, such as acceleration, proximity, light and magnetic field sensors, as well as cell-tower received signal

strength (RSS) to detect outdoor, semi-outdoor and indoor scenarios. They employ a light-detector using light intensity as a first component output, when the proximity sensor indicates, that a valid measurement is possible, and the time suggests daytime. Additionally, they associate different patterns of cell-tower RSS with certain scenarios as a component output and lastly employ a magnetic field sensor to scan distinct magnetic patterns of indoor and outdoor environments. These component outputs are aggregated to compensate weaknesses of the different approaches. Their evaluation showed that IODetector is capable of detecting the scenarios with an overall accuracy of 82% [85].

#### 3.7.2 Pocket Detection

A smartphone will produce invalid measurements when the microphone is covered while measuring noise, because the sound waves are disturbed and do not represent real outside noise. Additionally, while a phone is residing in a pocket or a bag, there might be newly created sources of noise that are close to the microphone coming from friction or clattering keys next to the phone.

The mobile application must therefore be able to reliably prevent such measurements in opportunistic scenarios, where it is automatically performing the measurement activities. One approach might be, to only measure when the user is actively using the phone, which is detectable for example by checking if the screen is turned on or off. A more complex approach would use techniques to provide contextual awareness and determine the placement and movement of the smartphone to predict, whether the microphone is covered or not.

**Related Work on Pocket Detection** is conducted by Yang et al. with an approach, that uses a combination of values coming from the proximity- and light-sensors of smartphones [82]. They synchronized both sensors before feature extraction of 40 distributed samples contained in a 4-second-wide window. In a subsequent step they average the 40 values and use thresholds to get input variables for a joint rule-based classifier that can for example decide, that a phone is “likely in pocket”, when proximity value is close and light value is dark. To prevent certain false-alarms they apply validation

### *3 Project Context: Noise Sensing*

by checking for example the pattern of the proximity distribution within that 4 seconds. They claim that the overall accuracy of their system is above 98% with a negligible CPU processing power consumption.

#### **3.7.3 In-Vehicle Detection**

Measurements taken when a person is sitting in a car is also a class of measurements, that is not interesting for the crowd, because they are not representing the environmental noise level in the streets. Some cars are well-shielded against outside noise and therefore engine sounds and sounds from multimedia systems are the main source of sound, which are however irrelevant for other users interested in environmental noise levels.

A simple approach could be, to calculate the distance a user has moved during the measurement and discard the measurement if the distance is above a certain threshold, which would indicate a fast traveling user. Although, this would not solve situations where a vehicle is waiting at traffic lights or in traffic jam. Another approach might be, to let the user activate a setting, that tells the application, which Bluetooth connected device is the multimedia system of car and block measurements for the time this device is connected.

**Related Work on In-Vehicle Detection** is conducted by Nikolic and Bierlaire in a review of multiple transportation mode detection approaches [58]. They concluded that several valuable data sources are not used by the reviewed approaches. Using real time traffic information and innovative sensors such as temperature, barometer or humidity could be exploited to form more accurate models for decision making regarding the current type of transportation mode. They also mentioned, that generative models are unpopular for classification tasks on the mobile device itself, because of higher computation costs. On a mobile device models based on Decision Trees provide the best combination of accuracy and resource usage. Overall the approach of Stenneth et al. [71] produced the most satisfactory accuracy.



# 4

## Backend Design

This Chapter introduces the structural- and architectural concepts we use to design the backend of a more generically usable platform for applications like the *Noisemap-Project*. Further, selected functional capabilities of such a platform are elaborated to show certain challenges and potential solutions.

### 4.1 Concept

The backend will make use of state-of-the-art technology and concepts. We think the cloud-native concept, introduced in Section 2.4, is a promising enabler to build an extendable cloud-based platform, that can scale with a growing amount of data and number of users. We also believe, that there will emerge additional use cases for the collected data in the future and therefore pay attention to maintain a decoupled architecture, that is making it easier for developers and researchers to introduce new features, new approaches for calculation or even completely new use cases in the long-term future.

The cloud-native design is following the microservices style, which divides functional responsibilities into different bounded contexts and is thereby separating functionalities in multiple microservices. These contexts will be hard borders for team collaboration because there is only exactly one team responsible for each context to keep all the decisions of the internal design within a small group of people, making it able to perform changes fast and prevent coordination overhead. This design is making it possible to have multiple developers or researchers work on independent parts of the platform without

## 4 Backend Design

interfering other components. Such an approach can be beneficial when researchers with different focuses work on new approaches in their field and like to test them within the platform. They can develop and deploy their features in their own pace, rather than being forced to wait for colleagues deciding to make a new deployment of the application in a monolithic approach. The only point where minimal coordination happens is at outside-facing interfaces of each context, making it possible to access data of other bounded contexts in a reliable way.

The technology we consider for the design is popular open-source technology, that has its focus on supporting the development of cloud-native applications by either being highly dynamic in terms of scalability and elasticity or providing automation for certain tasks related to development and operation. Because of the decoupled platform design with independent services, its architecture is effectively polyglot and therefore giving each team free choice of programming language, database technology and libraries. They can use what is best suited for their problem or simply preferred because of former knowledge or already present code snippets. Section 4.3 is elaborating on how we use self-contained deployment units that are dynamically orchestrated, to provide an abstracted infrastructure layer, that makes it possible to allow each team defining their own automated build- and deployment pipelines by declaratively specifying the pipeline and infrastructure configuration text-files.

### 4.2 Overall Architectural Design

First, we like to introduce the overall architectural design, which is a composition of multiple bounded contexts. To start, we identified the current core capabilities (others might be introduced as new services easily in the future) needed in a platform for crowdsensing of geospatial data and assign those to the best matching bounded contexts. We take into account the requirements of the Noisemap project as a guideline. Table 4.1 is providing an overview of the capabilities assigned to different bounded contexts, which define the first level of modularization for our architectural concept.

## 4.2 Overall Architectural Design

Table 4.1: Key business capabilities of the backend mapped to bounded contexts

Nr.	Capability	Bounded Context
1.1	Let users register and authenticate with the backend.	User Identity
1.2	Let users change their password and provide lost-password recovery.	User Identity
1.3	Let users deactivate and delete their account.	User Identity
2.1	Let users maintain a User Profile with personal information.	Social
2.2	Let users join groups and start, follow and contribute to discussions.	Social
2.3	Provide geospatial relation of groups and discussions.	Social
2.4	Trigger a notification to the user on new contributions in subscribed discussions or subscribed areas of interest.	Social
3.1	Collect measurements provided by smartphones and other IoT-devices and streamline them as a common input stream.	Measurements
3.2	Aggregate the measurements to provide min-, max- and average values within certain geospatial areas and time-based windows.	Measurements
3.3	Allow geospatial request filtering by specifying the area of interest and time windows.	Measurements
3.4	Allow access to single stored measurements with a pagination like limitation for the number of results.	Measurements
3.5	Provide an API that returns the results in a common geospatial format to allow straightforward visualization with frontend technology	Measurements
4.1	Track user contributions for authorization of additional functionality and to show them their progress	Incentive
4.2	Maintain awards and streaks for certain achievements that motivate users to continue in contributing measurements.	Incentive
5.1	Inform users about certain events via email.	Communication
5.2	Inform users about certain events via push-notifications.	Communication
5.3	Let the user define preferences for the type of events he likes to be informed.	Communication
6.1	Manage meta-information about statically deployed sensors.	Sensors

## 4 Backend Design

Each bounded context can have one or more services that are responsible for providing the desired functionality. Each context is also using their own database and a data-schema that fits their use case best. The proposed overall architecture is shown in Figure 4.1. The infrastructure fundamentals for our architectural design, is described in Section 4.3, where we introduce tooling, concepts and systems that allow to deal with the increased complexity of such a distributed system. In the subsequent Section 4.4, we are going to introduce Kafka as the data-system at the architecture's heart, which is allowing to process data in a stream processing manner and additionally is useful for messaging purposes in the communication between services. Kafka is concretely used in Section 4.5, which is covering the core functionality of collecting, processing and accessing the measured data. We are additionally providing details about authentication mechanisms for dealing with user identities in a distributed environment in Section 4.6. The remaining Sections briefly describe ideas useful for implementing the other contexts.

Integration of services beyond context boundaries is possible at defined interfaces via HTTP or RPC in the backend but should be kept at a minimum and make extensive use of caching and fault-tolerance. Otherwise a failure in one context might harm the functionality of another context, which results in a fragile application architecture. Certain integration can happen by asynchronous messaging via Kafka topics. However, where possible, we favor integration of data from multiple contexts in the frontend or at a specific API-Gateway that unifies endpoints of multiple services for specific use cases. For example, consider the frontend-task of showing a user profile, consisting of personal information, contributions to discussions and awards. According to our bounded contexts specified in Table 4.1, this would require data from social and incentive bounded contexts. We can load the data in the frontend separately from the two different endpoints and automatically graceful degrade<sup>1</sup> when one of the services has failures, because we still have something valuable to show to the user. We can either ignore the failure silently or show a warning to the user that something has gone wrong on one part of the data.

---

<sup>1</sup>Graceful degradation is the ability of a system to stay at least partly functional when some parts of the system break down.

## 4.2 Overall Architectural Design

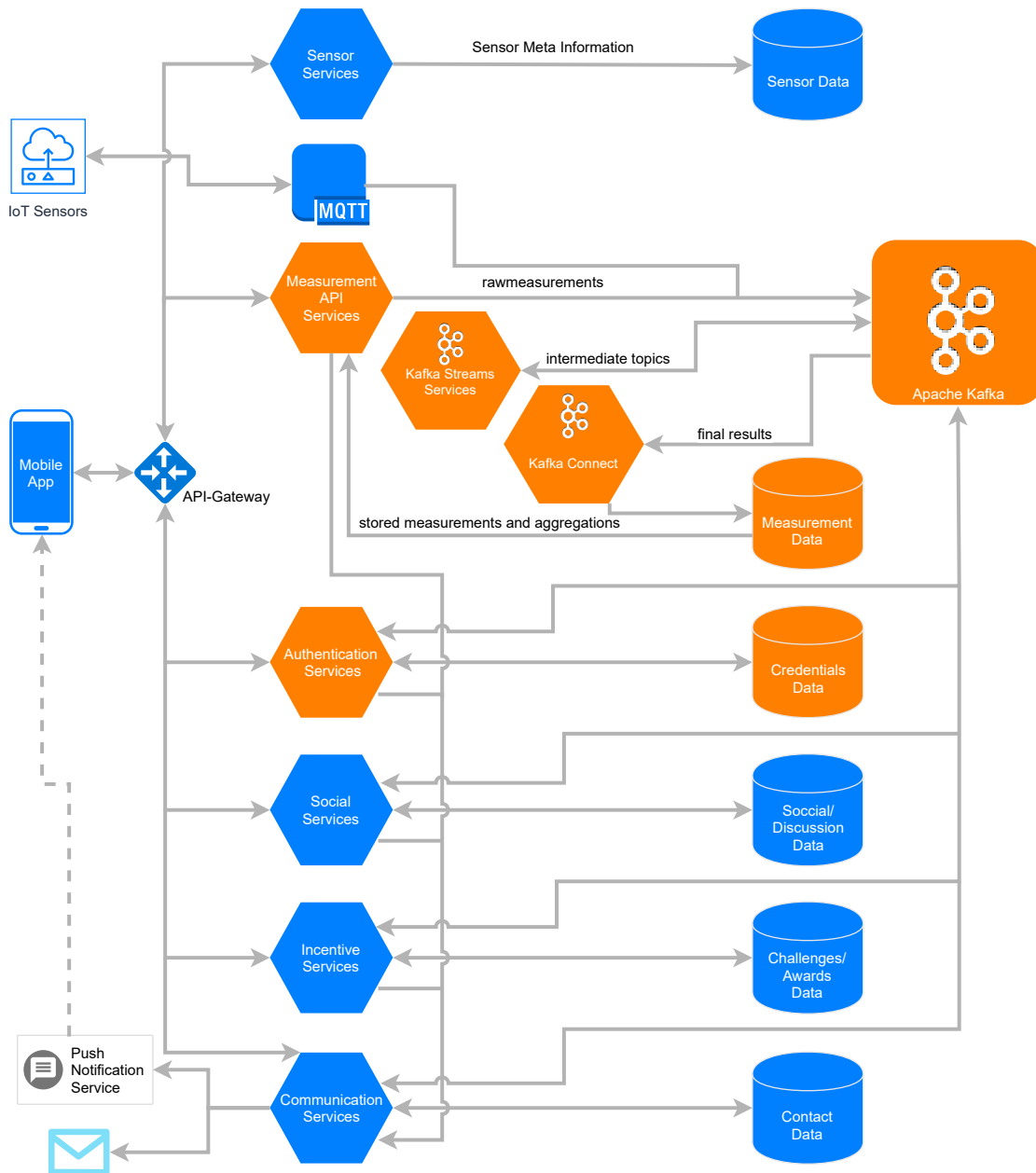


Figure 4.1: Graphical representation of the overall architecture. This work provides detail about orange components, the blue components are only described briefly for completeness.

## **4.3 Fundamental Infrastructure and Automation Principles**

The architectural fundamentals of the backend are aligned to the summarized cloud-native pillars described in Section 2.4.5.1. A good starting point to get an overview about cloud-native technology, tools and services, that can be used to design the overall architecture, is the cloud-native landscape showed in the Appendix A.1. The different categories presented there cover distinct functionalities in a cloud-native environment and often rely on the capabilities of underlying categories.

Our goal is to have a cloud-native application set up as a collection of self-contained microservices, running as containers in a horizontally scalable cluster of computing-nodes provided by a cloud-provider. We approach the architecture from the bottom up, starting with the infrastructural fundamentals.

### **4.3.1 Infrastructure and Provisioning**

In theory, every type of computing-node that can run containers conforming to the OCI specification is usable as the underlying computing primitive. Even dedicated non-virtualized machines would be suitable for a local development cluster, although with certain drawbacks regarding automatic provisioning and state-management due to the lack of storage APIs.

To avoid manual and repetitive steps in the infrastructure setup, we recommend to use tools specifically designed to support automatic provisioning of computing-nodes. They use declarative code, that is specifying operating system configuration and dependency requirements, to perform the necessary installation steps autonomously. Section 5.1.2 is describing how we provisioned the infrastructure of our integration cluster.

### **4.3.2 Cluster Runtime and Orchestration**

The next layer above the basic infrastructure is the cluster, which is the essential system for automatically schedule cloud-native applications across the nodes forming the cluster. It has important responsibilities such as providing persistent storage for

### 4.3 Fundamental Infrastructure and Automation Principles

stateful containers or creating a cluster-internal network for communication between containers. But most importantly it should provide a runtime to run the containers representing our self-contained deployment units.

To reduce the effort of getting a suitable cluster, many cloud-providers have developed PaaS products, that provide an already configured and usable cluster. For example, Amazon has the *Elastic Container Service for Kubernetes (Amazon EKS)*, Google has its *Kubernetes Engine (GKE)* and Microsoft has *Azure Kubernetes Service (AKS)*. They all have in common, that the consumer specifies the size of your cluster, for instance the number of computing-nodes with corresponding specifications for CPU, Memory and Storage. The cloud-provider offers API-access and often a graphical user-interface to the personal cluster, which is running on the reserved computing-nodes. The cluster and its coordinating resources however, is maintained by the provider. At the time of writing, Amazon and Microsoft are even able to completely abstract away the underlying computing resources with *Container as a Service (CaaS)* offerings. Using CaaS, a customer only specifies the resource requirements of a specific container. The provider handles the placement, orchestration and underlying resources transparently in a generic cluster. Obviously because of the greater management effort for the provider such services would usually become more expensive than a well-sized regular cluster.

We will be using *Kubernetes* for orchestrating containers across the cluster, because it is the industries state-of-the-art and has a very good integration with other tools in the cloud-native context.

#### 4.3.2.1 Kubernetes

Kubernetes (K8S) has been originally developed internally by Google and later open sourced as an orchestrator for containerized applications [15]. Because K8S provides us with tooling to build reliable and scalable distributed systems it is a perfect fit to deploy cloud-native applications in form of microservices.

The overall architecture of K8S is composed of master-nodes and worker-nodes as shown in Figure 4.2. The core component is a distributed key-value store called *etcd*<sup>2</sup>,

---

<sup>2</sup><https://coreos.com/etcd/>

## 4 Backend Design

which is used as storage for cluster state. All of the cluster management is performed automatically by constant alignment of two state abstractions. A large part of Kubernetes' idea is built around reconciling those two states [8]:

- *Desired State*: Describes which resources should be deployed including correct configuration and scheduling restrictions. This is the state that corresponds to a declarative description we provide to Kubernetes via deployment descriptors.
- *Actual State*: Describes the current state of the overall cluster, which is collected by K8S system components. All central components are constantly monitoring their state and try to reconcile with the desired state.

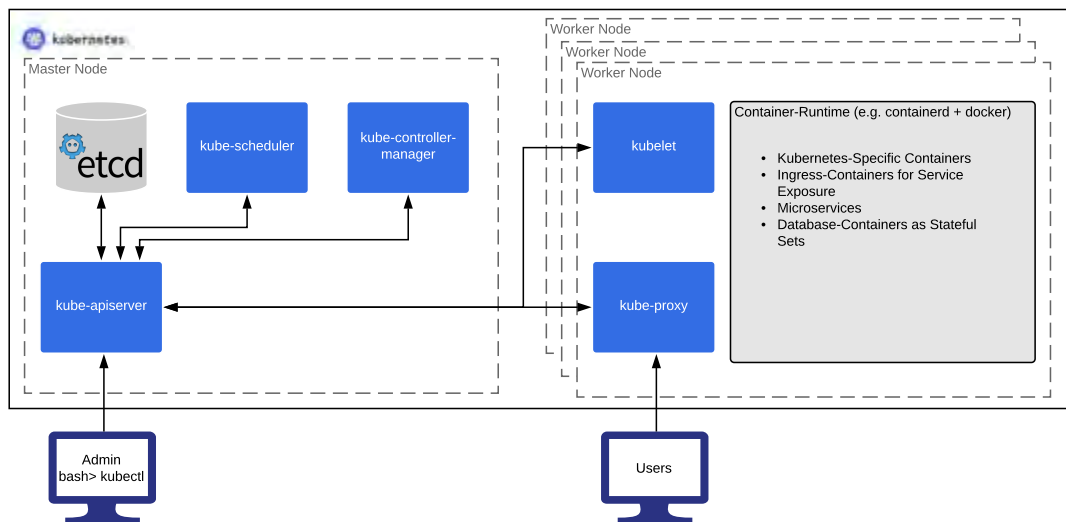


Figure 4.2: Internal architecture of Kubernetes. Based on a drawing from the official documentation [44].

**A Pod** is a K8S specific notion and instead of single containers, the primary entity for scheduling. It describes a package of one or multiple containers, that are closely related or required to work closely together. K8S will make sure, that containers forming a pod are scheduled and replicated together to minimize network latency. However, in most use cases pod is equal to exactly one container.



### 4.3 Fundamental Infrastructure and Automation Principles

**Master-nodes** are the coordinating part of the cluster and are persisting information, that describes the desired state of the cluster in *etcd*. In a larger cluster these nodes are typically dedicated computing instances, which are not considered by the scheduler when actual application workload has to be placed within the cluster. Additionally, they are replicated for fault-tolerance reasons by leveraging the distributed consensus capabilities of *etcd*. Besides *etcd*, the master-nodes are hosting other important components:

- *Scheduler*: Is responsible for distributing the pods across the worker-nodes in the cluster. The scheduler makes sure that each pod is getting the resources needed and that replicas of pods are distributed between the worker-nodes. It does that by annotating the desired state with information that binds the pod to a specific node. This information is posted to the API-server and persisted in *etcd*.
- *Controller-Manager*: This component is actively working with the API-Server to check if at any given time the number of pods is equal to the desired number. If not, it will trigger the scheduler component to create pods that fill the gap.
- *API-Server*: This is the point, where all internal components and external tools can retrieve and change the state of the cluster. Posting our deployment descriptions to the API-Server will tell Kubernetes how a deployment of our application should look like and how many replicas of each component we would like to have.

**Worker-nodes** are only running two small components, in order to save computing capacity for actual application workloads. The *kube-proxy* is responsible for managing the host-network and publishing specific ports of services for external access. The more important component is *kubelet*, which connects the worker to the master-node. It is synchronizing with the API-Server, to gain its part of the “desired state”. The remainder of the worker is left to a container runtime engine, such as *containerd*. That engine is essentially responsible for pulling container information from a registry, running the containers and providing metrics about the runtime-state back to *kubelet*.

**Deployment of services** to Kubernetes can happen with the help of a command-line tool called *kubectl*. As Kubernetes follows a declarative approach, we must specify all

## 4 Backend Design

resources, that should be deployed in the cluster, in structured *YAML*-Files (see Listing A.3 for an example) and apply them utilizing the *kubectl* tool. As the example flow in Figure 4.3 shows, K8S recognizes the desired resource and automatically configures itself to provide the desired service.

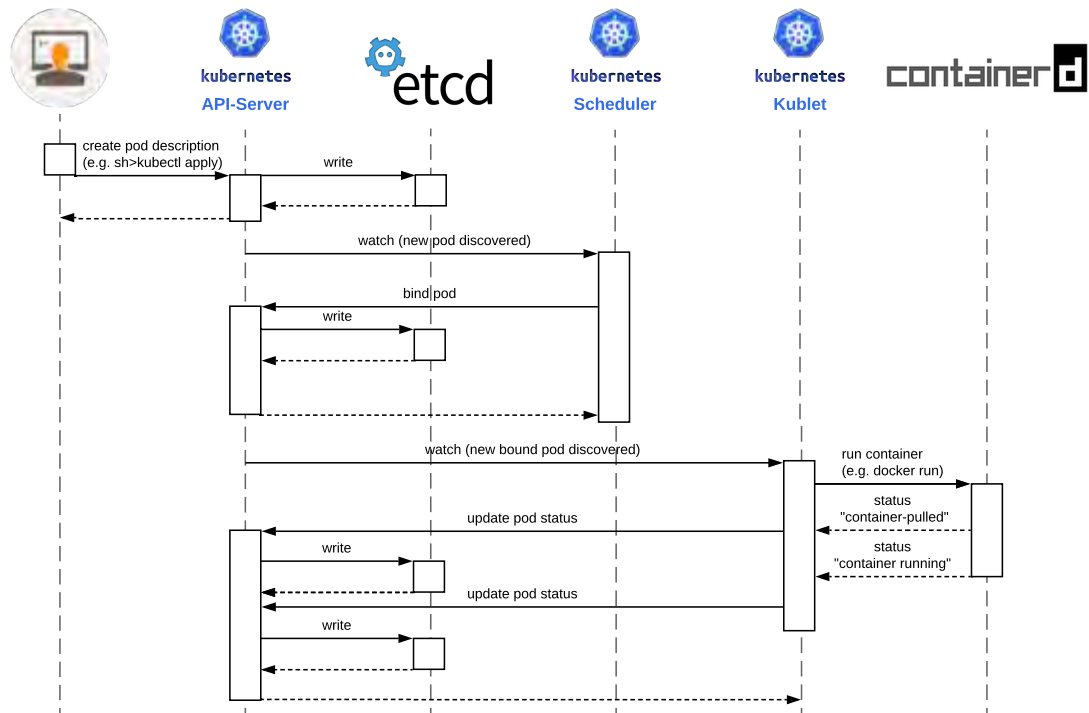


Figure 4.3: Example flow of information through Kubernetes for creating a pod. Based on a diagram in [9].

### 4.3.3 Version Control and Image Builds

While developing software, it is essential to version code changes. This is especially helpful when multiple developers work on the same software, because merging two features often lead to broken versions of the software. *Git* is the most popular tool to version software and additionally a safe way of developing in branches that can be

merged together. There is usually one master-branch into which feature-branches are merged after a feature is completed.

Because we develop a set of independent microservices, we are facing the decision between using one “Mono-Repository<sup>3</sup>” for all microservices or multiple repositories, where each microservice’s code is separated in a distinct repository.

“Mono-Repositories” are highly beneficial in terms of management effort because there is minimal overhead in configuring the repository and managing user-access. Every developer can simply clone the repository and has access to all the code relating to the project. Additionally, code-sharing would be easier, because libraries can be managed within that single repository and is accessible by referencing the corresponding library directory. However, they limit configuration possibilities needed for independent microservices and additionally allow developers to make changes beyond their assigned context. This will inevitably introduce coupling of contexts in the long-term, because developers will come up with the easiest workaround for their specific problem and therefore create a large distributed monolith, when certain boundaries are not enforced.

Our approach uses individual repositories for every microservice with additional project and group abstractions. This approach enables us to define fine-grained user-access on project and group-level, but more importantly we are able to configure independent pipelines and have independent image-registries, that can be used to realize automated continuous delivery, which is one core part in a cloud-native microservices-style.

#### 4.3.3.1 Continuous Delivery

Continuous Delivery has been introduced by Humble et al. as an extension on *Continuous Integration*, which in essence is “the practice of working in small batches and using automated tests to detect and reject changes that introduce a regression [39]”. It is a concept preventing anti-patterns like deploying software manually in scheduled release plans or deploying to production environments without ever having operated the application in a production-like environment at first.

---

<sup>3</sup>All of the source code is kept in one large repository. Code for different services is often structured via subfolders within that repository.

## 4 Backend Design

Instead, every code change that is committed into the repository is validated and subsequently built into a deployable artifact, which is possibly tested against certain requirements and optionally automatically deployed into a development-, integration- or even production environment.

### Note

This is one point where using container technology, such as Docker, is advantageous over using virtual machine (VM) images or software-packages as deployment artifacts. Building a new VM-image for every code change is not feasible because of their size, which is sometimes 10x as large as a container-image. On the other side, software-packages, such as *.jar* files or executable binaries, lack the defined environment allowing to use independent library versions and configuration, that is required for self-contained deployment units. Additionally, smart layering technology of certain container formats is beneficial in terms of image persistence, because only the piece within the container that has changed is persisted as a new layer, which sometimes only account for a few megabytes.

Many GIT-hosting platforms, such as Gitlab, allow to define custom pipelines for a continuous deployment concept, which is described in the following steps:

1. Once a user pushes or merges a new change into the repository, a pipeline-run is triggered and performs the steps defined in a *gitlab-ci.yml* file.
2. In a first step the pipeline performs optional unit-tests and a build of the software to gain an executable software-package. This can be either a binary or a *.jar* file. If the build fails, we instantly stop the pipeline because the changes probably introduced faults into the software.
3. If build and tests are successful another step is automatically creating a new version of the docker container for that service, which can be used as a deployment ready artifact representing our self-contained deployment unit. The image is uploaded into the provided container-registry<sup>4</sup> and tagged with the hash of the commit to uniquely identify the image.

<sup>4</sup>A hosted service for storing container-images. Docker instances on various machines can then use the service to pull the images and run them on their machine.

#### 4.4 Apache Kafka - A Distributed Streaming Platform

4. If the change has been on the master-branch or the pipeline-run is caused by a release, we automatically trigger a service upgrade in our Kubernetes cluster.

If anything occurs suspicious after the deployment, we can easily rollback by running the deployment step of the previous working version. With this flow, we can use a simple *GIT*-command to deployment a new version into our integration environment.

#### 4.4 Apache Kafka - A Distributed Streaming Platform

A large part of our platform is about collecting and shifting data around. If many users take part in our Crowdsensing initiative, we might have to deal with a lot of data, requiring a scalable approach for data processing. One system currently emerging as the industries standard for dealing with Big Data in the IoT is *Apache Kafka*, from here on just called “Kafka”. Used as a common component, it can provide decoupled information exchange between different services in an asynchronous manner. Services can publish certain events to Kafka topics and every interested service can act as a consumer that polls messages from such topics. The real benefit however is, that libraries such as *Kafka-Streams* and *Kafka-Connect* allow an easy implementation of Stream-Processing, introduced in Section 2.2.3. This approach is helpful in order to streamline the data coming from smartphones and IoT-devices for a subsequent aggregation of that geospatial measurement data.

The official documentation [1] and the book by Narkhede et al. [56] describe the internal fundamentals best. Kafka is based on a distributed commit log that is maintained by a cluster of “*Brokers*” coordinated by *Apache Zookeeper*<sup>5</sup>, a system capable of forming a consensus in a distributed and fault-tolerant manner, a precondition for Kafka’s scalability and replication capabilities. As Zookeeper is mostly used to store the configuration state of the Kafka-Cluster, such as how many Brokers, consumers and topics are there, it is not exposed to heavy load. However, professional deployments should nonetheless consider to deploy Zookeeper in replicated fashion to prevent losing information about the Kafka-Cluster state.

---

<sup>5</sup><https://zookeeper.apache.org/>

### 4.4.1 Distributed Commit Log

The distributed commit log is the fundamental abstraction in Kafka and presented in more detail in Figure 4.4. The Figure is showing a graphical overview of how Producers, Consumers, Consumer-Groups and the distributed commit log are related to each other. The core notion of a distributed commit log is based on the following three pillars:

**Message** Is a generic key-value data record with additional meta-data belonging to an offset in a certain partition of a *topic*. It is somehow comparable to a row or record of a database table.

**Topic** is an abstraction for messages of the same semantic meaning, similar to a database table for rows. They can be separated into multiple partitions based on a partition strategy, which is defaulting to use the message-key.

**Partition** Is a single commit log of a topic and represents a distinct subset of all messages in a topic, since a message is only appended to exactly one partition. The concept is important for Kafka's scalability and redundancy, because each partition of a topic can be hosted by a different Broker. This distribution spreads messages of a topic across several Brokers, that together hold all messages and provide them to a set of consumers accordingly.

The distributed commit log is essentially based on multiple partitions, that each on its own represent a single commit log of append operations for messages. Because there are no transactions, Kafka can only provide message-ordering guarantees within a single partition, but not across the distributed commit log of a whole topic. Since our use case of averaging noise-measurements is a commutative operation, this drawback is acceptable.

However, if a global ordering of events is critical across partitions, such as a topic wide knowledge about which event happened before another, the implementation must take further measures. For example, a non-commutative matrix multiplication with messages from multiple partitions can issue indeterministic results, since there are no guarantees for global ordering and the result of the operation will depend on the order, in which messages are read from different partitions by the consumer calculating the product.

### 4.4.2 Components

From an architectural viewpoint Kafka is a regular client-server system, where the server is usually a cluster of Brokers and the clients are Producers and Consumers or higher-level clients such as Kafka-Streams or Kafka-Connect.

**Brokers** Are responsible for receiving, storing, replicating and responding messages.

Within a cluster, one Broker is the cluster controller. Additionally, all Brokers are leaders for a distinct set of partitions as mentioned before. For redundancy reasons however, it can be the case that a Broker is as well holding replicas of a partition led by another Broker.

**Producer** Is responsible for creating messages in specific partitions of topic. The exact partition for a message is usually determined by a hash strategy of the message key, to guarantee that all messages with the same key will get assigned to the same partition.

**Consumer** A consumer is assigned to a set of partitions of a topic and is reading messages from them. It is maintaining on its own, which offsets it has already read from a partition. Multiple Consumers are managed in Consumer-Groups, which are an abstraction to divide the responsibility for partitions of a topic between Consumers, which are then only responsible for a distinct subset of partitions.

**Kafka-Streams** Is a common Java Library that provides a high-level Domain Specific Language to easily create a topology of transformations, joins and aggregations on Kafka topics. The Library transparently handles consuming and producing of messages and therefore allows to implement a stream processing pipeline, using the Kafka-cluster as a means to shift data between certain pipeline steps.

**Kafka-Connect** Is a runtime directly implemented into Kafka for running connector-plugins, that allow a scalable and reliable way of moving data between Kafka and other datastores. This allows to easily inject data already present in conventional datastores into Kafka to have it usable for stream processing or to persist data from Kafka into conventional datastores or clusters such as Hadoop for analytical reasons.

## 4 Backend Design

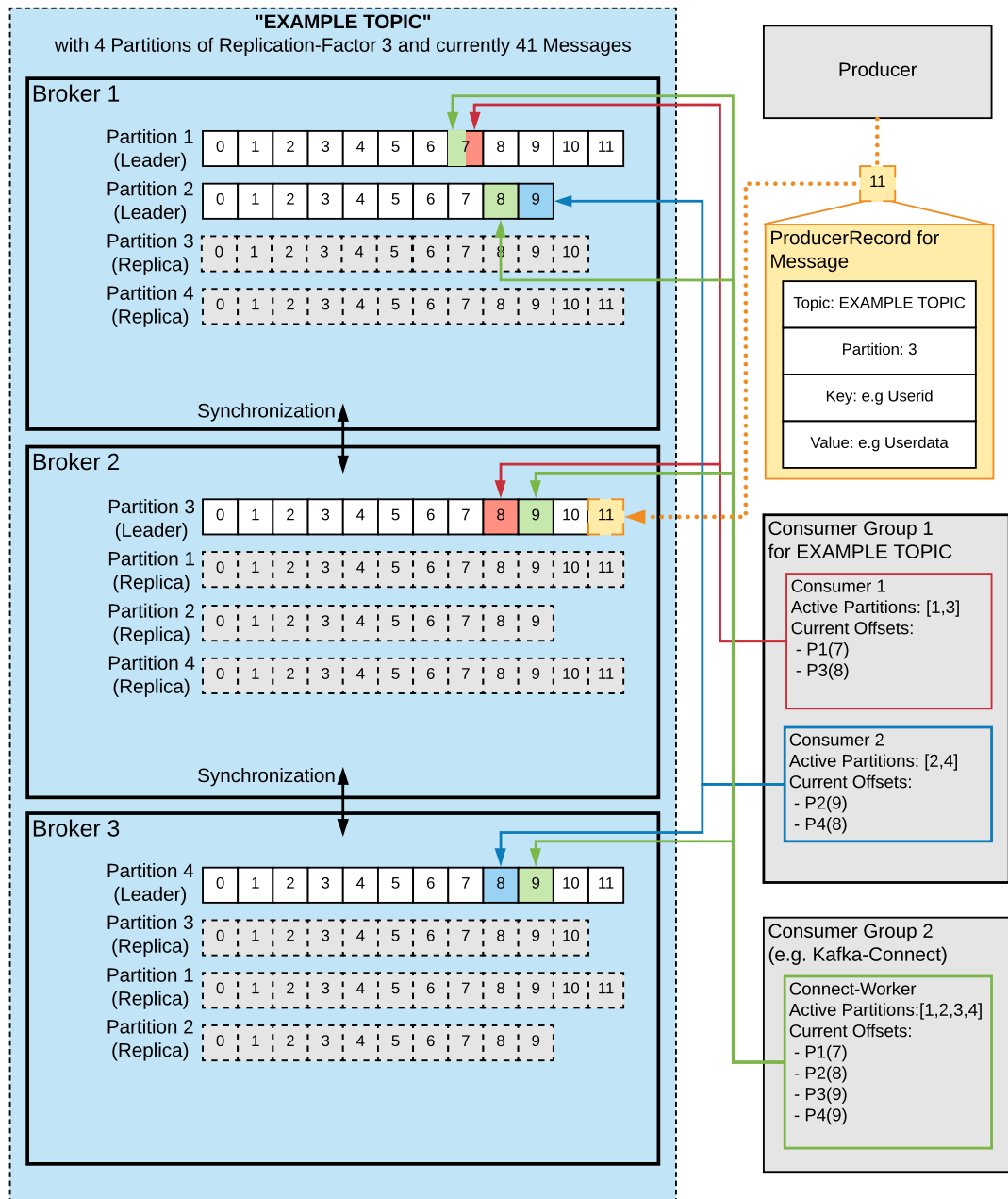


Figure 4.4: Structure of the distributed commit log of an “Example Topic” and the corresponding Producer and Consumer dependencies. Image is based on multiple drawings in [56].



The Figure in 4.4 shows a Kafka cluster with three brokers holding an “Example Topic” with 4 partitions and a replication-factor of 3, which will distribute 2 replicas of each partition across the brokers in the cluster. A producer is creating a “ProducerRecord” to append a message to the distributed commit log. Based on the given “Userid” it has previously determined the corresponding partition using a certain strategy. It will append the message to partition 3 at offset 11 on the end of the partition’s commit log in broker 2. This partition is currently covered by two Consumer-Groups with “Consumer 1” and a “Connect-Worker” for Kafka-Connect, which will both read all messages in the order of commit-offsets to constantly catch up with the end of the partition’s commit log. As shown, in “Consumer Group 1” two consumers are active and splitting the responsibility of partitions in a non-overlapping fashion.

## 4.5 Bounded Context: Measurements

Measurements are the core element of our system and hence this bounded context is more sophisticated compared to the others. In fact, it is internally divided into multiple microservices. The general responsibility of this context is to ingest measurements, pre-process and aggregate the measurement-records and finally provide access to the end results for visualization purposes. The business problem of collecting measurements is strongly related to IoT scenarios and Big Data, which is why we have to prevent conceptual failures regarding data-volume, processing-time and protocols. The following Sections provide further elaborations on our approach.

### 4.5.1 Volume and Variety

As we described in Section 4.2 our design must be able to deal with data coming from a variety of sources, which are not necessarily using HTTP. Additionally, we must assume that many users ingest massive amounts of data into the system.

### 4.5.1.1 Problem

In the World Wide Web, a request-response approach based on HTTP is typically used to post data to a system. The data is usually validated, possibly transformed and subsequently persisted in a database or file-system. When the performed validations are no longer simple filters but rather sophisticated statistical approaches, like calculating a standard deviation of a set of measurements from a specific region within a time window of 5 or 10 seconds to filter runaways, the round-trip time of a post request might however be of magnitude seconds, rather than a few milliseconds. This would not only be bad in terms of user experience, but rather more severe for IoT sensors focused on low-battery consumption. Indeed, such sensors often are not even powerful enough to communicate via a document-centric HTTP protocol and instead use more lightweight data centric protocol such as *Message Queuing Telemetry Transport (MQTT)*<sup>6</sup>, which transfers data as byte arrays using a publish-subscribe model. Our data ingestion strategy must be able to deal with these protocols as well and can therefore not solely rely on an HTTP endpoint for data ingress.

The data access latency might additionally be negatively affected by sophisticated ingress validations and a general constant load produced by incoming measurements. The other way around, high load on data access because many users are viewing visualizations would affect the data ingress response times and can potentially result in timeouts and therefore losing incoming measurements.

### 4.5.1.2 Solution

Our designed solution decouples the data-ingress and data-access into two distinct microservices and therefore making them load- and error independent. Failures because of implementation errors or overload in either ingress or access is not affecting the other part of the system. To maintain low latencies, each measurement that is posted to the ingress-service gets accepted after a short format-validation and the service is

---

<sup>6</sup>MQTT is a messaging-based protocol requiring a broker as a global database for every client's communication state. It enables M2M communication with low-performance devices over high latency connections with limited bandwidth.

responding to the submitting device immediately, after it has published the data to the Kafka topic for raw measurements.

Since measurements coming from an IoT sensor via MQTT might be in a different data-format, we have to streamline them with the measurements provided by smartphone users. Differences can be attributes, such as a sensor id rather than a user id. The streamlining should happen in a first stream processing step, to allow further check- and validation steps on a common data format.

### 4.5.2 Aggregation

In order to gain knowledge from the collected data, a user must be able to view and visualize the data on his device, which will likely be the smartphone running the associated mobile application. Interesting aspects of such data might include averages and min- and max values for areas of different sizes or within certain time windows as the functional requirements in the Noisemap project show in Section 3.6.1. We have to provide a variety of visualization options, which is requiring a summarization of measurements.

#### 4.5.2.1 Problem

While performing aggregation on the frontend device is absolutely reasonable, this approach is limited to visualization for a small set of measurements, because each has to be transferred to the device. At a certain number of requested measurements, the data to transfer reaches a size, that would affect not only the data-quota of a mobile user, but also results in increased latency from initial request to finished visualization. That latency is composed of multiple effects large set-sizes bring along. Costly database operations, JSON-serialization, data transfer over sometimes limited mobile bandwidth and finally JSON-deserialization as well as possibly complex aggregation and visualization on a mobile device with limited power are some of them. Therefore, the classical way is to provide the frontend with already aggregated results for visualization. Although a substantial improvement for large sets, the conventional approach of just storing raw measurements in a database and aggregate them on the fly for each visualization

## 4 Backend Design

request has drawbacks. Getting the matching measurements from the database is still transferring a large number of measurements between the service and the database for each request. Additionally, the set of measurements to be aggregated can also reach a size, where the aggregation would affect the response time of the request negatively.

### 4.5.2.2 Solution

To limit the load on the system and provide data for visualization rapidly, we apply certain preprocessing to have already aggregated results in the database, that can either directly be responded on a request or used as an intermediate result to apply further aggregation. The preprocessing assumes, that in general the measurements provided are conceptually just a continuous stream of sensor data making it suitable for *stream processing*. We have introduced the Kafka-Stream library to build simple microservices capable of performing stream processing without having an additional cluster. This is beneficial over other frameworks such as Apache Flink or Apache Spark.

Figure 5.3 in the prototypic implementation shows, how the stream processing phase uses simple microservices that are polling single measurements thereby in different steps calculate different aggregates of certain time windows and geospatial areas. These results will be stored in the database and can be accessed and returned to the user directly. This avoids numerous similar aggregations of hundreds of thousands of records for every request on the fly.

### 4.5.3 Proposed Internal Architecture

The approach can be described by the following three phases:

- **Data Ingress:** In this phase either a smartphone will post data an endpoint, which is handing the data over to Kafka, or it is coming from IoT-Sensors via MQTT and is directly ingested into Kafka.
- **Stream Processing:** This phase represents the actual processing business logic. Several validation checks can be made, such as filtering measurements with

nonsense or unrealistic values. The measurements will then be split into multiple data-fragments for privacy reasons. Further, geo-temporal information from measurements will be used to calculate average values for different sizes of time windows and areas. The last step of this phase offloads the data into a database using Kafka-Connect.

- **Data Access:** The data-access phase is decoupled from the other phases, since it only accesses data already available in the database. Multiple database-indexes allow to query the data efficiently and filters specified in request-parameters narrow the matching result set.

The graphical representation of the phases in Figure 4.5 show how Kafka is used as a central data hub, where ingress-services put their data, stream processing services get their input from and put their results to and connectors consume the finished results. Kafka although, is grayed out, because we do not see it as a context internal component, but rather an infrastructure service that is provided within the cluster and usable by any service regardless of the context.

### 4.5.4 Geospatial Data-Handling

One central part of the measurement context is geospatial data. The geographic relation of the measurement data is important for aggregating the right data together, but also to visualize the data on a map. At first, we need a format to store and transfer data with a geospatial context. We would like to introduce GeoJSON for that purpose. Since it is state-of-the-art to use JSON for public REST APIs, GeoJSON can be seamlessly integrated and is additionally supported natively by many visualization- and mapping libraries used in frontend-applications.

#### 4.5.4.1 GeoJSON Format

The RFC7946 [31] specifies GeoJSON as a data interchange format for geospatial data. It is based on *Features* and *FeatureCollections* belonging to the OGC Simple Feature Access Specification introduced in Section 2.3.2. The concepts are derived from other

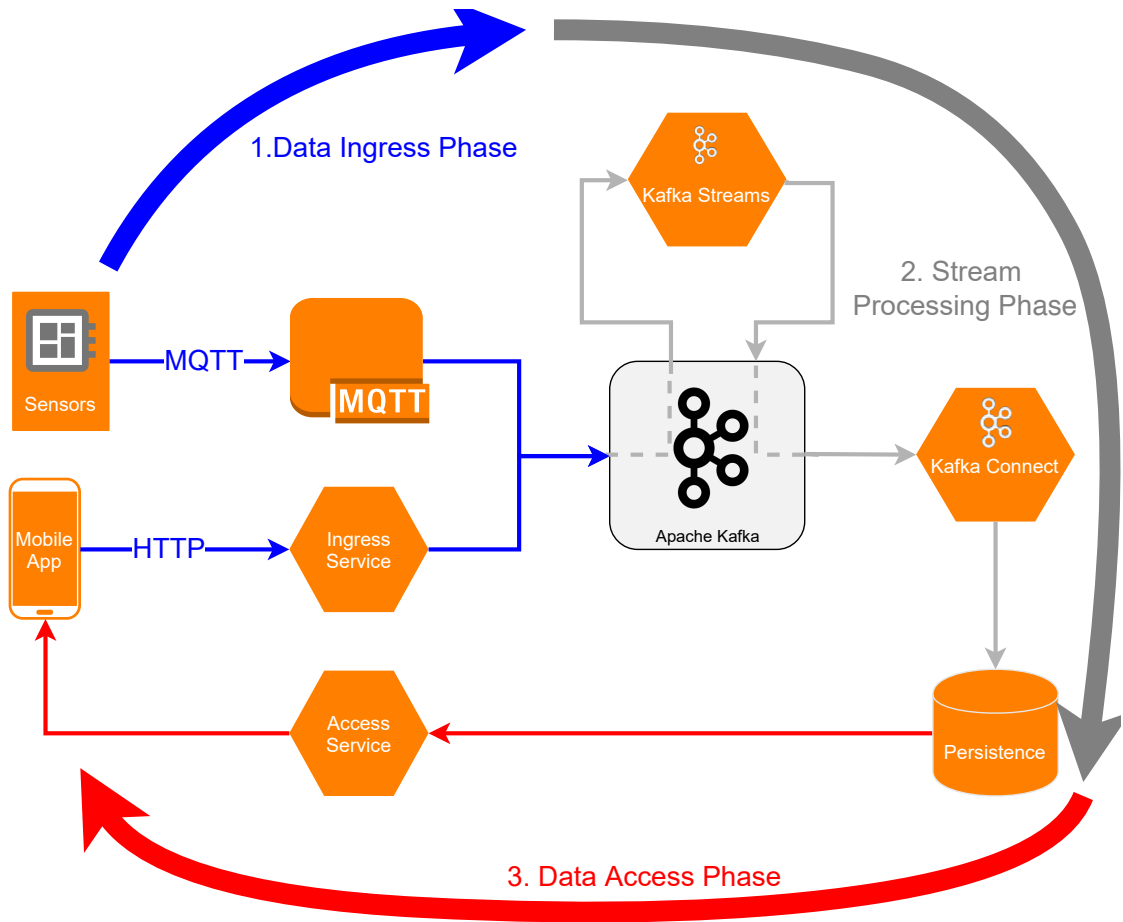





Figure 4.5: Conceptual internal architecture of the Measurement Context showing different phases for data processing and access.

formats, but have been adapted to be used with JSON, which better suits modern API development. In GeoJSON, a *FeatureCollection* is a JSON-Array of *Feature* objects, where one such *Feature* object has an attribute *type* with the value *Feature* and an attribute *geometry* with the value being a *Geometry object or null*. A *Geometry* object is of one of the possible types shown in Table 4.2. The fundamental construct however, is the notion of a *Position* array, which is the basic element for the *coordinates* attribute of each geometry type and has a maximum of three elements. The order of them is *longitude, latitude, elevation*, but the elevation is usually skipped for two dimensional contexts. A *coordinates* attribute can be differently shaped depending on the type of

geometry it is attached to. For a *Point* it is a usual *Position*, for *MultiPoint* and *LineString* it is an array of *Positions* and for a *MultiLineString* or *Polygon* it is represented as an array of *arrays of positions*. For a *MultiPolygon* however, the *coordinates* attribute is an array of multiple Polygon's *Coordinates* and for the *GemoetryCollection* more generically and array of multiple coordinates of several types.

Table 4.2: Geometric Types supported by GeoJSON.

Type	Example JSON	Graphical
Point	<pre> 1 { 2   "type": "Point", 3   "coordinates": [ 4     9.956971, 48.422899 5   ] 6 }</pre>	
MultiPoint	<pre> 1 { 2   "type": "MultiPoint", 3   "coordinates": [ 4     [9.956971, 48.422899], 5     [9.95528, 48.421694] 6   ] 7 }</pre>	
LineString	<pre> 1 { 2   "type": "LineString", 3   "coordinates": [ 4     [9.956971, 48.422899], 5     [9.95528, 48.421694] 6   ] 7 }</pre>	

continued ...

#### 4 Backend Design



Table 4.2 continued: Geometric Types supported by GeoJSON.

Type	Example JSON	Graphical
MultiLine-String	<pre>1 { 2   "type": "MultiLineString", 3   "coordinates": [[ 4     [9.956971,48.422899], 5     [9.95528,48.421694] 6   ], [ 7     [9.956971,48.422899], 8     [9.957479,48.422905] 9   ]] 10 }</pre>	
Polygon	<pre>1 { 2   "type": "Polygon", 3   "coordinates": [ 4     [ 5       [9.956971,48.422899], 6       [9.95528,48.4223494], 7       [9.955979,48.4217], 8       [9.956971,48.422899] 9     ], 10    [ 11      [9.95595,48.4218], 12      [9.95682,48.42281], 13      [9.9555,48.422271], 14      [9.95595,48.4218] 15    ] 16  ] 17 }</pre>	

continued ...



Table 4.2 continued: Geometric Types supported by GeoJSON.

Type	Example JSON	Graphical
MultiPoly- gon	<pre> 1  "type": "MultiPolygon", 2  "coordinates": [[ 3    [[9.956971,48.422899], 4     [9.95528,48.4223494], 5     [9.955979,48.4217], 6     [9.956971,48.422899]], 7    [[9.95595,48.4218], 8     [9.95682,48.42281], 9     [9.9555,48.422271], 10    [9.95595,48.4218]] 11  ],[ 12   [[9.956327,48.421807], 13    [9.95782,48.4226], 14    [9.9580868,48.422], 15    [9.956327,48.421807], 16    [9.956327,48.421807]] 17  ]] </pre>	
Geometry Collection	<pre> 1  "type": "GeometryCollection", 2  "geometries": [ 3    {"type": "Point", 4     "coordinates": [ 5       9.956971,48.422899]}, 6    {"type": "Polygon", 7     "coordinates": [[ 8       [9.956327,48.421807], 9       [9.95782,48.4226], 10      [9.9580868,48.422], 11      [9.956327,48.421807], 12      [9.956327,48.421807]]]} 13  ]] </pre>	

**Polygons** take an important role to represent areas on surfaces and because of a missing notion of circles in GeoJSON, they are used to approximate curved shapes like a circle or an ellipse. Since our geospatial indexing library described in Section 4.5.4.2 is based on hexagonal polygons, we are mostly going to use the Polygon type

## 4 Backend Design

aside from regular points. In order to be able to exclude areas within a polygon, the coordinates are specified as an array of arrays of positions. Each array of positions is a closed LinearRing, where the first and last element must be equal, which is why the boundary of a triangular polygon needs four position elements. The first LinearRing of a Polygon's coordinates attribute is the outer boundary and the rest of the LinearRings are holes, which are excluded from the area of the holistic polygon [31].

### 4.5.4.2 Geospatial-Indexing

When it comes to the question of which records should be aggregated together, we can answer differently depending on the specific use case. According to Section 2.3.3 we can use manually defined polygons and aggregate together all the records which are contained in that polygon. However, this is only suitable and feasible with reasonable effort, for a narrow number of custom defined areas and a limited granularity. Since we rather want the platform to work universally worldwide, a more general solution based on Geodesic DGGs is considered here as a viable option to assign measurements to geographical buckets. An additional benefit we get from indexing with DGGs, is an index for each bucket that can be easier handled than geographic coordinates, can act as a key for Kafka topics and a simple database index.

**H3**<sup>7</sup> is a “hexagonal hierarchical geospatial indexing system” based on DGGs, provided by Uber. It has originally been developed for optimizing ride pricing and dispatch but is now available as open-source. The system uses a hierarchical index based on hexagons and allows to allocate single data points into buckets assigned to different hierarchical layers. Each bucket is represented by a hexagon and has a unique index key. A detailed description of H3's theoretical and practical background is provided by the original Uber engineers in a blog post [14]. The basic functionality is transforming a given latitude and longitude of a point together with a specified resolution between 0 and 15 into a 64bit H3 index, that identifies a specific grid cell of the DGGs and is represented as a hexagon. The resolution is determining the size of the hexagon. The real-world

---

<sup>7</sup><https://uber.github.io/h3>

#### 4.5 Bounded Context: Measurements

hexagon's edge length reaches from 51cm at resolution 15 and ends at resolution 0 with a length of 1,107km. The layer in resolution 15 is divided into 569,707,381,193,162 small hexagons with unique indices, whereas resolution 0 only is composed of 122 large hexagons.

According to the characteristics of a DGGS described in Section 2.3.3, these 122 base cells are spread across the 10 faces of an icosahedron. Since it is not possible to tile an icosahedron only into hexagons there are 12 pentagons at each of the icosahedron's vertices. However, they use a spherical icosahedron orientation, which is the only known, that places all 12 vertices in the water when projected onto the earth's surface, minimizing the effect of the pentagons for applications targeting the mainland.

An example of how we can use H3 can explain the benefits of the library best. It allows us to automatically shard our data points based on spatial distribution and use the provided indices of the buckets for simple data access and as the key for aggregation as well as topic partitioning in Kafka. Before the actual aggregation in the stream processing phase, we use H3 on each single measurement, to gain the index-key in a high resolution. Based on the index-key we know exactly which data has to be aggregated together. They additionally represent the hexagonal geographic area a measurement is contained in and imply, that the measurements getting the same key are close to each other. Figure 4.6 shows an example of geospatial indexing of two locations on the campus of the University of Ulm. At resolution 10, shown in Figure 4.6b the locations are actually

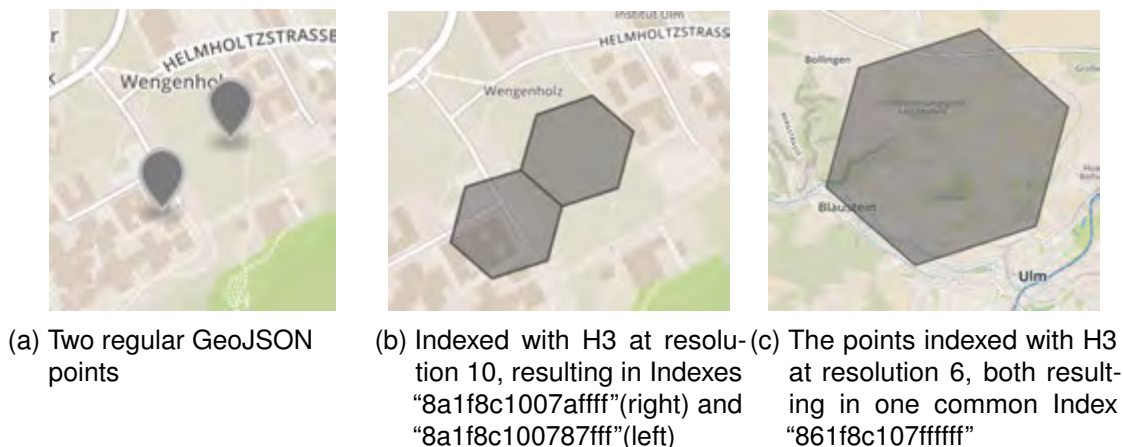


Figure 4.6: Plots of two regular Points and their differently sized H3 hexagons.

## 4 Backend Design

indexed with two different H3-indexes. Figure 4.6c shows the same two points indexed at resolution 6, resulting in one large hexagon that is covering a large part of the area north of the city of Ulm. All measurements in that area would get the same index-key at resolution 6 and will therefore be easily selectable for aggregation.

**The drawback** of indexing based on DGGS is shown in Figure 4.7. The statically assigned hexagons can have certain disadvantages for visualization of phenomena only relevant in a specific region. The hexagon shown, is placed unfavorable for visualizing a certain fact in the city of Philadelphia, because it is cutting the city in two almost equally sized halves. In such a scenario using real city borders can be a superior solution to a DGGS. Data for such borders can be retrieved from sources such as GADM<sup>8</sup> but integrating that data into a stream processing pipeline requires additional steps, such as scanning the GADM data in a database for each provided measurement to find the correct polygon.



Figure 4.7: Plot of a H3 index, splitting the city of Philadelphia in the middle.

### 4.5.5 Storage

The storage solution used for the measurements preferably supports geospatial queries natively. Two fundamental pillars of such a feature is efficient internal geospatial indexing at first and second, the possibility to specify sophisticated queries for different use cases related to geospatially distributed data. One perfectly suitable and battle-tested solution would be the relational database *Postgres* with the *PostGIS*<sup>9</sup> extension on top. However, *Postgres* has certain limitations in terms of scalability related to write-performance.

We already have experience with another solution called *MongoDB*<sup>10</sup>. The database is based on documents and one of the most mature NoSQL databases. It allows horizontal

<sup>8</sup><https://gadm.org> is providing spatial data for all countries and their administrative sub-divisions.

<sup>9</sup><https://postgis.net/>

<sup>10</sup><https://www.mongodb.com>

replication and scalability of the database via ReplicaSets and Sharding of Collections by a defined key. More details about the principles is provided in Section 6.2.2. One drawback of MongoDB is its proprietary API for specifying queries, rather than following the common SQL standard. However, that API has substantial geospatial features that rely on sophisticated geospatial indexes internally used by MongoDB. They can be created on coordinate pairs or GeoJSON-objects, but it is additionally possible to create compound indexes, which are a combination of geospatial indexes and non-geospatial indexes. Particularly the *2dsphere* index is helpful for our purpose, since it supports queries for inclusion, intersection and proximity of different reference geometry types on an earth-like sphere.

##### 4.5.6 Privacy

For users providing measurements there are privacy concerns regarding their location. Every measurement is referenced with the user id, which is critical if such information is being exposed via the API, since one could easily analyze all measurements, match locations to user ids and create a movement pattern of users. However, at the same time it is important for the system to maintain a relationship between users and measurements to enable incentive management, social features and certain personal statistics. Therefore we provide the user with an option to define fine-grained, which type of users should know, that a measurement has been made by that particular user. All users not included in that type will only be able to retrieve anonymized measurements at the API.

The conceptual privacy solution avoids storing user ids together with measurements in the first place. We split user-related information from measurements while processing and store a mapping of user ids and measurement ids, which we can use internally to recreate the relationship. This way, we can securely publish all measurements through our API without exposing critical user information. Although, each requesting user can explicitly tell the system, to check for each measurement in the result set about to be returned, whether the user is allowed to see the creator and populate that information. This however, is a costly operation and should only be used by frontend applications, if that information is actually used, which is not the case at simple visualization.

### 4.5.7 API Access

The API is an essential part, because it allows frontend applications to retrieve data in order to visualize it. It must support geographic bounding, time-based filtering or any form of pagination or it has to enforce certain limitations to prevent denial of service due to overload. Requests without bounding or filtering are therefore either rejected or performed with defaults, such as limitation for a few thousand elements per request.

Single measurements are only viable for small-scale views on streets or neighborhoods but would exceed rational sizes when requesting measurements of a whole city with many active users. To allow visualization of larger areas and broader timescales, the API must provide endpoints to request a set of already aggregated results, which is smaller by multiple factors and therefore much more mobile friendly in terms of quota-consumption and processing power required to visualize the results on a map.

**The downside of pre-aggregated windows** is, that they are unsuitable for real-time visualization. Because pre-aggregating “the last 15 minutes” or the “the last 24 hours” is a problem. The definition of “last” is constantly shifting with time, resulting in the need for constantly recalculating the aggregated result. For aggregation, we are currently focusing on tumbling time windows, which are fixed-sized, gap-less and non-overlapping time windows. Their advance interval is equal to their window-length, which has the effect, that a data-record belongs to exactly one window. For example, we can have 15-minute time windows with the beginning of window  $w_1$  at the full-hour and the beginning of window  $w_4$  at three-quarters [23]. We also can define windows with an interval of one day from midnight to midnight, noon to noon or any other suitable starting-time. The point is, that these tumbling windows are badly satisfying the definition of the constantly shifting “last”. We can only decide between either using the last completed time window or using the current unfinished window, which might be only a few seconds old and is not containing any values yet. A solution might be to use overlapping hopping windows, as defined in [56], which we did not consider because of unknown effects on stream-processors at massive data-volumes.

## 4.6 Bounded Context: User Identity

Handling user identities is a critical part of most information systems. For certain functionalities (e.g. in the Social- and Communication Context) it is inevitable to maintain user identities with at least a username and an email. In the Measurement Context however, user identities are just another way of identifying sources of data apart from unique device-ids of IoT-sensors. For collecting statistics user identities are a better option, because they are easier to transfer between devices and a user will not lose any statistical progress on device change, such as using a new smartphone.

### 4.6.1 User Identification in a Distributed Application

An additional typical use case is authorization, which is important for all systems with a public API. Sometimes even just to prevent fraud and misuse. A high-level view of processing a request to an API can be split in three parts:

1. *Authentication*: Identify the subject (often a user), that is making the request.
2. *Authorization*: Apply checks to authorize the subject to perform the request.
3. *Business Functions*: Execute requested business logic on behalf of the subject.

In a traditional monolithic application all three parts are performed by one single application that has access to all necessary information to authenticate a subject, such as stored credentials. In a distributed environment, where multiple services must authorize requests, there are certain challenges.

#### 4.6.1.1 Problem

All services have to perform the *authorization* autonomously, but at the same time autonomous *authentication* at each service is a bad pattern, because it would require storing the subject's credentials in multiple services [81]. As shown in Figure 4.8, authorization in a distributed system of microservices has to be handled differently compared to a monolithic application. The authentication should be performed in an

## 4 Backend Design

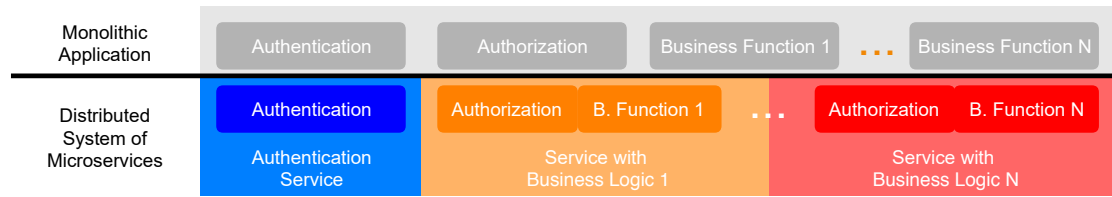


Figure 4.8: Authentication and Authorization in a monolithic vs distributed environment.

isolated authentication-service that can be especially trimmed for security, because it is the only point where credentials are used. However, in order to authorize requests at each of the business services we somehow need to know, that the subject performing the request, has a valid identity. Redirecting each request from the business services to the authentication-service is a suboptimal option, because it introduces additional latency and load in the system. What we want is, that the business services are already presented with a valid authenticated identity, that can be used to authorize the request, without performing a precedent authentication.

### 4.6.1.2 Solution

This issue is solved by using cryptographically signed tokens, that are issued by the authentication-service. That service is the only point where a subject can authenticate and get a testimony, that says “user xyz has been correctly authenticated by our service”. We call that testimony an *Access Token*, which can be used to perform requests at business services. They do not have to perform an authentication again, but rather only validate the token and subsequently perform the authorization based on the information in the token. The identity used in the authorization logic (e.g. a user id) is often encoded into the token in form of claims, that can be relied on after proper validation of the token.

**OAuth 2.0** is the standard tool-set used for the flow of authorization-related information and is specified in RFC6749 [34]. However, it is not a protocol for authentication, but rather a tool for authorizing a *“third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval*



*interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf [34].* An example would be a third-party application for analyzing measurements of a user, that is requesting measurements made by that user at our API. To close the authentication gap, the OpenID Foundation developed the *OpenID Connect (OIDC)* specification as a simple identity layer on top of OAuth 2.0 [57].

Strongly simplified, OIDC and OAuth2.0 know the following main terms:

- *End-User*: Human Participant, typically the Resource Owner.
- *Relying Party (RP)*: In the simplest form an application controlled by the End-User.
- *OpenID Provider (OP)*: A server that is capable of authenticating the End-User and providing Access Tokens to a Relying Party.
- *Resource Server*: The service that has to respond on requests for the protected resource after validating Access Tokens.
- *Claim*: One piece of information about an entity.

Access Tokens are usually *Bearer* tokens. A party in possession of such a token “[...] can use the token in any way that any other party in possession of it can [35]” and does not require any of the parties to proof possession of other cryptographic material. Hence, Access Tokens should be protected from disclosure to unwanted parties, because any party in possession can use the token equally as long it is valid. However, for a cloud-native microservices architecture they give us the benefit, that a service can append the token on subsequent requests to other internal upstream-services, which would then be able to validate and authorize the request in the same manner.

**JSON Web Token (JWT)** are one way of transferring claims between multiple parties. They are specified in RFC7519 [12] and are widely used in state-of-the-art cloud- and mobile applications, where they make up for the missing concept of cookies. These tokens consist of header, payload and signature and are usually sent in the form of an encoded string. Header and payload are cryptographically signed and therefore not changeable without rendering the token invalid. Both, synchronous and asynchronous

## 4 Backend Design

cryptography is suitable for signing the tokens. The decoded content of a typical JWT, representing an Access Token that can be used as a *bearer token*, is presented in Listing 4.8.

```
1 header:{
2   "alg": "RS256", //type of the cryptographic
3   "typ": "JWT" //type of the token
4 },
5 payload:{
6   "exp": 1534350840, //expiration time, how long the token is valid
7   "nbf": 1534347340, //not valid before
8   "iat": 1534347240, //issued at time
9   "iss": "auth_service", //who issued the token (Our authentication-service)
10  "aud": "auth_service", //who issued the token (Our authentication-service)
11  "sub": "5b729f090132e5000db53b42", //subject of authentication (userid for
      Reference)
12  "role": "member", //users role level which is important for authorization
13  "typ": "Bearer" //OIDC the type of the token (Refresh,Bearer, Identity)
14 }
```

Listing 4.8: Decoded Content of our JWT-Token

### 4.6.2 Authentication Example

Figure 4.9 shows an example of information flow of requesting and using Access Tokens. Credentials of users are only processed by an authentication-service, which has access to a distinct database holding credentials information. For signing the tokens, the service uses either a private key or it is using a shared secret with the other services. Validating the token can happen with the shared secret or a public key matching the private key used for signing.

1. The mobile application is presenting an input for the user's credentials.
2. The user enters the credentials and triggers the login flow.
3. a) The mobile application is requesting an Access Token by passing the user's credentials.

#### 4.6 Bounded Context: User Identity

- b) If the user has already logged in and the mobile application is in possession of a Refresh Token, it will pass the Refresh Token to request a renewed Access Token.
4. After the authentication-service has checked the credentials or the Refresh Token for validity, it will issue a new Access Token (and on initial login an additional Refresh Token) and pass them back to the mobile application.
5. The mobile application can now use the Access Token to request protected resources on other data services, that trust our authentication-service. For example, the user's protected profile from the social-service, which will verify the token and use the provided user-id claim to authorize the request.
6. The protected profile in step 5 may include statistics, which the social-service has to get from the upstream services, which will need the same Access Token to perform their own authorization.

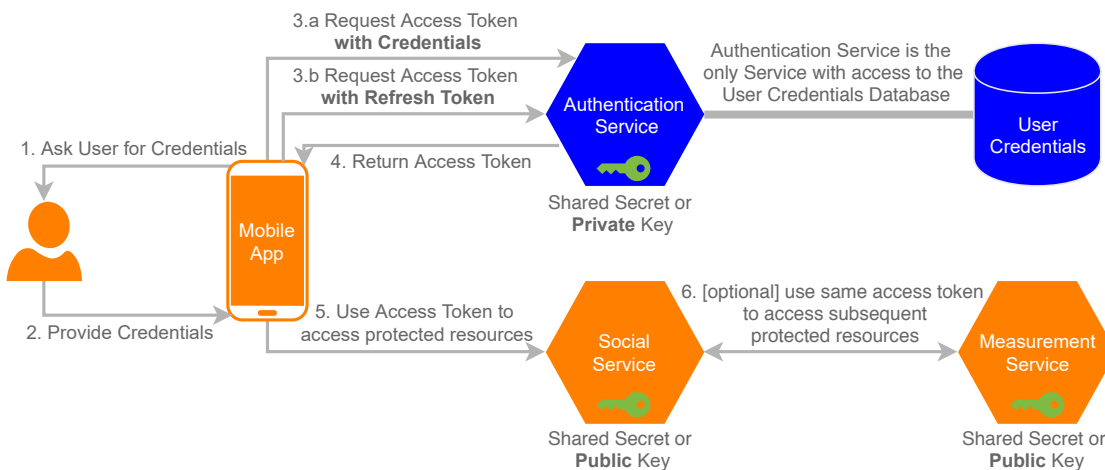


Figure 4.9: Example of a typical information flow using distributed authentication.

Usually, it is not possible to revoke or blacklist an already issued Access Token based on JWTs, because they are meant to be usable by multiple other resource servers on their own without double checking their status. Hence, they must expire after a relatively short time, for example after 10 minutes, in order to prevent damage, if an Access Token actually gets disclosed to an unwanted party. In that case it is at least only

valid for a short time. Since Refresh Tokens are only meant to be used on the issuing authentication-service, they can be easily stored and revoked later, if a user suspects his client application compromised. Their lifetime can therefore be longer, for example multiple days.

### 4.7 Bounded Context: Social

The Social Context can be composed of a simple microservice and a dedicated database. That service is covering any functionality related to social activity of users on the platform, which is briefly explained in this Section.

Users should be able to engage in discussions about phenomena in different geospatial areas. Since we already have introduced *DGGS* as a means to index and separate geospatial area, we want to follow this approach in the design for social activities. We propose a structural approach for discussions, that is similar to common Internet forums, with a tree of sub-forums and multiple topics within one specific sub-forum. The forum-tree is aligned to the hierarchical index of the *DGGS*, in our case the different cells produced by *H3*. This approach has the benefit, that a user can open a discussion within exactly one geographical region, specified by the hexagonal cell in a certain resolution. Child cells of that geographical region are realized as further sub-forums.

Since *DGGS* are not always covering exactly the area of interest and a larger cell is to over-sized. We propose the additional concept of *groups*. Groups can be created and joined by any registered user. The *group owner* should be able to name the group, provide a simple description and most importantly, specify multiple geographical *H3*-cells, that group is specifically focused on. To keep the concept simple, we propose a flat hierarchy with only one level of discussions within groups. However, tags with the *H3*-index can be used to filter discussions geographically within groups. These tags could also be used to show discussions actually opened within groups in the regular forum-structure under the corresponding sub-forum and mark it as a discussion opened in a group.

Additionally, users can maintain a simple profile with picture and description. They should be able to follow other users and discussions to get notified when new contributions happen. By creating a relationship between users, we can allow users to notify their followers of recent submitted measurements and achieved awards. Therefore, this bounded context might use a separate Kafka-Streams or Kafka Consumer instance to listen on measurement- and incentive topics, in order to notify the corresponding followers.

While the concept is simple and obvious, the user-experience for social activities is strongly relying on the specific frontend-side implementation and how well social features are integrated into the application. For example, there should be a map-based geographical search for certain forums showing the different cells on a map. Additionally, a user might be interested in related discussions, when clicking on a item in the measurement-visualization.

## 4.8 Bounded Context: Incentive

The Incentive Context is responsible for managing user engagement and to keep users motivated for contributing measurements. We currently see two types of incentives as important pillars of our platform, *Service Incentives* and *Entertainment Incentives*.

### 4.8.1 Service Incentives

As specified in Section 3.5, we use Service Incentives to limit platform features for users that are not contributing and in turn reward actively contributing users with certain exclusive features. Two major challenges for using such incentives within our system are related to actually calculating the different Member Levels of users in a continuous way and then eventually make that information available to the *access-service* in order to perform authorization with those Member Levels at the data access API. Since these levels are dependent on the number of contributed measurements within a given

## 4 Backend Design

timespan, certain business logic must check incoming measurements of all users to notice when a user has achieved the requirements for a new Member Level.

### 4.8.1.1 Calculation

The calculation could be performed using the same Stream-Processing principles, as described in Section 2.2.3. The proposed design of the *Measurement Context* is already performing stream processing and has certain measurement data and mappings of measurement ids and user ids already present in Kafka topics. The Incentive Context could also use these topics with own microservices leveraging the Kafka-Streams library to perform custom processing of measurement-user relationships and aggregate certain metrics within different time windows to continuously refresh the Member Level of users.

### 4.8.1.2 Publishing

Publishing of Member Level changes could as well be performed using a specific Kafka topic. Every time the implementation concludes, that a certain user has reached a new Member Level it will publish the change to Kafka and interested services can read those topics. A level expiry can be added to these updates, to inform other interested parties for how long this level is valid, because for example *Active Member* will be downgraded after a certain timespan with no contributions. Additionally, an API Endpoint for requesting the latest level for a user would also be beneficial to give services like the access-service, the possibility to request the level of a user on-demand and cache such level until expiry or state-updates via the Kafka topic.

Another approach would leverage capabilities of JWTs to transport claims. This would however require the frontend application to be aware of the current Member Level and Member Level changes, because it must maintain the *Incentive-JWT* as a second token aside from the *Access Token*. We could use the Incentive-JWT in a similar way as the approach used at authentication in a distributed system (see Section 4.6.2), where they transport the claim, that a user has successfully authenticated with the system. We could provide an API Endpoint to issue an Incentive-JWT containing the current Member Level,

which is valid until the predicted time the user will lose his Member Level. On request for data access the frontend application must provide both tokens to the access-service in order to access data with the correct Member Level. However, the first approach has the benefit, that is completely transparent for the frontend application, because level updates are handled solely in the backend.

### 4.8.2 Entertainment Incentives

The Entertainment Incentives we have addressed in Section 3.5.2 are based on the collection of statistics. We can use another stream processing service to listen on every single measurement provided by a user, transform them into statistics containing attributes like values, H3-Indexes and timestamps. Gamification however, is lot harder to realize, because each challenge requires a Kafka-Streams instance with a custom implementation of the business logic describing the challenge. Once a user has fulfilled a certain challenge he is rewarded with an Award and the information is persisted in a database to be requested via an API-Endpoint shown within a profile view in some frontend implementation.

## 4.9 Bounded Context: Communication

The role of this context is more kind of a helper-role than an actual business use case. However, having a central context for communicating with the user, has the benefit that only that context is needing the contact information. An additional benefit of a central Communication Context is easier legal auditing, because every communication could be logged in the database. It is responsible for sending native device notifications, emails or, if any use case has the need for, even SMS. A typical use case might be the Social Context recognizing, that there is a new post in a discussion a user follows. Depending on the settings of the user, the Social Context will order the Communication Context to trigger a notification, email or even both. The implementation of this context is then responsible for delivering such information to the user.

### 4.9.1 Email

The context must know at least the user id and the matching email address. Because of the decoupled design, it must get access to such information before it can send emails to the user. Since the user is probably already using an email to register with our platform in an authentication-service, we could integrate a step within the registration process, that is posting the contact information to a communication-service which stores it in its own database. Since a user could change his login-information and therefore his email address at the authentication-service, we would have to make sure that an update in the Communication Context is performed, whenever a user changes his contact information in order to prevent inconsistent information. Inconsistencies would result in a situation where the user would get emails to an obsolete address and might never be able to read them. Additionally, such an implementation would render the two contexts tightly coupled in which case merging the contexts altogether should be considered.

Alternatively, we could use a publish/subscribe messaging approach using Kafka and have the authentication-service make information changes of users available globally. Every interested service, such as a communication-service could follow these changes and store them in their database for fast access. Another reactive approach would be, to let the communication-service request the contact information of users at the authentication-service every time it has the need for something like an email address and probably cache it for a certain time.

The cleanest approach in terms of the data-model, would shift the responsibilities for contact information from the authentication-service to the communication-service and let that context be the owner of that data, since contact information is a means to communicate with users. This approach is still requiring the authentication-service to post such information once during the registration process and then forget the email address and use information such as a unique username for authentication.



### 4.9.2 Notification

Notifications are usually sent via an externally hosted push service operated by a provider. To push any notification, the communication-service must know a unique *device registration id*. That device id has to be mapped with a currently logged in user, to have the external push service contact the correct device for the user. The communication-service must provide an API-Endpoint to allow mobile devices setting and deleting such device registration ids whenever it changes, which is different depending on the operating system.

### 4.9.3 Proposed Internal Design

Figure 4.10 shows a proposed design of the context. It is composed of one service with a database for persistence of contact information and device registration ids. As mentioned before, upon registration the authentication-service must create the user object within the context, which it should anyway to initiate a verification of the email address by sending an email to the provided address. Any subsequent contact information related changes are performed directly at the API of the communication-service. A mobile device can push and delete certain device registration ids at corresponding endpoints to be able to start and stop receiving push notifications. The actual information that has to be communicated, must be triggered at two independent API-Endpoints for email and notification. However, these Endpoints should not be published externally or must be secured with a different authorization mechanism than user related JWTs. Otherwise emails and notifications could be sent by any user capable of using an HTTP-API, which should be prevented. Emails and notifications are typically not sent on behalf of a user, but rather asynchronously when certain events occur in other contexts. The communication-service can get the corresponding email address and device registration id for the user from the database and subsequently send the information via email to the user's email account or trigger the corresponding Push Provider to deliver a notification to the user's device. The service can also possibly be extended to a "Customer Relationship Management Service" with capabilities like newsletters or reminder-notifications. This

## 4 Backend Design

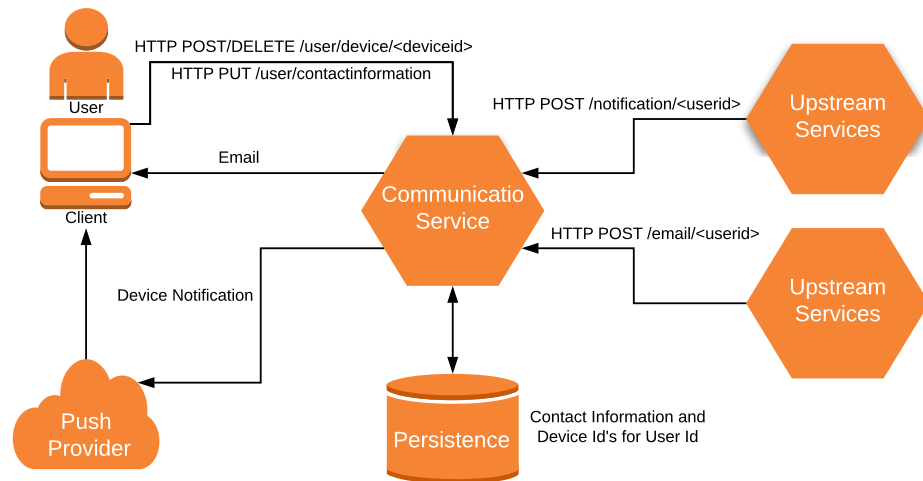


Figure 4.10: Overall architectural design of the Communication Context.

could be one way for a Crowdsensing platform to remind passive users and motivate them to become active again.

# 5

## Implementation

This Chapter explains some interesting implementation aspects of the platform specified in the previous Chapter in more detail. First, we describe some deployment files, we used to provision our development cluster and subsequently deploy our services via an automated deployment pipeline following the continuous delivery concept. Afterwards, we show how the measurement context representing the core functionality can be implemented using the stream processing capabilities of Kafka and a lightweight Java-Framework for HTTP-based data access.

### 5.1 Infrastructure

We are going to set up our infrastructure using the same principles and cloud-based primitives we would assume to be available in a production environment. The ability to have the software running in a production-like environment from the beginning is of great importance. This can highlight architectural- and configuration shortcomings. At the same time, it yields a first tendency of how the system will behave in production.

#### 5.1.1 Integration Cloud Environment

Because the project is maintained by a research institution, we have access to a cloud-provider specifically offering free resources for scientific purposes, which are sufficient for our development cluster. The BWCloud SCOPE project (SCience, OPerations and Education)<sup>1</sup> has developed a private cloud offering hosted and maintained by a

---

<sup>1</sup><https://www.bw-cloud.org/en/project>

## 5 Implementation

federation of Universities from Mannheim, Ulm, Karlsruhe and Freiburg. The IaaS offering is based on the open source software Openstack, which provides features comparable to commercial providers. Although, our BWCloud quota has some limitations in load-balancing, networking and availability of public IPv4 addresses. We could use the graphical UI to provision our virtual resources, this however, involves many manual steps and is hard to reproduce. Reproducibility is of great importance, especially for an integration cluster, which will likely break because its purpose is exactly to test configurations there rather than in a production environment.

### 5.1.2 Infrastructure as Code

Constantly tearing down and setting up a cluster from scratch can become cumbersome and often some steps are forgotten or performed in the wrong order. Additionally, manual rollbacks may not be possible after wrong configuration. To solve this problem, there are tools, that allow to describe the desired state of an infrastructure via structured text-files, which are used to provision and configure that infrastructure using APIs provided by cloud-providers. This has the benefit, that we can use a version control system, such as *GIT*, to have all different infrastructure states versioned and documented in text-files, which allow easier rollbacks when misconfiguration has happened. This concept is called *Infrastructure as Code (IAC)* and is one of the central parts in building an application based on cloud-native infrastructure [13].

We use a popular tool called *Terraform* to provision the infrastructure, because we have previous experience with the domain specific language needed to describe the infrastructure. Virtual Machines, Internal-Networks, Routers, Security- and Firewall-Rules as well as FloatingIPs, used to access the Virtual Machines via the Internet are specified via a text file similar to Listing A.2. The following commands in Listing 5.1 will set up environment variables with credentials of our account at the BWCloud and trigger Terraform to provision the infrastructure shown in Figure 5.1.

The result is a basic 3-node *Kubernetes*-Cluster managed by *Rancher2*.

Listing 5.1: Shell setup and Terraform commands to provision the infrastructure

```

1 $ source ulm_dbis_crowdsensing-openrc.sh
2 $ terraform init #only relevant on first run
3 $ terraform apply

```

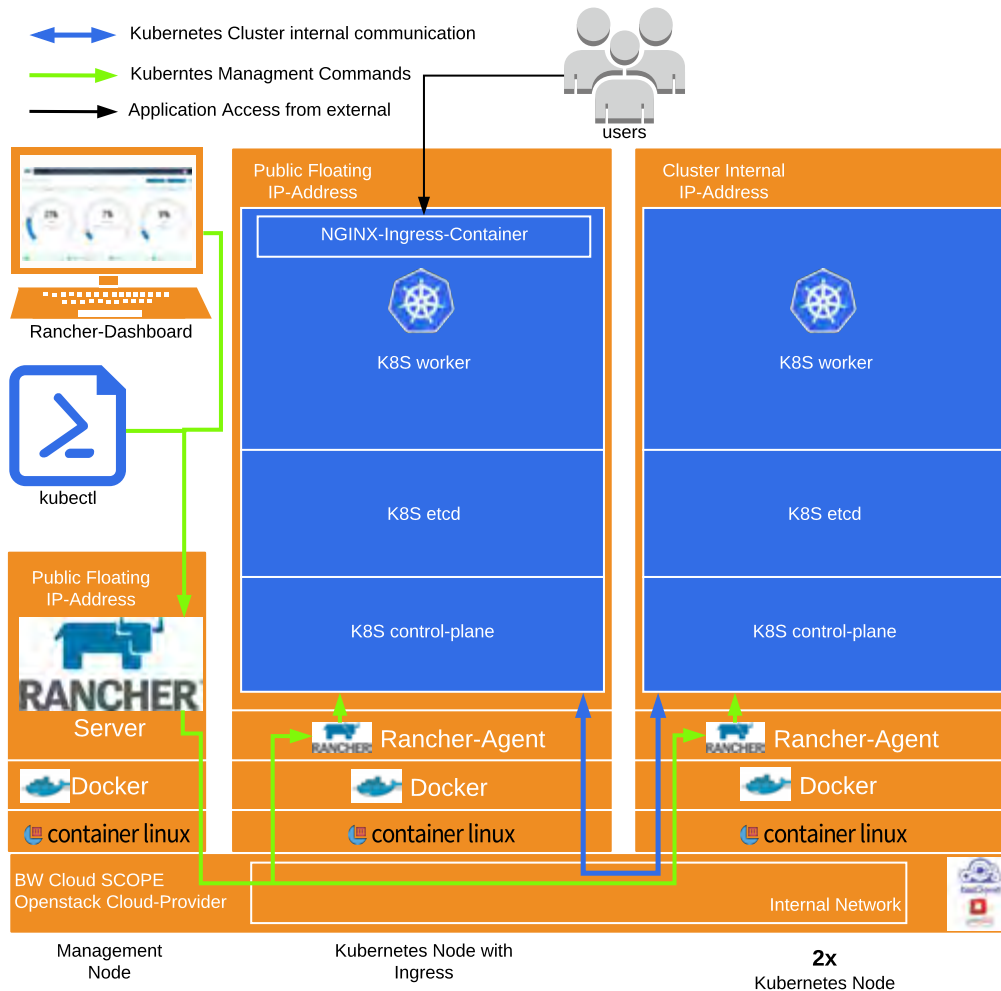


Figure 5.1: Diagram showing the integration infrastructure deployed by Terraform.

After a few minutes everything from networking to the computing-nodes is set up and the specified commands for provisioning the individual nodes have run automatically. We use *CoreOS Container Linux* as our operating system on the computing-nodes, because

## 5 Implementation

it is a lightweight Linux derivative specifically designed for cloud-native purposes. Other than for example Ubuntu or CentOS, it has support for Docker out of the box, which we require to run the Rancher and Kubernetes components.

### 5.1.3 Kubernetes Cluster

It requires a lot of effort to setup a functional and reasonable stable Kubernetes cluster from scratch. Therefore, we use *Rancher2* as a wrapper around Kubernetes to simplify the creation of the Kubernetes cluster. Rancher2 is one of the easiest ways to set up and manage a Kubernetes cluster to date. It minimizes the burden of knowing all the little detailed configuration parameters, that are required for a stable Kubernetes cluster. In fact, it can be a full-time job itself, to maintain the cluster once it has reached a certain size. Rancher2 proxies all actions to the standard Kubernetes API via its Agents as visualized with the green lines in Figure 5.1.

Since our quota is limited, we currently can only use two Floating IPv4 addresses to access our machines via the Internet. As you can see in Figure 5.1, we have configured the cluster with only one ingress-node, which is specified in the cluster configuration with a *node-selector* and a label *app: ingress* (see Listing A.1). Kubernetes will automatically make sure, that all pods having the same label are scheduled onto that particular node. This will of course also apply to its own ingress-container, that is exposing services to the outside via dynamically generated URLs usable to access our services. Additionally, we have quota for eight data-volumes provided by the Openstack Cloud-Storage provider *Cinder*, which is essentially a network attached storage solution and dynamically provisioned and mounted to cloud-instances as regular data-volumes. The *cloud\_provider* object specified in Listing A.1 gives K8S access to the Cinder-API in order to manage these data-volumes transparently according to the placement of pods that need to persist their state.

## 5.2 Deployment

After having set up all the infrastructure and a running Kubernetes cluster, we can start to think about deploying our applications into the cluster.

### 5.2.1 Service Deployment

The initial deployment of our application can be made either by using Rancher's graphical user interface, which is more intuitive for non-experienced developers, or via the *kubectl* command-line tool, which uses a YAML-based Kubernetes deployment file, as shown in Listing A.3.

The commands in Listing 5.2 show how to deploy multiple Kubernetes resources via *kubectl* and subsequently show all resources now deployed to the *default* namespace. We assume that there is a *kubeconfig.cfg*-file, needed for authorization at the Kubernetes API, and multiple *.yaml* Kubernetes deployment-files in the *deployment/*-folder in our directory. The tool will automatically find all matching files in that folder and deploy them one after another.

Listing 5.2: Kubectl commands for applying and viewing deployment specifications

```
1 $ kubectl --kubeconfig=./kubeconfig.cfg apply -f ./deployment/*
2 $ kubectl get all --namespace default
```

### 5.2.2 Continuous Delivery

We implement continuous delivery with a custom deployment pipeline specified in a *gitlab-ci.yml*-File. It describes a topology of steps performed after another. A detailed description of the functionalities can be found in the official Gitlab-Documentation<sup>2</sup>. Other Platforms, such as *Atlassian Bitbucket*<sup>3</sup> offer similar capabilities with a file called *bitbucket-pipelines.yml*. As explained in Section 4.3.3, we favor distinct repositories

<sup>2</sup><https://docs.gitlab.com/ee/ci/>

<sup>3</sup><https://confluence.atlassian.com/bitbucket>

## 5 Implementation

for each microservice, instead of a mono-repository. This is important, because each microservice requires an independent pipeline configuration due to different build and deployment specifications as services could be implemented in different programming languages. For example, for the services in the Measurement Context we use build and test steps conforming to *Java*. The authentication-service based on *GoLang* however, requires a different configuration.

### 5.2.2.1 Pipeline Configuration

The pipeline is based on different *Docker Containers* and scripts, that specify what actions to perform in the pipeline. There are two main primitives for specifying a pipeline:

- **Job:** Defines a certain step within the pipeline. It has certain attributes, such as *image*, *stage* and *script*, that define the environment as well as, when and what the job is doing.
- **Stage:** Is a global element describing an order of levels. Jobs are then assigned to stages, which is defining the topology of all jobs. Jobs in different stages are executed successively. Jobs in the same stage are executed concurrently.

For example, our pipelines defined in Listing A.4 for the *measurement-access-service* uses a build-, bake- and deploy-stage. We skipped a test-stage since we did not write any tests for our prototypic implementation. While simple Unit tests would be performed in the build-stage by a build-management-tool like Maven or Gradle, for production use every pipeline should perform acceptance- or at least smoke-tests using the just created container image, to ensure that there is no faulty image deployed to production.

**The build-stage** , shown in Listing 5.3, has only one job and is effectively triggering a Maven build. Because we are using the official Maven Docker image we have access to the tool *mvn* out of the box. The Gitlab-Runner will pull and start the *maven:3-jdk-8* container and fetch the git-repository into the running container. All actions defined in the *script* attribute are now executed in the root of the repository. In this example *mvn clean install* will discover a *pom.xml* and execute the corresponding build. The *artifacts*



Listing 5.3: Specification of the Gitlab CI pipeline for the build step

```

1 build:
2   image: maven:3-jdk-8
3   stage: build
4   script:
5     - mvn clean install
6 #declare artifacts which should be passed to the next stage of the pipeline
7   artifacts:
8     paths:
9     - ./target

```

attribute is specifying the files or folders, that are passed to the next stage in the pipeline, after all the scripts have been successfully executed. However, these files are not added to the repository as a committed change. In our case the build-artifact is an executable *.jar* file,

**The bake-stage** , shown in Listing 5.4, differentiates between the type and branch of the commit. Only the master-branch or tags, which are used for official releases of a new service version, are going to be baked into a new container-image. We must declare the build-job as a dependency to get access to the created artifact of that job. With the *only*-attribute we specify which job is executed on either a commit to master or a commit of a tag. The job is logging into the Docker-Registry using environment variables automatically injected by Gitlab into the build-context. Subsequently Docker builds a new image using the definitions specified in a *Dockerfile*<sup>4</sup> placed in the root directory of the repository. For referencing container-images to the exact version of the git commit, we use the unique commit-hash as an image-tag. When a new version of the service has to be released, we additionally reference the image with the tag-name, which usually has the form of “*v{majorversion}.{minorversion}.{fixnumber}*”

---

<sup>4</sup>A file that describes the actions docker must perform in each layer on a build. Every available Docker-Container can be used as a base container to stack the layers on top.

## 5 Implementation

Listing 5.4: Specification of the Gitlab CI pipeline for the bake step

```
1 bake-master:
2   image: docker:latest
3   services:
4     - docker:dind
5   stage: bake
6   dependencies:
7     - build
8   script:
9     - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
10       $CI_REGISTRY
11     - docker build --pull -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" .
12     - docker push "$CI_REGISTRY_IMAGE"
13   only:
14     - master
15 bake-tag:
16   image: docker:latest
17   services:
18     - docker:dind
19   stage: bake
20   dependencies:
21     - build
22   script:
23     - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
24       $CI_REGISTRY
25     - docker build --pull -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" .
26     - docker tag "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" "$CI_REGISTRY_IMAGE
27       :$CI_COMMIT_TAG"
28     - docker push "$CI_REGISTRY_IMAGE:$CI_COMMIT_TAG"
29   only:
30     - tags
```

The **deploy stage** , shown in Listing 5.7, requires us to configure some previous settings. Since Kubernetes performs authorization, we must provide a *kubeconfig*-file to the kubectl-tool.

### Security Note

**Problem:** It is considered bad practice to maintain credentials as files within in the repository, since it is introducing a critical security vulnerability. If we ever plan to give third-parties any access to the code in the repository for any reason, even just as “Guests”, they would be able to use exactly those credentials (e.g. the kubeconfig-file) to perform actions on behalf of us (e.g. change any configuration in our Kubernetes cluster). Deleting the credentials before has no effect, since even deleted files are accessible in a versioned environment.

**Solution:** Gitlab allows to configure secret variables in the repository settings, that are only accessible with extended rights on the repository. These variables are injected into the pipeline context as hidden environment variables and therefore can be used to secretly inject critical credentials into the pipeline and use them in the scripts as regular environment variables.

The *kubeconfig* however, is a structured text-file that cannot be used as an environment variable without further measures. We use base64-encoding, as shown in Listing 5.5, to encode the file into a regular string that we can use as a secret environment variable.

Listing 5.5: Bash command to encode a kubeconfig into an environment variable.

```

1 $ ENCODED_KUBECONFIG=$(base64 kubeconfig.cfg)
2 $ echo $ENCODED_KUBECONFIG
3 Output: #large string
4 YXBpVmVyc2lvcj...DogImJ3Y2xvdWQi

```

We have to reverse that step in the pipeline-job using the command in Listing 5.6 to get back the original kubeconfig-file in a structured form. Otherwise it will not be accepted by kubectl. Now that we have a secure way to inject our kubeconfig into the job, we can use

## 5 Implementation

Listing 5.6: Bash command to retrieve the original kubeconfig-file from an environment variable holding the encoded string.

```
1 $ echo $ENCODED_KUBECONFIG | base64 -d
2 Output: #large string
3 apiVersion: v1
4 kind: Config
5 clusters:
6 - name: "bwcloud"
7   cluster:
8   ...
```

any image that contains the kubectl-tool, like the image provided by Docker-Hub-User “roffe”<sup>5</sup>, to issue commands to our Kubernetes cluster. This step is only executed, when triggered manually in the Gitlab-UI, and only available at a service release in form of a GIT-tag. The job orders Kubernetes to change the *desired-state* with the new image,

Listing 5.7: Specification of the Gitlab CI pipeline for the deployment step

```
1 deploy:
2   stage: deploy
3   image: roffe/kubectl
4   only:
5     - tags
6   when: manual
7   script:
8     - echo $KUBE_CONFIG_INTEGRATION | base64 -d > kubeconfig
9     - export SERVICE=nynm-measurements-access
10    - kubectl --kubeconfig kubeconfig set image deployments $SERVICE $SERVICE
11    =${CI_REGISTRY_IMAGE}:${CI_COMMIT_TAG}
    - kubectl --kubeconfig kubeconfig rollout status deployments $SERVICE |
      grep successfully
```

which triggers a merge with the actual state and applies the new image in the deployment. We can query the rollout-state in the same manner and wait until the change has been successfully deployed.

---

<sup>5</sup><https://hub.docker.com/r/roffe/>

## 5.3 Measurement-Context

After introducing our conceptual and architectural approach for the Measurement Context in Section 4.5 and explaining our integration infrastructure and deployment methods in the former Section, we are now providing a prototypic implementation for the Measurement Context. The functional guideline is the Noisemap project introduced in Section 3. As discussed in the design chapter, the functionality is internally divided into multiple services, which we have implemented in Java because certain Libraries such as H3 and Kafka-Streams are best supported in that language. For the API-facing services we use a Java micro-framework called *Javalin* as it allows a fast and simple implementation of HTTP endpoints with filtering capabilities for authorization.

### 5.3.1 Model

At first we want to use this Section to outline the the data model we identified and which is shown in Figure 5.2. Measurements and aggregates are modeled as Simple-Features using *Type*-, *Geometry*- and *Property*-attributes for better compatibility with certain geo-libraries and data-storage in MongoDB, which supports indexing on GeoJSON structures. Each measurement gets a unique *id*, certain timestamps and is additionally geo-indexed using the *H3-Library*. The *Properties* attribute contains data relevant for the subject-matter of collecting noise measurements. Each measurement can be composed of multiple measurement types that are representing different weightings (*LAeq*, *LCpeak*, *TWA*). One such type can be either related to a single measurement, in which case only type and value are of relevance or it can be related to an *AverageFeature*, where min-, max-, mean- and count-values are of interest. To cover the requirement of using different sensor types and triggers, each measurement is tagged with the type of the sensor (smartphone, statically mounted IoT-sensor and possibly future devices that allow in-the-canal measurements<sup>6</sup>) and the type that triggered the measurement (manual, schedule, event). These tags allow users to filter different types of measurements for a visualization. Additionally, future analytical processing can take different types into

---

<sup>5</sup><https://javalin.io/>

<sup>6</sup>A type of measurement, where the SPL is measured directly in the canal of the human ear.

## 5 Implementation

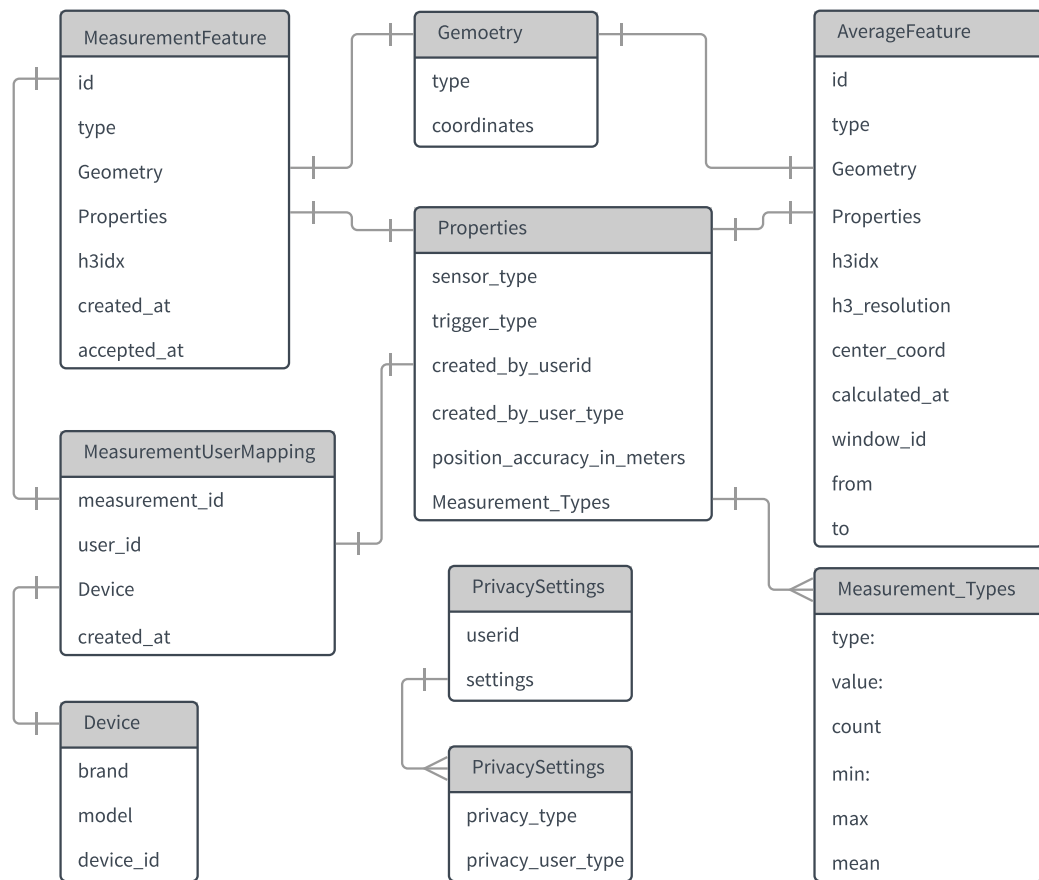


Figure 5.2: Diagram showing the data-model of the Measurement Bounded Context.

account, which allow to augment the platform to be usable with all kinds of measurement-sources in the future. If required, we also could weight various types of sensors and triggers differently based on their accuracy, to calculate more realistic averages.

To model the averages, we use *AverageFeatures*, which represent our aggregates. The geographical area an aggregate is covering, is a hexagonal polygon described by the *Geometry*-attribute. The different timestamps are describing the time window, for which the average is calculated, and when the calculation has been performed. We additionally maintain the center-coordinate of the polygon, the H3 index and the resolution of that hexagon, which is already encoded in the h3index but would not be extractable without

the library. However, we do not want to force frontend applications to use the library too, just to extract the resolution of the cell.

As described in Section 4.5.6, we want to maintain the privacy of users and pursue an approach, where we do not store any private data of users attached to a measurement object. After pre-processing, the *created\_by\_userid* attribute in the Properties attribute is erased and only the user-type (guest or member) is directly stored in the MongoDB collection for measurements. This separation allows to publish any measurement without privacy concerns. Instead, we store a mapping of user id and measurement id in a separate collection. Additional information about the device used for the corresponding measurement is also maintained in that mapping, which can be of interest for applying other correcting factors for specific devices, if a laboratory can identify a generic pattern for calibration in the future.

### 5.3.2 Implementation Overview

We have refined the conceptual architecture shown in Figure 4.5 and implement all three phases. A typical data-flow of one measurement through the implemented phases is shown in Figure 5.3.

### 5.3.3 Data Ingress Implementation

The first capability of the context is collecting measurements. This section focuses on Steps 1-4 of Figure 5.3. Every time a smartphone has finished a measurement it is posting a data object, shaped like the example in Listing 5.8, to an HTTP endpoint of the ingress-service. The service is reconstructing the Measurement object using the provided JSON data. The endpoint handler is attaching timestamps and the user id as an *created\_by\_userid*-attribute, if any JWT has been provided for authorization. Subsequently the configured Kafka Producer is publishing the measurement to the Kafka topic for *noise-raw-measurements* using the *created\_at*-timestamp as a key for equal distribution on multiple partitions. If successful, it returns an *HTTP-202 Accepted*

## 5 Implementation

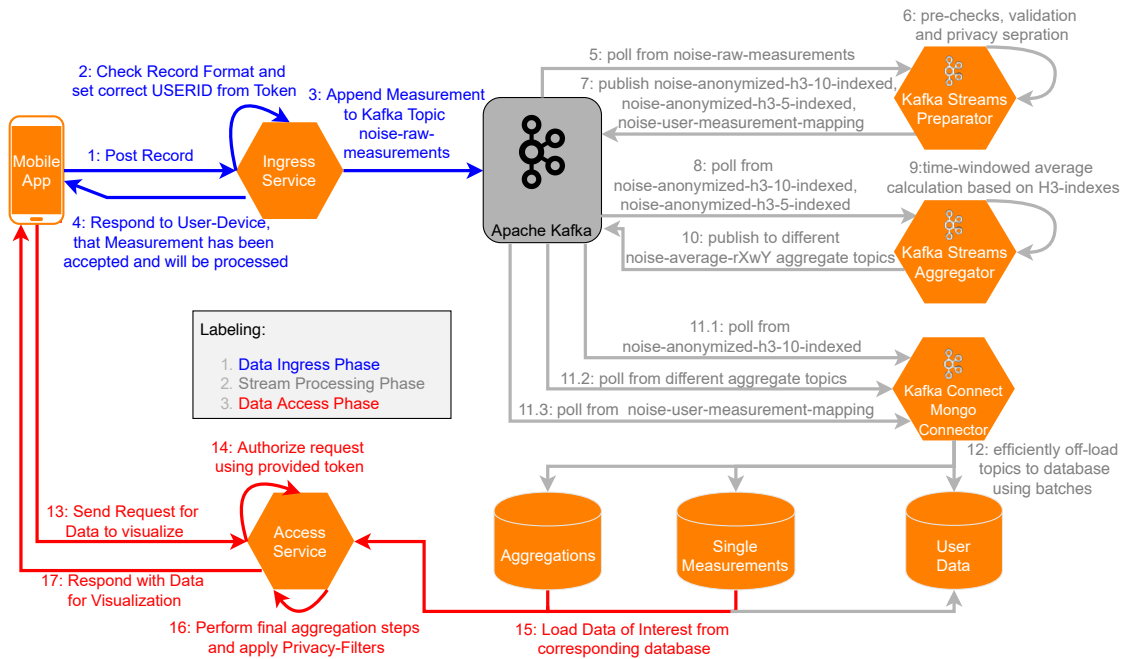


Figure 5.3: Concrete internal implementation of the Measurement Context shown all major steps of the data-flow.

response to the smartphone, signaling that the data has been accepted but is still in processing and not yet accessible.

**IoT-Sensors** related ingress was skipped for our prototypic implementation because of limited time resources. Outlined however, this would involve an MQTT-Broker to collect measurements from the sensors and publish all messages to a separate Kafka topic, which would be called something like *noise-iot-measurements*. Either the pre-processor or a distinct third Kafka-Streams service must then streamline the byte-data in that topic into the common format of the *noise-raw-measurement* topic. Additionally, there should be a service exposing an API for managing and linking deployed sensors to user accounts and geospatial locations.



### 5.3.4 Stream Processing Implementation

This section focuses on Steps 5-10 of Figure 5.3. The stream implementation is composed of two different services implemented as regular Java applications that leverage the Kafka-Streams library. Table 5.1 shows a list of all topics currently used in the stream processing topology. To eventually persist the calculated results into MongoDB we use Kafka-Connect with a configured MongoDB-Connector plugin. To prevent endless growing topics, Kafka topics can be configured with data-retention policies to remove

Listing 5.8: Example structure for posting a measurement.

```

1 {
2   "type": "Feature",
3   "geometry": {
4     "type": "Point",
5     "coordinates": [
6       9.967101535538086,
7       48.384883089298114
8     ]
9   },
10  "properties": {
11    "position_accuracy_in_meters": 10,
12    "sensor_type": "-1", //type of the sensor (-1 corresponds to
13      testdataservice)
14    "trigger_type": "-1", //type of the trigger like notificatoin, schedule
15      or manual (-1 corresponds to test)
16    "measurement_types": {
17      "LAeq":{
18        "type": "LAeq",
19        "value": 70.5179
20      },
21      "LCpeak":{
22        "type": "LCpeak",
23        "value": 102.255226
24      },
25      "TWA":{
26        "type": "TWA",
27        "value": 61.82866
28      }
29    },
30    "device":{
31      "brand": "apple",
32      "model": "iphone4,1",
33      "device_id": "uuid" //optional
34    }
35  }
36 }

```

## 5 Implementation

Table 5.1: Current topics used in the noise stream-processing

Nr.	Topic	Key	Description
1	noise-raw-measurements	created_at	Is the entrance topic for every measurement.
2	noise-user-measurement-mapping	userid	Contains mapping objects that relate measurement-id and user-id
3	noise-anonymized-h3-10-indexed	H3idx (10)	Contains measurements that are filtered and anonymized. The key is an H3 index of resolution 10, in order to be correctly assigned to partitions used in average aggregation.
4	noise-anonymized-h3-5-indexed	H3idx (5)	Contains measurements that are filtered and anonymized. The key is an H3 index of resolution 5, in order to be correctly assigned to partitions used in average aggregation.
5	noise-average-r10w15	H3idx(10)	Contains small aggregations in H3-Resolution 10 and a window-length of 15 minutes.
6	noise-average-r10w60	H3idx(10)	Contains small aggregations in H3-Resolution 10 and a window-length of 60 minutes.
7	noise-average-r5w60	H3idx(5)	Contains larger aggregations in H3-Resolution 5 and a window-length of 60 minutes.
8	noise-average-r10w1440	H3idx(10)	Contains small aggregations in H3-Resolution 10 and a window-length of 1 day.
9	noise-average-r5w1440	H3idx(5)	Contains larger aggregations in H3-Resolution 5 and a window-length of 1 day.

data after a certain time or when a certain size threshold has been reached. We use a default retention time of one day, which means, that Kafka marks messages for removal after 24 hours. It is up to Kafka however, when exactly it removes the messages from disk.

### 5.3.4.1 Pre-Processor

The *pre-processor* is processing all messages on the *noise-raw-measurements* topic. First it is filtering measurement types that are not of the type *LAeq*, *LCpeak* or *TWA* and is then checking each of the three remaining values if they exceed certain thresholds that would not make any sense. A second step is then separating the *created\_by\_userid*

attribute together with the measurement id and the device information into a separate *noise-user-measurement-mapping* topic for privacy reasons. Next, it is trimming the coordinates to 5 decimal digits which is about 1m precision (compare Table 2.1) for efficiency reasons. We are using those coordinates to identify the H3-index in resolution 10, which is producing hexagons with an edge length of about 66m and the smallest aggregation size we are going to support. Additionally, we create a unique id for the measurement, which can be used by the database later. Apart from the index for small areas in resolution 10 we publish the same anonymized measurement additionally with an H3-index in an intermediate resolution of 5. This second index is used by the average processors to calculate intermediate results for larger areas, which is beneficial in the access-service for custom aggregations on requests in resolution < 5.

The resulting *noise-anonymized-xyz* topics do not have any user id attribute attached, but rather keep the information whether a guest or a user has produced that particular measurement for possible filtering requirements in a frontend implementation.

#### 5.3.4.2 Average-Processor

The *average-processor* is taking the *noise-anonymized-xyz* topics as input, because they are already filtered for fraud. It is producing multiple output topics in the format *noise-average-r<resolution-size>-w<window-minutes-length>*, which contain the aggregations for the different resolution and window combinations. Since Kafka-Streams is a high-level streaming library, certain aggregation operations are already usable via simple methods.

An example for aggregating measurements within a 15-minute time window based on the H3 index key is provided in Listing 5.9. As a first step we create the KStream object representing the *noise-anonymized-h3-10-indexed* stream and group it by the H3 index key in Line 4. Then we specify the window-length and set a retention-time in Line 8, which specifies for how long the window is refreshed, if measurements for that window arrive late. Line 7 specifies the generator for the initial aggregate and the actual aggregation method that should be used for each measurement. In the actual aggregation method, we calculate the average for each measurement-type using the *addValue(Type valueToAdd)* method of the Type-Class. The *mean-Value* is calculated

## 5 Implementation

using a sophisticated formula due to the logarithmic nature of decibels. In Line 8-10 we add certain window- and timestamp information to the message and publish it to the specified topic.

### 5.3.4.3 Kafka-Connect

This section focuses on Steps 11 and 12 of Figure 5.3. All the results produced by the Stream Processors is still residing in partitioned topics within the Kafka Cluster. To allow the access-service to efficiently query the data, we must persist the data in a database that is able to index it appropriately. Kafka-Connect is the native runtime to transfer existing data in- and out of Kafka using *Source- or Sink-Connector Plugins* running as “*Tasks*” on *Connect Workers*. Since Kafka-Connect is already integrated tightly within the Kafka-Cluster, all we need to provide is one or more Instances acting as the Connect Workers and equip these instances with the corresponding Connector Plugins in form of Java *.jar* packages. After the Connect Worker container is running and connected to the Kafka-Cluster, we configure the different Connect Tasks by posting a JSON-configuration, like the one shown in Listing 5.10, to an internal Connect-Endpoint in order to tell the Connect Workers what to read from Kafka, how to transform and where to persist the data. The corresponding Collections are defined by the *connect.mongo.kcql*-attribute in a configuration scheme. We persist topics 2-9 of Table 5.1 with an “upserting”-strategy. The *UPSERT* command is a made-up word, specified as “update if present else insert” and is persisting the data coming from the specified topic into the corresponding Collection based on the defined primary key.

### 5.3.4.4 Database

We use MongoDB as the persistence layer in our Measurement Context because its geospatial support is based on the GeoJSON-Format, which lets us integrate the solution seamlessly with our services. The support has been introduced to allow efficient queries by using certain indexing mechanisms, which are a key enabler for efficient data access of geospatial data. Otherwise a database would need to check the coordinates of all

Listing 5.9: Code-snippet for aggregating noise measurements based on time-windows with Kafka-Streams (certain Java Code is hidden for better readability).

```

1 //create the stream object and group by the H3 index key
2 KStream<String , MeasurementFeature> anonymized = builder.stream(" noise-
   anonymized-h3-10-indexed", Consumed.with(stringSerde , measurementSerde));
3 int retentionHours = 1; int lengthMinutes = 15; int resolution = 10;
4 anonymized.groupByKey(Serialized.with(stringSerde , measurementSerde))
5     .windowedBy(TimeWindows.of(TimeUnit.MINUTES.toMillis(lengthMinutes))
6     .until(TimeUnit.HOURS.toMillis(retentionHours)))
7     .aggregate(AverageFeature::new, performAggregation(h3),/* third
   parameter hidden for simplicity*/)
8     .mapValues(setWindowsAndTimestamps())
9     .toStream((key, value) -> value.getH3idx())
10    .to("noise-average-r" + resolution + "-w" + durationMinutes , Produced
   .with(stringSerde , averageSerde));
11
12 /**Method that performs the aggregation using a single measurement that is
   added ontop of an already present aggregate based on the h3idx*/
13 private Aggregator performAggregation(H3Core h3) {
14     return (h3idx, measurementFeature, averageFeature) -> {
15         /*initialization hidden for simplicity*/
16         //For each measurement type recalculate the average seperately with
   its own count
17         measurementFeature.getProperties().getMeasurementTypes()
18         .forEach((type, measurementType) -> {
19             Type averageValue = averageFeature.getProperties().
   getMeasurementTypes().getOrDefault(type, new Type(type,0));
20             averageValue.addValue(measurementType);
21             averageFeature.getProperties().getMeasurementTypes().put(type, v)
   ;
22         });
23         return averageFeature;
24     };
25 }
26
27 /*Method of the Class Type for adding a new Value to the average*/
28 public void addValue(Type value) {
29     if (count == 0) {
30         min = value.min;
31         max = value.max;
32     }
33     min = (value.min < min) ? value.min : min;
34     max = (value.max > max) ? value.max : max;
35     mean = (mean != null) ? mean : this.value;
36     value.mean = (value.mean != null) ? value.mean : value.value;
37     int totalcount = count + value.count;
38     //calculate mean dB over all previous values plus the new value
39     mean = 10 * Math.log10((count * Math.pow(10, 0.1 * mean) + value.
   count * Math.pow(10, 0.1 * value.mean)) / totalcount);
40     count = totalcount;
41 }

```

## 5 Implementation

Listing 5.10: Example configuration for the MongoDB connector to persists the 15-minute average results.

```
1  "connector.class": "com.datamountaineer.streamreactor.connect.mongodb.sink.
   MongoSinkConnector",
2  "tasks.max": "1",
3  "topics": "noise-average-r10w15",
4  "connect.mongo.database": "measurements",
5  "connect.mongo.db": "measurements",
6  "connect.mongo.batch.size": "10",
7  "connect.mongo.kcql": "UPSERT INTO averagel5-10 SELECT * FROM noise-average
   -r10w15 PK window_id",
8  "connect.mongo.connection": "mongodb://mongodb:27017",
9  "transforms": "tstransform",
10 "transforms.tstransform.type": "org.apache.kafka.connect.transforms.
   TimestampConverter$Value",
11 "transforms.tstransform.field": "to_date",
12 "transforms.tstransform.format": "yyyy-MM-dd'T'HH:mm:ss.SSSXXX",
13 "transforms.tstransform.target.type": "Timestamp",
14 "transforms.tstransform.type": "org.apache.kafka.connect.transforms.
   TimestampConverter$Value",
15 "key.converter.schemas.enable": "false",
16 "value.converter.schemas.enable": "false",
17 "value.converter": "org.apache.kafka.connect.json.JsonConverter",
18 "key.converter": "org.apache.kafka.connect.storage.StringConverter"
```

records and match them with a requested area, which is a costly computation. To query geospatial results, there are certain helper functions that operate on a *2dsphere* index.:

- *\$geoIntersects*: Selects geometries that intersect with a GeoJSON geometry.
- *\$geoWithin*: Selects geometries within a bounding GeoJSON geometry.
- *\$near*: Returns geospatial objects in proximity to a point. Requires a geospatial index.
- *\$nearSphere*: Returns geospatial objects in proximity to a point **on a sphere**. Requires a geospatial index.

Listing 5.11 shows a simple example using MongoDB's geospatial feature set. The example is presented in JavaScript, since the MongoDB proprietary query language is based on it. Database drivers in other languages can have minor differences to build those commands and queries. We are creating an index on a *measurements* collection, create an item in the form of a GeoJSON-Feature within that collection and use the geospatial query feature to define a bounding polygon for the former

created measurement. The query will return the created measurement and all other measurements contained within the defined query-polygon.

Listing 5.11: Example of a MongoDB geospatial query

```

1 //Create a geospatial 2dsphere index in mongodb
2 db.measurements.createIndex({ geometry: "2dsphere" })
3 //Create a GeoJSON Feature representing a Measurement in the Database
4 db.measurements.insert({
5   "type":"Feature",
6   "geometry":{
7     "type":"Point",
8     "coordinates":[9.96438,48.38262]
9   },
10  "properties":{ /* certain measurement related properties*/ },
11  "_id":"5bc8a46055149a0001f01a88",
12 })
13 //Query with the $geoWithin operator using a rectangle.
14 db.measurements.find({
15   geometry: {
16     $geoWithin: {
17       $geometry: { //defines the box used to query the database entries
18         "type":"Polygon",
19         "coordinates": [[
20           [10.002403,48.396015],
21           [10.002403,48.364034],
22           [9.933223,48.364034],
23           [9.933223,48.396015],
24           [10.0024033,48.396015]
25         ]]
26       }}}}
27 })

```

We define the following Collections and associated indexes in the database:

**aggregations{15,60,1440}-{5,10}** These Collections have a *2dsphere* index on the *geometry* and *center\_coord* attribute for efficient access. Further, they have a regular index on the *h3idx* and the *\_id* attribute to efficiently retrieve single items from that Collections.

**measurements** Uses a *2dsphere* index on the *geometry* attribute and regular indexes on *\_id* and *created\_at* attributes to allow time-based filtering of measurements.

**measurementusermappings** Uses regular indexes on *\_id* and *created\_at* attributes to allow time-based filtering of measurement mappings with users.

## 5 Implementation

The *average15-10* is supposed to only hold the latest completed window to prevent an endless growing dataset, therefore we have put a time-to-live index on the window-end timestamp. This will tell the regularly scheduled MongoDB daemon to clean obsolete data-items from the collection. Since the window-length is 15 minutes, this *tll* index is set to 900 seconds on the *to\_date*, which will keep that document for a minimum of 15 minutes or a little bit longer in the database once the window is finished.

### 5.3.4.5 Data Access Implementation

This last Section focuses on Steps 13-17 of Figure 5.3 and is dealing with data access for the visualization via an API. The service exposes an HTTP-based REST API with multiple endpoints to access the measurement data in raw and aggregated form with various optional filters applied. We have public routes and routes behind the top-level-route *"/internal"*, that require authorization because as specified in Section 3.5.1, certain Member-Levels have limited access to the available data. Since *Guests* have access to pre-aggregated results that are one day old, there are the public routes */aggregations/60* and */aggregations/1440*, that represent hourly and daily aggregated data. At the hourly route, a *"from"* and *"to"* query-parameter must be present and specify a timestamp before the beginning of the current day.

Authorization is performed at internal routes, where the presence of a valid JWT is checked and the corresponding user-information, such as roles, is added to the request context. With that information, each of the following endpoints can determine whether a user has the correct status to be able to access the corresponding data:

***/internal/aggregations/{60,1440}*** Returns multiple already aggregated results for certain H3indexes in a specific boundary and timestamp. The result will contain a Map of H3 indexes, where each key is containing a list of results in different time windows.

***/internal/aggregations/{15,60,1440}/latest*** Returns results of the latest completed pre-aggregated window, depending on the given window-size specified by the URL-parameter, a specified boundary and target H3 resolution.



**/internal/measurements** Returns single measurements within a specified boundary, time period and quantity limitation.

**/internal/measurements/liveaggregation** Returns aggregations for specified custom time periods within a specified boundary and a given target H3 resolution. The underlying handler is retrieving all matching measurements from the database and subsequently performing an iterative live-aggregation of the desired area and time period.

The access-service is using the official MongoDB database driver in version 3.7.1 to construct queries and get the corresponding results from the database. It is making use of the geospatial indexes and query capabilities. Most of the queries are of the relatively simple type “*geoWithin(bounding-polygon)*” and are similar to the Javascript version in Listing 5.11. Certain handlers, such as for live-aggregation, make use of an iterative database query, where the results are aggregated one by one, to avoid memory overflows because of a data set too large for the available memory.

**Privacy preserving implementation** is based on a simple Key-Value setting, which users can store in an *privacysettings* Collection in the database, where the Key is a *PrivacyType* of:

- **accurate\_coordinates**: Link user id with single measurements.
- **average**: Publish an average value of measurements performed by that user.
- **min\_max**: Publish minimum and maximum value of measurements performed by that user.
- **fuzzy\_coordinates**: Link user id with measurements where coordinates are blurred.

The value is a *PrivacyUserType*, that specifies which users can access the corresponding information of the above *PrivacyTypes*:

- **none**: No one is allowed to see the information.
- **following**: Only users, that the data-owner is following in a social-service implementation, are allowed to see the information.

## 5 Implementation

- **group**: Only users, that have one group in common with the data-owner in a social-service implementation, are allowed to see the information.
- **user**: All registered users are allowed to see the information.
- **guest**: Everyone, even guests, are allowed to see the information.

As Section 4.5.6 described, if not specified explicitly, all measurement information is returned without any user-related information. For requesting single measurements or a set of measurements, a requesting user can specify a query-parameter *populateUser*, which will take the measurement ids and find all corresponding users from the *measurementusermappings* Collection. For each of that users it is getting their corresponding *privacysettings* from the database, to check if the requesting user is allowed to see that information.

# 6

## Discussion

This Chapter is going to reason the feasibility of the approach and is subsequently explaining the overall scalability related aspects of the concept proposed in Chapter 4. Further, we compare how technical and conceptual design decisions would cover the requirements of the Noisemap project introduced in Chapter 3, which are exemplary for a platform dealing with crowdsensing of geospatial data.

### 6.1 Feasibility of the Approach

We showed in Chapter 5 how a concrete implementation of the proposed design concept can look like. We have implemented an end-to-end delivery pipeline to deploy code changes into a cloud-native environment using the hosted service of *Gitlab.com*. Committed code changes are built and baked into new images. Each new version release is automatically deployed into a Kubernetes cluster, which is specifically set up as our integration environment to have a non-production system running for testing new features and developing frontend applications without the risk of harming a production environment.

To show that the approach is working, we implemented a *Testdata-Service*, that is continuously posting random measurements located around Ulm, New-York-City and Worldwide at certain intervals to the ingress-service. Additionally, we have implemented a simple web frontend, capable of showing measurements and aggregations on a map. The frontend is implemented as a Javascript Single-Page Application, which is consuming the API of the access-service using asynchronous Javascript requests. Once

## 6 Discussion

the API results are received, they are shown on the map as depicted in Figures 6.1 and 6.2, using a visualization that colorizes certain noise levels using exemplary thresholds.

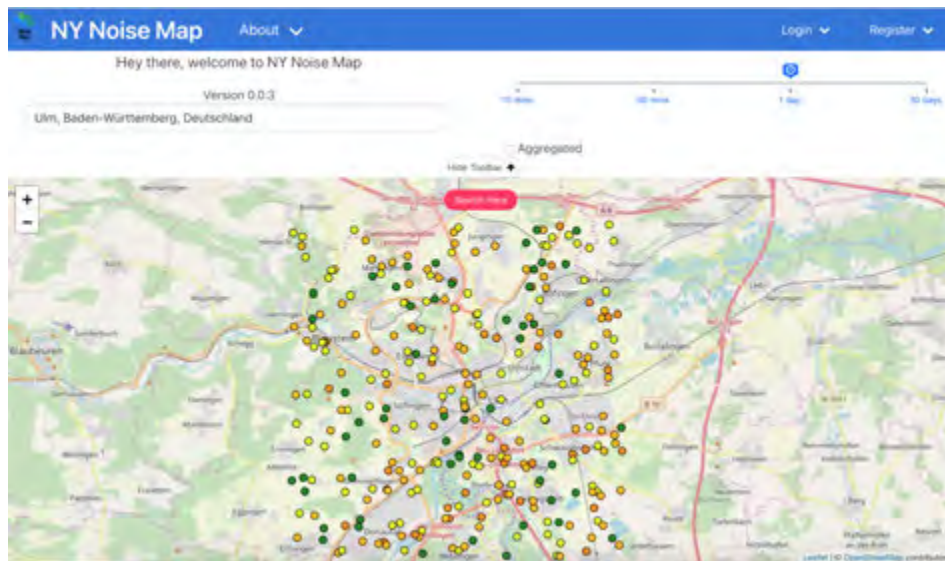


Figure 6.1: Prototypic web frontend showing a visualization of single test measurements.

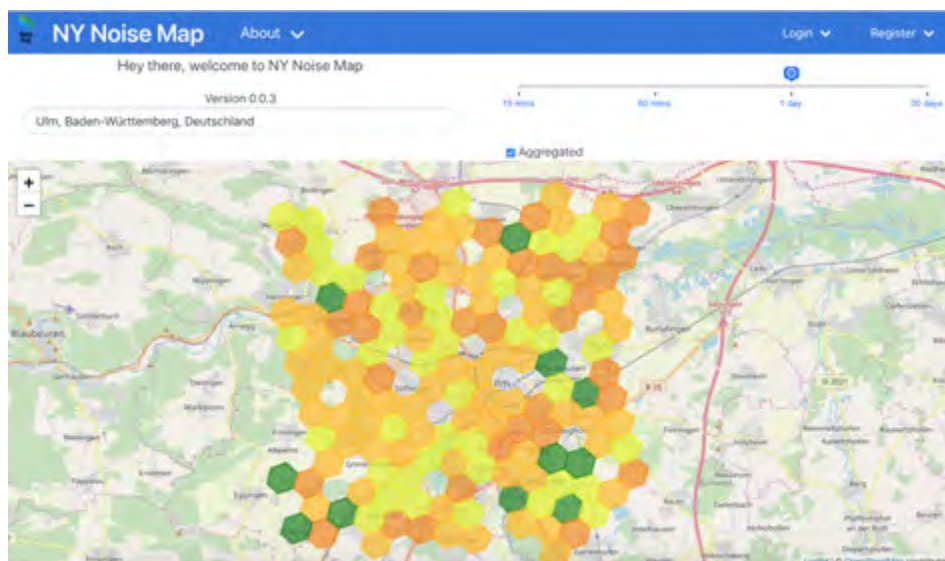


Figure 6.2: Prototypic web frontend showing a visualization of aggregations in H3 resolution 8 calculated by the stream processing pipeline.

## 6.2 Reasoning the Scalability

We are going to reason platform scalability by conceptually explaining how each used technology scales on its own and therefore contributes to the overall increased performance. The reasoning follows along the line, that each of the Microservices is independently horizontally scalable, but depends on a scalable persistence and communication layer. Since we use a scalable database technology and the Kafka ecosystem is scalable as well, the fundamental scalability is only constrained by the available computing resources. The resources are provided transparently and infrastructure agnostic by Kubernetes, which is built as a scalable cluster technology from the ground up and therefore allows to scale the available resources by adding usual computing resources in virtualized or non-virtualized form.

The following sections are elaborating on the scalability of each component.

### 6.2.1 Microservices

The cloud native concept of microservices allows to scale the services independently of each other. At times where there are a lot of measurements submitted, but only a few visualizations are requested, we can increase the number of ingress-services and stay with a small number of access-services. Vice versa, when there are a lot of requests for data, but less contributions, we can increase the number of access-services. Since all our services, which are exposing a publicly accessible API, are stateless, we can endlessly scale these services horizontally to provide enough resources to handle the number of API requests coming in from users or devices. The degree of scalability is however limited to scalability-aspects of upstream components which are dealing with state-handling, such as a database or Kafka.

### 6.2.2 Database

The scalability of the database is dependent on the technological solution that is used. While all solutions naturally scale vertically by increasing the power of a single instance,

horizontal scalability by increasing the number of instances is often not easily achievable. We focus on the horizontal scalability aspect in this section.

We use MongoDB in our prototypic implementation. MongoDB can be operated as a horizontally scalable cluster of multiple nodes, which is important to fit the performance needs of massively scaled downstream services, such as the access-service or upstream services like Kafka-Connect. MongoDB has two concepts that can be used to provide horizontal scalability. The first concept is *ReplicaSets*, which is similar to a Master-Slave replication. That concept is however merely providing scalability in terms of read-performance and is mostly used for resilience and fault-tolerance related reasons. Each ReplicaSet has a primary instance for read and write access to the data and one or more secondary instances for data replication, which can also be used for read access.

### 6.2.2.1 Sharding

More important than the concept of ReplicaSets, is the concept of *sharding*. It focuses on the scalability of read- **and** write-performance, because it “is a method for distributing data across multiple machines. MongoDB uses sharding to support deployments with very large data sets and high throughput operations [53].” Figure 6.3 shows the conceptual cluster architecture necessary to enable sharding in MongoDB. Instead of replicating the data, as it is the case with ReplicaSets, the data of one Collection is partitioned and distributed across multiple shards within the cluster. When every shard is only responsible for a part of the data in a Collection, in theory it is only exposed to a fraction of the overall load. The practical distribution of load however, is dependent on the right strategy and attribute-key for distributing chunks of data across the cluster. In an ideal case the data is distributed in a way, that the each shard is getting more or less equal requests, but at the same time the accessed data in one request should be on as less Shards as possible, because the *Query-Routers* will redirect the data-request to all shards, that are possibly containing data matching the request. These routers know where certain parts of a Collection are stored based on metadata stored in the *Config-Servers*.

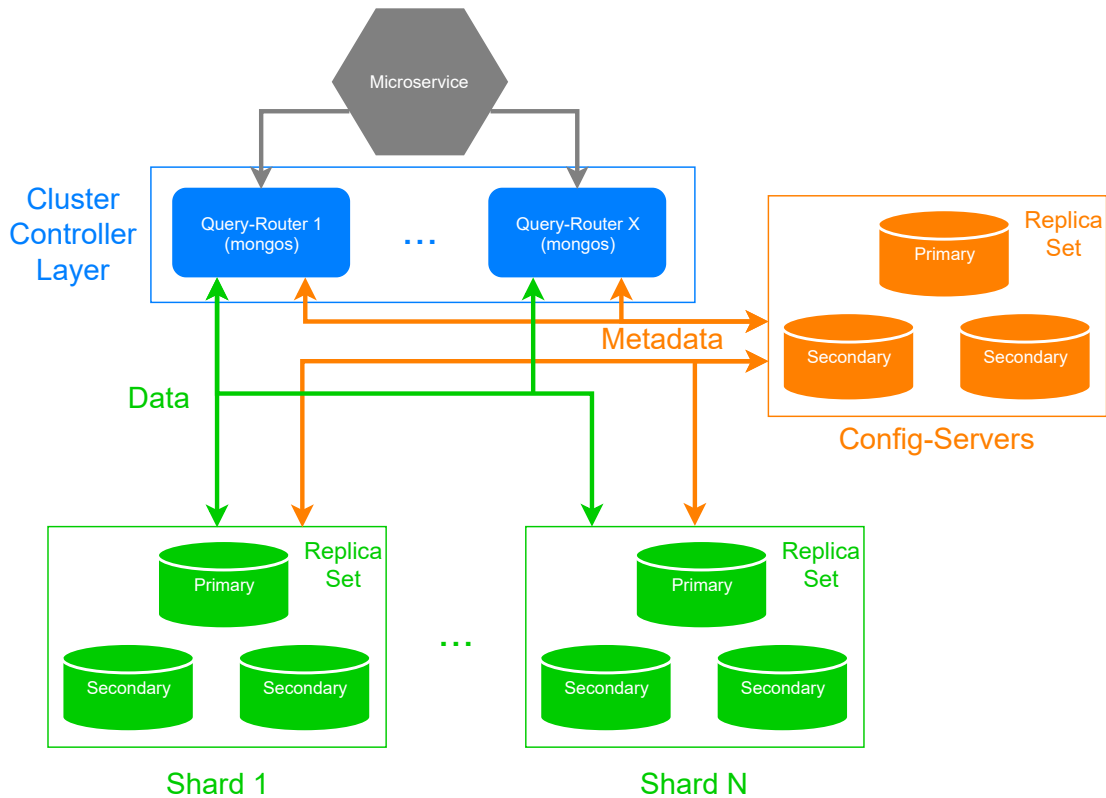


Figure 6.3: Scalable MongoDB cluster architecture for accessing data shards. Based on [53].

### 6.2.3 Kafka

The scalability aspect in Kafka must be viewed differentiated. One aspect is related to the scalability of the cluster by adding more nodes and therefore provide horizontal scalability and fault-tolerance. The other aspect is the scalability of the throughput of one single topic in Kafka. A Kafka cluster can easily be scaled horizontally by simply adding more nodes. All coordination and cluster specification is handled externally of the cluster by a dedicated Zookeeper instance. However, merely increasing the number of nodes in the Kafka cluster is not scaling the throughput of a topic directly, although it is a prerequisite to be able to scale the throughput of topics.

Topics are actually scaled using a concept called *partitions*, which is similar to MongoDB's sharding-concept. Each topic can have one or more partitions and messages are

## 6 Discussion

assigned to exactly one of these according to the message-key. Adding more partitions is therefore theoretically providing more throughput for the topic, in case of a well-chosen partitioning strategy that is able to distribute the messages equally between partitions. The partitions are itself distributed across Kafka Brokers within the cluster. If not enough Brokers are present, one Broker will get multiple partitions, which is not effective in terms of the topic throughput. But it can still be a valid option to have more partitions than Broker-nodes to have the ability to scale the client applications that consume from Kafka topics or produce to them, like *Kafka-Streams* or *Kafka-Connect* do.

### 6.2.3.1 Kafka-Streams

Since Kafka-Streams is merely a library to implement custom microservices for stream processing, scalability is at first relying on the scalability of these microservices. We have to be able to horizontally scale these services to process a larger amount of data. As mentioned before, partitions are the key concept to scale topic throughput but also the topic-dependent components like stream processors using the Kafka-Streams library. We are able to deploy multiple instances of the same stream processing application, which will form a *Consumer Group* and internally coordinate all instances to distribute the responsibility for available partitions equally between all consuming stream processing services. Each partition is exactly assigned to one instance, which limits the degree of parallelism to the number of partitions a topic has. If a consumer is added or removed from the group, partition reassignment is performed [56].

### 6.2.3.2 Kafka-Connect

The same principles of Consumer Groups and Consumers is applied in Kafka-Connect, where a group of worker nodes is performing *Tasks* defined by a specific Connector configuration. The degree of parallelism is defined by the number of Tasks running for one Connector configuration, which would be for example moving data of one specific Topic into a database. As with Consumers in Consumer Groups, the partitions of a topic are assigned to one of the Tasks. The Tasks are transparently distributed between the



group of workers, which can be increased or decreased and coordinate themselves via “internal” Kafka topics.

### 6.2.4 Cloud-Native Cluster

The previous sections have shown, that all our core technologies and components are able to provide horizontal scalability and therefore are able to cope with an increasing number of users and measurement data. However, to actually be able to scale services and components horizontally, an appropriate infrastructure is needed. The cloud-native concept we used throughout our design is providing a combination of infrastructure, automation and decoupled design, that is enabling infrastructure agnostic horizontal scalability of all components. All components of the previous sections are operated and scaled within the Kubernetes cluster, which abstracts away certain infrastructural notions such as specific nodes and service discovery.

The scalability of the overall system is therefore fundamentally dependent on the scalability of the Kubernetes cluster, because all components share the computing resources available within the cluster. Since Kubernetes clusters can easily be extended with further worker-nodes and a scheduler is transparently handling the placement of the components within the cluster, it is satisfying the fundamental requirement of being able to increase computing resources available to the application components.

## 6.3 Requirements Comparison

We specified certain requirements for the Noisemap Project, introduced in Section 3.6, which have been used in Chapter 4 to identify certain generic capabilities, that a platform for crowdsensing of geospatial data should have. The proposed design has been used to implement a prototypic version for parts of the platform. Table 6.1 and Table 6.2 briefly discuss how the design is covering different functional and non-functional requirements and especially how the prototypic Measurement Context implementation fits the Noisemap requirements.

Table 6.1: Discussion of the design in relation to functional requirements of the Noisemap-Project.

Nr.	Requirement	Description
1	Noise Level Measurements	We designed and implemented a system, that offers an HTTP-API usable by smartphones to post objective measurements.
1.1	Noise-Dose	This is currently not explicitly implemented in the backend system, since it is possible to calculate the value on the frontend device.
1.2	Triggers	Triggers have to be implemented on the frontend application. However, our system allows to specify the type that has triggered the measurement in the measurement-object, which could be used for filtering and statistics.
1.3	Subjective noise Level	This requirement has not been considered in the design, but external Frameworks and Systems like QuestionSys [69], could easily be integrated to provide a tool-set for defining and collecting questionnaires.
1.4	Offline-capability	The frontend application has the possibility to post measurements at anytime with a specified <i>created_at</i> -attribute. The stream processing phase in the Measurement Context however, will not consider late arriving measurements older than one day for aggregation in the current prototypic implementation.
2	Visualization	The system offers a data access API to request measurements and aggregations in the GeoJSON format, that can be used to visualize their values on a map. The data access API allows to request within specific geospatial boundaries allowing the frontend to request only data for the area the map is currently showing. Data access however, is dependent on certain user Member Levels because of incentive-related reasons.
2.1	Heatmap	Heatmaps are not rendered in the backend. However, a frontend application can access single measurements within a certain region and use client-side Heatmap implementations to show the data on a map.

continued ...

Table 6.1 continued: Discussion of functional requirements.

Nr.	Requirement	Description
2.2	Aggregations	Aggregations are not rendered in the backend. However, a frontend application has data access to already aggregated polygons represented in the GeoJSON format, that can be directly used for visualization on a map like shown in Figure 6.2.
3	Users	We have designed an authentication-service that offers registration and login in order to access certain other APIs of the system via a distributed concept based on JWTs.
3.1	User Profile	A user profile can be maintained in the Social Context of the system. The design proposes to use a simple HTTP API to access such profiles from a frontend application.
3.2	Groups	The Social Context is designed to provide mechanisms to form groups and discussions in a forum like manner. The geospatial context can be created by geo-tagging discussions and groups using the same geospatial indexes as the Measurement Context. Therefore, the functionality can be seamlessly integrated into visualization of a frontend application.
4	Notifications	We have not exactly specified when users get notified and how to identify when a user enters an interesting area, however we have specified and designed the technical needs within a Communication Context, which is able to send emails and push notifications to the user's device.
5	Incentives	We have proposed a system composed of two different incentive mechanisms and briefly outlined how these mechanisms could be realized within our platform design.
5.1	Restrictions / Rewards	Restrictions and Rewards are based on certain Member Levels which can be calculated within an Incentive Context using similar stream processing approaches as used in the Measurement Context. We have described two approaches for publishing Member Level state changes for users to interested services either via messaging in the backend or transportation using JWTs, which is favorable to maintain a decoupled backend design but requires more effort in frontend applications.

continued ...

## 6 Discussion

Table 6.1 continued: Discussion of functional requirements.

Nr.	Requirement	Description
5.2	Gamification	We have only briefly mentioned the possibility to implement different Challenges and collection of statistics. Such statistics could be used to award certain users with medals or pins presented in their public user profile.
6	Privacy	Privacy related features are not centrally managed, this is rather the responsibility of each context alone. We have designed a key-value based approach in the Measurement Context that allows each user to specify, who has access to certain measurement-related user information.
7	Fraud-Protection	The prototypic implementation uses only a pretty rudimentary approach to filter certain measurements based on thresholds. However, the pre-processor implementation in the stream processing pipeline can be extended to implement different analytical and pattern-based approaches to identify fraud and redirect questionable measurements into explicitly designed topics.

Table 6.2: Discussion of the design in relation to non-functional requirements of the Noisemap-Project.

Nr.	Requirement	Description
1	Efficiency	We have implemented stream processing to aggregate certain results in advance. Aggregations are favorable in terms of data size of data-requests for visualization compared to thousands of single measurements. However, we do not aggregate in every available h3-resolution in order to save required disk space. But the two available aggregations can be used as intermediates to efficiently calculate aggregates for different resolutions.
2	Modifiability	The cloud-native principles help to maintain a decoupled design. We have separated the system into different functional Bounded Contexts, which can maintain their own data model and implementation logic. The services can make use of each other's API or published messages to a Kafka topic. As long as certain interfaces are not changed, any context can extend their functionalities independently of other services, which is beneficial to provide further features in the long-term.
3	Portability	We have implemented and design the platform around cloud-native principles and package every service in a Docker container including all libraries to form a self-contained deployment unit. Configuration can happen via injected environment variables, which allows us to run the same container in different environments, such as development, integration or production clusters. Since the Docker containers provide an environment, that already satisfies the actual application code inside, the cluster scheduler can place those containers on any computing-node that has free resources.
4	Reusability	Certain parts that we have implemented as prototypes, are tailored towards noise-measurements with custom logic for averaging them. However, the overall system design is usable for any kind of geospatial crowdsensing. Other contexts have not even been specified in detail. The briefly described concept of Incentives and Social discussions can be used for crowdsensing of various environmental phenomena generically.

continued . . .

Table 6.2 continued: Discussion of non-functional requirements.

Nr.	Requirement	Description
5	Scalability	The cloud-native design and technology choices have been explicitly chosen to provide a scalable system. As discussed in Section 6.2, the conceptual design is based on a lot of cluster technologies on different layers, which can scale horizontally.
6	Verifiability	We have not specified any tests and verification methods in neither the design concept nor the prototypic implementation. However, because we use <i>continuous delivery</i> and have already specified certain pipelines in the prototypic implementation, integration of a testing phase before automatically releasing to certain environments is possible.
7	Availability	All the contexts are operated independently of each other. Even a short outage of the authentication-service should not be critical, since already issued JWTs are still valid and can be used for requests on other services. We separated ingress, processing and data access of measurement data to form distinct failure zones in the Measurement Context, in order to still be able to consume or deliver measurements when errors happen in parts of the Measurement Context.
8	Installability	This is an external requirement not specified in our backend-design, since it is subject to frontend technologies. However, we do provide a prototypic web-based implementation for visualization-only, that has no installation effort at all.
9	Integrity	The system is not designed for 100% data integrity, because we are not relying on transactions in neither the database nor the stream processing pipeline. However, all technologies are based on clusters and can replicate data to prevent loss. Additionally, a loss of a single measurement is assumed not to be critical and can be neglected.
10	Interoperability	All contexts offer standard HTTP-APIs to post and get certain data and can therefore be integrated with other contexts and external systems. Certain APIs should however only be accessible internally, because they are only designed for system-internal use like the proposed design for the Communication Context explains.

continued . . .

Table 6.2 continued: Discussion of non-functional requirements.

Nr.	Requirement	Description
11	Performance	The most critical part for frontend performance will be visualization and API requests. We have explicitly designed the system to pre-aggregate certain results in order to be able to respond quickly to the user and maintain a relatively low data size to enable fast data transfer and visualization. However, certain functionalities still require loading substantial amounts of data which can take certain time for data-transfer and visualization.
12	Reliability	Since the cloud-native concept is based on clusters, we have already certain fault-tolerant capabilities built into the system. However, the effectiveness is dependent on how well certain clusters are configured and how well the underlying cloud resources are operated. Our custom configured integration cluster did not prove to be very reliable, because of certain resource-quota limitations we have been regularly exceeding. Although, in a hosted Kubernetes environment with enough resources the system should be able to cope with certain component and node failures and reschedule the component to still working nodes in the cluster.
13	Robustness	In order to detect certain component failures, one would need to implement health checks for each service, which we did not focus on. Currently only a few components of the prototypic implementation have health checks. However, they can be easily configured in Kubernetes with certain configuration parameters, which will enable the cluster to recognize failed components and replace them.
14	Safety	There has not been any special requirement.
15	Security	We designed the system with authentication based on JWTs to secure all public facing API Endpoints. User credentials are only stored within one database, to reduce the effort for keeping that information secure.
16	Usability	This is no requirement for the backend system.

## 6.4 Food for Thoughts

The last section of this chapter is dedicated to interesting aspects that we noticed while researching, designing and implementing parts of the platform.

### 6.4.1 Secondary Benefits of the Cloud-Native Design

The decoupled design following the Microservices-Style has other benefits than providing a scalable, expendable and maintainable platform. Projects, like the *Noisemap-Project*, are often used to conduct secondary research on different related topics and aspects, especially in the scientific context. For example, one group of researchers could be interested in the noise data to draw certain conclusions about hearing loss in different regions, another research group might be interested in how different incentive mechanisms affect crowdsensing and what possibilities for implementation are well-suited. Others might be interested in social opinions and social participations around geospatial or noise-related topics. The decoupled design allows each research group to cover their own research interest in a Bounded Context and use their specific data-model and technology fitting their problem best. They can extend and deploy their own services as fast as they want and are not relying on other research groups. This would be the case in a conventional monolithic system design, where releases and code changes must be coordinated.

### 6.4.2 Fragility of the Design

While we showed, that the design is satisfying the use case and is feasible to implement, we also acknowledge that it is still in a very fragile state. This is mostly related to the fact, that Kubernetes and Kafka itself are very sophisticated and complex technologies that require a lot of knowledge for correct configuration and a stable operation. Most of the instabilities in our integration cluster are related to narrow resources on the computing-nodes for the Kubernetes cluster, like to less disk space to let Kubernetes pull and deploy further containers onto a specific node, and therefore at some point has



not been able to recover from certain connection errors of some components. This is eventually tearing down specific parts of the processing pipeline such as either Stream-Processors, Connectors or Kafka itself and therefore prevented our generated test data to be persisted into the database. We recommend to use hosted PaaS offerings for Kubernetes from one of the available cloud-providers for the operation of the system in production.

We also recommend implementing a solid monitoring and alerting concept for production use. It is important to have an overview about the state of all components and the ability to react on problems, that Kubernetes is not able to recover from. Most of the Kafka components are already exposing certain metrics via *JMX*<sup>1</sup> out of the box, that could be collected and visualized using specifically designed monitoring systems, such as *Prometheus*<sup>2</sup>.

### 6.4.3 Global Scalability

We designed the platform to be natively usable for any area worldwide due to the use of a global grid system for geospatial data. However, we did not cover how to deploy the platform at global scale across multiple data centers in various regions of the world. Briefly outlined, this would at first require mirroring the data persisted in MongoDB across the globally distributed data centers, in order to allow a European user connected to a European data-center access data collected in a northern American data-center. Furthermore, sophisticated DNS resolution techniques, such as *Global Server Load Balancing*, are required to redirect the user to the closest located data center, in order to prevent that users in Europe are redirected to a northern American data-center, which would introduce a lot of latency.

---

<sup>1</sup>Java Management Extensions (JMX) is a specification integrated into Java to make runtime aspects of applications available for monitoring.

<sup>2</sup><https://prometheus.io/>



# 7

## Conclusion

This Chapter is recapitulating the thesis with a summary in Section 7.1, where we briefly describe what we have covered. The closing outlook in Section 7.2 is providing our view on further development towards a production-ready application.

### 7.1 Summary

Our target has been to develop a conceptual design usable as a starting point for a platform-backend in the context of crowdsensing of geospatially-related data. In order to understand the subject and technological backgrounds, we introduced and defined important terms and concepts. To develop the generic use case, we described the Noisemap project as a specific use case in crowdsensing of geospatially-related data. The overall objective of the project is to use the smartphones of the crowd to measure noise, then use their internal GPS capabilities to relate the produced data to geographic space, send the data to a backend, where it can be stored, combined with other data and provided via an API for data visualization on the user's device. We used the refined requirements of that project to derive capabilities for a general platform-design capable of dealing with geospatial data. The design uses a state-of-the-art cloud-native approach to create a decoupled architecture in form of Microservices, that are hosted by highly automated infrastructure and deployment concepts. The processing of incoming data is designed to be performed by a stream processing methodology, in this case precisely implemented with the help of the Kafka ecosystem and its provided libraries to create the stream processing business logic. Geospatial capabilities are covered with the help of the H3 library, which implements a discrete global grid system that can

## 7 Conclusion

be used to aggregate and index geospatial data globally without manual effort. We have additionally briefly described, how the different crowdsensing specific requirements, such as social features and incentive management, can be realized as distinct bounded context according to the design principles.

The prototypic implementation of the measurement context is showing the feasibility of the stream processing methodology, but the operation in the test cluster also revealed fragility when the infrastructure is not professionally operated and monitored, as discussed in Section 6.4. Finally, the scalability is reasoned by forming a scalability-chain of components dependent on another and describe their specific technical scalability to show, that the overall concept is providing sufficient scalability with the proposed technology.

## 7.2 Outlook

We acknowledge, that the platform is not specified detailed enough to be instantly used for a production use case. Further research and specification has to be done to implement a minimum viable product, that is actually usable by the crowd.

For the frontend applications of various sensing applications it is crucial to know the context of the device for opportunistic sensing actions. Section 3.7 mentioned some of the challenges particularly for noise sensing, but they are still relevant for other environmental sensing use cases as well. The goal of such research must be a framework for mobile developers, helping them to easily access the current situation via a context API. For example, this could be an enhanced form of something like the *Google Awareness API*<sup>1</sup> for Android, that unifies location, sensor and context signals to detect the user status, like walking or driving, as well as the weather conditions at the current location.

Future topics regarding the backend system are most importantly related to further specifying the different bounded contexts and implementing first prototypes of them. For example prototypic implementations of the Incentive and Social Context will unveil how well the measurement data in Kafka is usable for other contexts and whether

---

<sup>1</sup><https://developers.google.com/awareness/>

the feature separation of our design into the different contexts has been optimal. As mentioned in Section 6.4.1, such further research and specification can be performed by different groups independently to some degree. We recommend to involve corresponding experts for refining the remaining capabilities. Furthermore, infrastructural refinement is necessary as acknowledged in Section 6.4.2 and Section 6.4.3. There must be a suitable monitoring solution to support maintenance and ensure that all the automation that is related to DevOps principles does not harm operational stability. Also, further reflections regarding global scalability will be needed when such a platform is starting to become successful in some time.

Taken together all covered aspects and remaining question marks of this thesis, we believe that such a platform can support transparency of environmental conditions by unleashing the crowd's potential. But refinement for different use cases and of course a natural user experience in frontend applications is needed, to make participating for users a great experience of social participation and environmental awareness.



# Bibliography

- [1] *Apache Kafka Documentation*. <https://kafka.apache.org/documentation.html> Last Accessed at: 2018-08-26
- [2] *WGS 84 - WGS84 - World Geodetic System 1984, used in GPS - EPSG:4326*. <https://epsg.io/4326> Last Accessed at: 2018-09-23
- [3] ACOUSTICATOR: *Illustrates how sound pressure waves are converted to an electrical signal and displayed as a sound pressure level on a sound level meter*. [https://commons.wikimedia.org/wiki/File:Sound\\_level\\_meter\\_with\\_sound\\_waves.png](https://commons.wikimedia.org/wiki/File:Sound_level_meter_with_sound_waves.png). Version: Jan. 2016. – Last Accessed at: 2018-08-10
- [4] AITCHISON, A. : *Spatial Reference Systems*. Version:2012. [http://dx.doi.org/10.1007/978-1-4302-3492-0\\_1](http://dx.doi.org/10.1007/978-1-4302-3492-0_1). In: AITCHISON, A. (Hrsg.): *Pro Spatial with SQL Server 2012*. Apress. – DOI 10.1007/978-1-4302-3492-0\_1. – ISBN 978-1-4302-3492-0, 1-20
- [5] ALEX CONTINI: *Static Cloud Native Landscape*. <https://github.com/cncf/landscape>. Version: Jul. 2018. – Last Accessed at: 2018-07-12
- [6] AMERICAN NATIONAL STANDARD: *Specification for Sound Level Meters, American National Standards Institute ANSI S1.4-1983*. <https://law.resource.org/pub/us/cfr/ibr/002/ansi.s1.4.1983.pdf>. Version: 1992. – Last Accessed at: 2018-11-12
- [7] ASBACH, C. ; HELLACK, B. ; SCHUMACHER, S. ; BÄSSLER, M. ; SPREITZER, M. ; POHL, T. ; WEBER, K. ; MONZ, C. ; BIEDER, S. ; SCHULTZE, T. ; TODEA, M. : *Anwendungsmöglichkeiten und Grenzen kostengünstiger Feinstaubsensoren*. (2018), S. 10
- [8] BAIER, J. : *Getting Started with Kubernetes, Second Edition*. s.l : Packt Publishing, 2017. – ISBN 978-1-78728-336-7. – OCLC: 992151361

## Bibliography

- [9] BEDA, J. : *Core Kubernetes: Jazz Improv over Orchestration*. <https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca>.  
Version: Mai 2017. – Last Accessed at: 2018-10-09
- [10] BERGER, E. H.: *The Noise Manual*. AIHA, 2003. – ISBN 978–1–931504–02–7. – Google-Books-ID: hCB1LYIGgdcC
- [11] BIRK, F. : *Microservices - Eine State-of-the-Art Bestandsaufnahme und Abgrenzung zu SOA*, Ulm University, Bachelor Thesis, Dez. 2016. <http://dbis.eprints.uni-ulm.de/1460/>
- [12] BRADLEY, J. ; SAKIMURA, N. ; JONES, M. : JSON Web Token (JWT) / Internet Engineering Task Force (IETF). Version: Mai 2015. <https://tools.ietf.org/html/rfc7519> (rfc7519). – Request for Comment (RFC). – Last Accessed at: 2018-08-15
- [13] BRIKMAN, Y. : *Terraform: Up and Running: Writing Infrastructure as Code*. "O'Reilly Media, Inc.", 2017. – ISBN 978–1–4919–7705–7. – Google-Books-ID: rn1VDgAAQBAJ
- [14] BRODSKY, I. : *H3: Uber's Hexagonal Hierarchical Spatial Index*. <https://eng.uber.com/h3/>. Version: Jun. 2018. – Last Accessed at: 2018-09-10
- [15] BURNS, B. ; GRANT, B. ; OPPENHEIMER, D. ; BREWER, E. ; WILKES, J. : Borg, omega, and kubernetes. In: *Queue* 14 (2016), Nr. 1, S. 10
- [16] BUSINESS OF APPS: *Prognose zur Anzahl der Smartphone-Nutzer weltweit von 2012 bis 2021 (in Milliarden)*. <https://de.statista.com/statistik/daten/studie/309656/umfrage/prognose-zur-anzahl-der-smartphone-nutzer-weltweit/>. – Last Accessed at: 2018-12-02
- [17] CHUCK KARDOUS ; CHRISTA L. THEMANN ; THAIS C. MORATA ; W. GREGORY LOTZ: *Understanding Noise Exposure Limits: Occupational vs. General Environmental Noise || Blogs | CDC*. Version: Febr. 2016. <https://blogs.cdc.gov/niosh-science-blog/2016/02/08/noise/> Last Accessed at: 2018-09-06



- [18] CNFC-TECHNICAL OVERSIGHT COMMITTEE (TOC): *What is cloud native?* Version: Jun. 2018. <https://www.cncf.io/about/faq/> Last Accessed at: 2018-07-11
- [19] CONCHA-BARRIENTOS, M. ; CAMPBELL-LENDRUM, D. ; STEENLAND, K. : *Occupational noise: assessing the burden of disease from work-related hearing impairment at national and local levels*. Geneva : World Health Organization, 2004. – ISBN 978–92–4–159192–8. – OCLC: 165043430
- [20] COREY, J. : *Dynamic Range Control. 2*. Focal Press [http://proquest.tech.safaribooksonline.de/9781317541042/cover\\_xhtml](http://proquest.tech.safaribooksonline.de/9781317541042/cover_xhtml). – ISBN 978–1–138–20142–2
- [21] CRANSTON, C. J. ; BRAZILE, W. J. ; SANDFORT, D. R. ; GOTSHALL, R. W.: Occupational and recreational noise exposure from indoor arena hockey games. In: *Journal of Occupational and Environmental Hygiene* 10 (2013), Nr. 1, S. 11–16. <http://dx.doi.org/10.1080/15459624.2012.736341>. – DOI 10.1080/15459624.2012.736341. – ISSN 1545–9632
- [22] DE ROECK, D. ; SLEGGERS, K. ; CRIEL, J. ; GODON, M. ; CLAEYS, L. ; KILPI, K. ; JACOBS, A. : I would DiYSE for it!: a manifesto for do-it-yourself internet-of-things creation. In: *Proceedings of the 7th Nordic Conference on Human-Computer Interaction Making Sense Through Design - NordiCHI '12*. ACM Press. – ISBN 978–1–4503–1482–4, 170
- [23] DOCUMENTATION, K. S.: *Streams DSL*. <https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html#windowing> Last Accessed at: 2018-10-18
- [24] ESTELLÉS-AROLAS, E. ; GONZÁLEZ-LADRÓN-DE-GUEVARA, F. : Towards an integrated crowdsourcing definition. In: *Journal of Information science* 38 (2012), Nr. 2, S. 189–200
- [25] EVANS, E. : *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1 edition. Boston : Addison-Wesley Professional, 2003. – ISBN 978–0–321–12521–7

## Bibliography

- [26] EVANS, E. : *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014. – ISBN 978–1–4575–0119–7. – Google-Books-ID: ccRsBgAAQBAJ
- [27] FOWLER, M. ; LEWIS, J. : *Microservices - a definition of this new architectural term*. <http://martinfowler.com/articles/microservices.html>. Version: März 2014. – Last Accessed at: 2018-09-21
- [28] GANTI, R. K. ; YE, F. ; LEI, H. : Mobile crowdsensing: current state and future challenges. In: *IEEE Communications Magazine* 49 (2011), Nr. 11
- [29] GARTNER: *What Is Big Data? - Gartner IT Glossary - Big Data*. <https://www.gartner.com/it-glossary/big-data/> Last Accessed at: 2018-09-18
- [30] GERMANY, C. for: *Stuttgart – Making Air Pollution Visible*. <http://codefor.de/en/stadtgeschichten/1feinstaub/>. Version: Dez. 2016. – Last Accessed at: 2018-10-20
- [31] GILLIES, S. ; BUTLER, H. ; DALY, M. ; DOYLE, A. ; SCHAUB, T. : The GeoJSON Format / Internet Engineering Task Force (IETF). Version: Aug. 2016. <https://tools.ietf.org/html/rfc7946> (RFC7946). – Request for Comments (RFC). – Last Accessed at: 2018-08-28
- [32] GUBBI, J. ; BUYYA, R. ; MARUSIC, S. ; PALANISWAMI, M. : Internet of Things (IoT): A vision, architectural elements, and future directions. In: *Future Generation Computer Systems* 29 (2013), Sept., Nr. 7, 1645–1660. <http://dx.doi.org/10.1016/j.future.2013.01.010>. – DOI 10.1016/j.future.2013.01.010. – ISSN 0167–739X
- [33] HACKERSPACES.ORG: *Hackerspaces*. <http://hackerspaces.org/>. – Last Accessed at: 2018-10-20
- [34] HARDT, D. : The OAuth 2.0 Authorization Framework / Internet Engineering Task Force (IETF). Version: Okt. 2012. <https://tools.ietf.org/html/rfc6749> (rfc6749). – Request for Comment (RFC). – Last Accessed at: 2018-08-15

- [35] HARDT, D. ; JONES, M. : The OAuth 2.0 Authorization Framework: Bearer Token Usage / Internet Engineering Task Force (IETF). Version: Okt. 2012. <https://tools.ietf.org/html/rfc6750> (rfc6750). – Request for Comment (RFC). – Last Accessed at: 2018-08-16
- [36] HERBST, N. R. ; KOUNEV, S. ; REUSSNER, R. H.: Elasticity in Cloud Computing: What It Is, and What It Is Not. 13 (2013), S. 23–27
- [37] HOWE, J. : The Rise of Crowdsourcing. In: *Wired* (2006), Jun. <https://www.wired.com/2006/06/crowds/>. – ISSN 1059–1028
- [38] HOWE, J. : *Crowdsourcing: How the power of the crowd is driving the future of business*. Random House, 2008
- [39] HUMBLE, J. ; MOLESKY, J. ; O'REILLY, B. : *Lean Enterprise: How High Performance Organizations Innovate at Scale*. "O'Reilly Media, Inc.", 2014. – ISBN 978–1–4919–4655–8. – Google-Books-ID: ivixBQAAQBAJ
- [40] INC., O. G. C. ; JOHN R. HERRING (Hrsg.): OpenGIS® Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture / Open Geospatial Consortium Inc. Version: Mai 2011. [http://portal.opengeospatial.org/files/?artifact\\_id=25355](http://portal.opengeospatial.org/files/?artifact_id=25355) (OGC 06-103r4 Version 1.2.1). – OpenGIS® Implementation Standard. – Last Accessed at: 2018-12-12
- [41] IWESS: *Deutsch: Sensor zur Feinstaubmessung und weitere Komponenten des Feinstaubsensoren-Bausatzes für Citizen Scientist beim Workshop von luftdaten.info auf dem 33c3 in Hamburg 2016*. [https://commons.wikimedia.org/wiki/File:Komponenten\\_des\\_Feinstaubsensoren-Bausatzes\\_beim\\_Workshop\\_auf\\_dem\\_33c3\\_2016.jpg](https://commons.wikimedia.org/wiki/File:Komponenten_des_Feinstaubsensoren-Bausatzes_beim_Workshop_auf_dem_33c3_2016.jpg). Version: Dez. 2016. – Last Accessed at: 2018-10-26
- [42] KRAFT, R. : *NY NoiseMap App*. Ulm, 2018. – Developed at Databases and Information Systems Department

## Bibliography

- [43] KRATZKE, N. ; QUINT, P.-C. : Understanding cloud-native applications after 10 years of cloud computing - A systematic mapping study. In: *Journal of Systems and Software* 126 (2017), Apr., 1–16. <http://dx.doi.org/10.1016/j.jss.2017.01.001>. – DOI 10.1016/j.jss.2017.01.001. – ISSN 01641212
- [44] KUBERNETES: *Concepts Underlying the Cloud Controller Manager*. Version: Sept. 2018. <https://kubernetes.io/docs/concepts/architecture/cloud-controller/> Last Accessed at: 2018-10-09
- [45] "LANDESAMT FÜR DIGITALISIERUNG, BREITBAND UND VERMESSUNG": *Bayrische Vermessungsverwaltung - Über uns - Zentrale Stelle Hauskoordinaten und Hausumringe (ZSHH)*. <https://www.ldbv.bayern.de/ueberuns/zshh.html>. – Last Accessed at: 2018-09-29
- [46] LANE, N. D. ; EISENMAN, S. B. ; MUSOLESI, M. ; MILUZZO, E. ; CAMPBELL, A. T.: Urban sensing systems: opportunistic or participatory? In: *Proceedings of the 9th workshop on Mobile computing systems and applications* ACM, 2008, S. 11–16
- [47] LE PRELL, C. G.: Potential contributions of recreational noise to daily noise dose. In: *CAOHC Update* 28 (2016), Nr. 1, S. 1
- [48] LINDTNER, S. : Hackerspaces and the Internet of Things in China: How makers are reinventing industrial production, innovation, and the self. In: *China Information* 28 (2014), Jul., Nr. 2, 145–167. <http://dx.doi.org/10.1177/0920203X14529881>. – DOI 10.1177/0920203X14529881. – ISSN 0920–203X, 1741–590X
- [49] MARTIN, R. C.: *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002
- [50] MCLEOD, K. S.: Our sense of Snow: the myth of John Snow in medical geography. In: *Social Science & Medicine* 50 (2000), Apr., Nr. 7-8, 923–935. [http://dx.doi.org/10.1016/S0277-9536\(99\)00345-7](http://dx.doi.org/10.1016/S0277-9536(99)00345-7). – DOI 10.1016/S0277-9536(99)00345-7. – ISSN 02779536
- [51] MELL, P. ; GRANCE, T. u. a.: The NIST definition of cloud computing. In: *National institute of standards and technology* 53 (2009), Nr. 6, S. 50

- [52] MERKEL, D. : Docker: Lightweight Linux Containers for Consistent Development and Deployment. In: *Linux J.* 2014 (2014), März, Nr. 239. <http://dl.acm.org/citation.cfm?id=2600239.2600241>. – ISSN 1075–3583
- [53] MONGODB, INC: *Sharding — MongoDB Manual*. Version:2018. <https://docs.mongodb.com/manual/sharding> Last Accessed at: 2018-11-04
- [54] MURPHY, E. ; KING, E. A.: Testing the accuracy of smartphones and sound level meter applications for measuring environmental noise. In: *Applied Acoustics* 106 (2016), Mai, 16–22. <http://dx.doi.org/10.1016/j.apacoust.2015.12.012>. – DOI 10.1016/j.apacoust.2015.12.012. – ISSN 0003–682X
- [55] NADAREISHVILI, I. ; MITRA, R. ; MCLARTY, M. ; AMUNDSEN, M. : *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016
- [56] NARKHEDE, N. ; SHAPIRA, G. ; PALINO, T. : *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. O'Reilly Media, Inc., 2017. – ISBN 978–1–4919–3613–9. – Google-Books-ID: a3wzDwAAQBAJ
- [57] NAT SAKIMURA ; JOHN BRADLEY ; MICHAEL B. JONES ; BRENO DE MEDEIROS ; CHUCK MORTIMORE: *Final: OpenID Connect Core 1.0*. [http://openid.net/specs/openid-connect-core-1\\_0.html](http://openid.net/specs/openid-connect-core-1_0.html). Version:2014. – Last Accessed at: 2018-08-15
- [58] NIKOLIC, M. ; BIERLAIRE, M. : Review of transportation mode detection approaches based on smartphone data. (2017), S. 20
- [59] OPEN CONTAINER INITIATIVE: *Open Container Initiative Runtime Specification / Version 1.0.0*. <https://github.com/opencontainers/runtime-spec/releases/download/v1.0.0/oci-runtime-spec-v1.0.0.html>. Version: Jul. 2017. – Last Accessed at: 2018-08-11
- [60] PRESS, O. U.: *Oxford Dictionaries | English*. <https://en.oxforddictionaries.com/definition/noise>. Version:2018. – Last Accessed at: 2018-08-02

## Bibliography

- [61] PRYSS, R. ; PROBST, T. ; SCHLEE, W. ; SCHOBEL, J. ; LANGGUTH, B. ; NEFF, P. ; SPILIOPOULOU, M. ; REICHERT, M. : Mobile Crowdsensing for the Juxtaposition of Realtime Assessments and Retrospective Reporting for Neuropsychiatric Symptoms. In: *2017 IEEE 30th International Symposium on Computer-Based Medical Systems (CBMS)*. IEEE. – ISBN 978-1-5386-1710-6, 642–647
- [62] PRYSS, R. ; REICHERT, M. ; SCHLEE, W. ; SPILIOPOULOU, M. ; LANGGUTH, B. ; PROBST, T. : Differences between Android and iOS Users of the TrackYourTinnitus Mobile Crowdsensing mHealth Platform. In: *2018 IEEE 31st International Symposium on Computer-Based Medical Systems (CBMS)* IEEE, 2018, S. 411–416
- [63] PRYSS, R. ; SCHLEE, W. ; LANGGUTH, B. ; REICHERT, M. : Mobile Crowdsensing Services for Tinnitus Assessment and Patient Feedback. In: *AI & Mobile Services (AIMS), 2017 IEEE International Conference on IEEE*, 2017, S. 22–29
- [64] PRYSS, R. ; SCHOBEL, J. ; REICHERT, M. : Requirements for a Flexible and Generic API Enabling Mobile Crowdsensing mHealth Applications. In: *2018 4th International Workshop on Requirements Engineering for Self-Adaptive, Collaborative, and Cyber Physical Systems (RESACS)* IEEE, 2018, S. 24–31
- [65] PRYSS, R. ; PROBST, T. ; SCHLEE, W. ; SCHOBEL, J. ; LANGGUTH, B. ; NEFF, P. ; SPILIOPOULOU, M. ; REICHERT, M. : Prospective crowdsensing versus retrospective ratings of tinnitus variability and tinnitus–stress associations based on the TrackYourTinnitus mobile platform. In: *International Journal of Data Science and Analytics* (2018), März. <https://link.springer.com/article/10.1007/s41060-018-0111-4#citeas>. – ISSN 2364–4168
- [66] PURSS, M. : Topic 21: Discrete Global Grid Systems Abstract Specification / Open Geospatial Consortium (OGC). Version: Aug. 2017. <http://docs.opengeospatial.org/as/15-104r5/15-104r5.html> (Version: 1.0). – Abstract Specification. – Last Accessed at: 2018-10-23
- [67] SAHR, K. ; WHITE, D. ; KIMERLING, A. J.: Geodesic discrete global grid systems. In: *Cartography and Geographic Information Science* 30 (2003), Nr. 2, S. 121–134

- [68] SCHLEE, W. ; PRYSS, R. C. ; PROBST, T. ; SCHOBEL, J. ; BACHMEIER, A. ; REICHERT, M. ; LANGGUTH, B. : Measuring the Moment-to-Moment Variability of Tinnitus: The TrackYourTinnitus Smart Phone App. In: *Frontiers in Aging Neuroscience* 8 (2016). <http://dx.doi.org/10.3389/fnagi.2016.00294>. – DOI 10.3389/fnagi.2016.00294. – ISSN 1663–4365
- [69] SCHOBEL, J. ; PRYSS, R. ; SCHICKLER, M. ; REICHERT, M. : A Configurator Component for End-User Defined Mobile Data Collection Processes. In: DRIRA, K. (Hrsg.) ; WANG, H. (Hrsg.) ; YU, Q. (Hrsg.) ; WANG, Y. (Hrsg.) ; YAN, Y. (Hrsg.) ; CHAROY, F. (Hrsg.) ; MENDLING, J. (Hrsg.) ; MOHAMED, M. (Hrsg.) ; WANG, Z. (Hrsg.) ; BHIRI, S. (Hrsg.): *Service-Oriented Computing – ICSSOC 2016 Workshops*, Springer International Publishing, 2017 (Lecture Notes in Computer Science). – ISBN 978–3–319–68136–8, S. 216–219
- [70] SCHWEIZER, I. ; BÄRTL, R. ; SCHULZ, A. ; PROBST, F. ; MÜHLÄUSER, M. : NoiseMap-real-time participatory noise maps. In: *Second International Workshop on Sensing Applications on Mobile Phones Citeseer*, 2011, S. 1–5
- [71] STENNETH, L. ; WOLFSON, O. ; YU, P. S. ; XU, B. : Transportation mode detection using mobile phones and GIS information. In: *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems ACM*, 2011, S. 54–63
- [72] TINGAY, J. ; ROBINSON, D. : A practical comparison of occupational noise standards. (2014), S. 8
- [73] TOM RUEN: *Geodesic polyhedron as subdivided icosahedron*. [https://commons.wikimedia.org/wiki/File:Geodesic\\_icosahedral\\_polyhedron\\_example.png](https://commons.wikimedia.org/wiki/File:Geodesic_icosahedral_polyhedron_example.png). Version: Febr. 2017. – Last Accessed at: 2018-11-23
- [74] TRULS GJESTLAND: *Background noise levels in Europe / SINTEF ICT*. Version: Jun. 2008. [https://www.easa.europa.eu/sites/default/files/dfu/Background\\_noise\\_report.pdf](https://www.easa.europa.eu/sites/default/files/dfu/Background_noise_report.pdf). SINTEF ICT (SINTEF A6631). – Forschungsbericht. – 25 S. – Last Accessed at: 2018-08-08

## Bibliography

- [75] VAQUERO, L. M. ; RODERO-MERINO, L. ; CACERES, J. ; LINDNER, M. : A break in the clouds: towards a cloud definition. In: *ACM SIGCOMM Computer Communication Review* 39 (2008), Dez., Nr. 1, 50. <http://dx.doi.org/10.1145/1496091.1496100>. – DOI 10.1145/1496091.1496100. – ISSN 01464833
- [76] VELTE, A. T. ; VELTE, T. J. ; ELSENPETER, R. C.: *Cloud computing a practical approach*. McGraw-Hill <http://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=291254>. – ISBN 978–0–07–162695–8. – OCLC: 495705646
- [77] VERMA, A. ; PEDROSA, L. ; KORUPOLU, M. ; OPPENHEIMER, D. ; TUNE, E. ; WILKES, J. : Large-scale cluster management at Google with Borg. In: *Proceedings of the Tenth European Conference on Computer Systems ACM*, 2015, S. 18
- [78] WANG, W. ; CHANG, Q. ; LI, Q. ; SHI, Z. ; CHEN, W. : Indoor-Outdoor Detection Using a Smart Phone Sensor. In: *Sensors (Basel, Switzerland)* 16 (2016), Sept., Nr. 10. <http://dx.doi.org/10.3390/s16101563>. – DOI 10.3390/s16101563. – ISSN 1424–8220
- [79] WIEGERS, K. E. ; BEATTY, J. : *Software requirements*. 3rd ed. Redmond, WA : Microsoft Press, 2013. – ISBN 978–0–7356–7965–8
- [80] WIXTED, A. J. ; KINNAIRD, P. ; LARIJANI, H. ; TAIT, A. ; AHMADINIA, A. ; STRACHAN, N. : Evaluation of LoRa and LoRaWAN for wireless sensor networks. In: *2016 IEEE SENSORS*, 2016, S. 1–3
- [81] WOLFF, E. : *Microservices: Grundlagen flexibler Softwarearchitekturen*. 1. Auflage. Heidelberg : dpunkt.verlag, 2016. – ISBN 978–3–86490–313–7
- [82] YANG, J. ; MUNGUIA-TAPIA, E. ; GIBBS, S. : Efficient in-pocket detection with mobile phones. In: *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication - UbiComp '13 Adjunct*. ACM Press. – ISBN 978–1–4503–2215–7, 31–34



- [83] ZAPPATORE, M. ; LONGO, A. ; BOCHICCHIO, M. A.: Crowd-sensing our Smart Cities: a Platform for Noise Monitoring and Acoustic Urban Planning. 13 (2017), Jun., Nr. 2, 53–67. <http://dx.doi.org/10.24138/jcomss.v13i2.373>. – DOI 10.24138/jcomss.v13i2.373. – ISSN 1845–6421
- [84] ZHANG, X. ; YANG, Z. ; SUN, W. ; LIU, Y. ; TANG, S. ; XING, K. ; MAO, X. : Incentives for Mobile Crowd Sensing: A Survey. In: *IEEE Communications Surveys & Tutorials* 18 (2016), Nr. 1, 54–67. <http://dx.doi.org/10.1109/COMST.2015.2415528>. – DOI 10.1109/COMST.2015.2415528. – ISSN 1553–877X
- [85] ZHOU, P. ; ZHENG, Y. ; LI, Z. ; LI, M. ; SHEN, G. : Iodetector: A generic service for indoor outdoor detection. In: *Proceedings of the 10th acm conference on embedded network sensor systems* ACM, 2012, S. 113–126



# A

## Appendix

Listing A.1: Example of a Kubernetes cluster configuration to setup the cluster.

```
1 addon_job_timeout: 30
2 authentication:
3   strategy: "x509"
4 bastion_host:
5   ssh_agent_auth: false
6 cloud_provider:
7   name: "openstack"
8   openstackCloudProvider:
9     block_storage: #Allows Kubernetes to create dynamic volumes and attach
10      them to Pods
11     ignore-volume-az: true
12     trust-device-path: false
13   global:
14     auth-url: "https://idm01.bw-cloud.org:5000/v3"
15     domain-name: "Default"
16     password: "<xSecretx>"
17     region: "Ulm"
18     tenant-id: "<Tenant-UUID-String>"
19     username: "<user>@<domain>.de"
20   load_balancer:
21     create-monitor: false
22     manage-security-groups: false
23     monitor-delay: 0
24     monitor-max-retries: 0
25     monitor-timeout: 0
26     use-octavia: false
27   metadata:
28     request-timeout: 0
```

## A Appendix

```
28 ignore_docker_version: true
29 ingress:
30   node_selector: #Specifies a Node-Selector for the Ingress-Container
31     app: "ingress"
32   provider: "nginx"
33 kubernetes_version: "v1.11.2-rancher1-1"
34 monitoring:
35   provider: "metrics-server"
36 network: #Defines the Network Type (others are: calico, flannel, weave ...)
37   plugin: "canal"
38 services:
39   etcd: #Important for Cluster Coordination of Kubernetes
40     extra_args:
41       election-timeout: "5000"
42       heartbeat-interval: "500"
43     snapshot: false
44   kube-api:
45     pod_security_policy: false
46   kubelet:
47     fail_swap_on: false
48 ssh_agent_auth: false
```

Listing A.2: Terraform File describing the infrastructure via structured text.

```
1 provider "openstack" {
2   user_name   = "${var.user_name}"
3   tenant_name = "${var.tenant_name}"
4   password   = "${var.password}"
5   auth_url   = "${var.auth_url}"
6   region     = "${var.region}"
7 }
8
9 resource "openstack_networking_network_v2" "kube" {
10  name          = "kube"
11  region       = "${var.region}"
12  admin_state_up = "true"
13 }
14 resource "openstack_compute_keypair_v2" "kube" {
15  name          = "SSH keypair for kube instances"
16  region       = "${var.region}"
17  public_key   = "${file("${var.ssh_key_file}.pub")}"
18 }
19
20 resource "openstack_networking_subnet_v2" "kube" {
21  name          = "kube"
22  region       = "${var.region}"
23  network_id   = "${openstack_networking_network_v2.kube.id}"
24  cidr         = "${var.tenant_net_cidr}"
25  ip_version   = 4
26  enable_dhcp  = "true"
27 }
28
29 resource "openstack_networking_router_v2" "kube" {
30  name          = "kube"
31  region       = "${var.region}"
32  admin_state_up = "true"
33  external_network_id = "${var.external_gateway}"
34 }
35
36 resource "openstack_networking_router_interface_v2" "kube" {
37  region = "${var.region}"
```

## A Appendix

```
38   router_id = "${openstack_networking_router_v2.kube.id}"
39   subnet_id = "${openstack_networking_subnet_v2.kube.id}"
40 }
41
42 resource "openstack_networking_floatingip_v2" "float" {
43   count = 1
44   depends_on = ["openstack_networking_router_interface_v2.kube"]
45   region    = "${var.region}"
46   pool      = "${var.pool}"
47 }
48
49 resource "openstack_networking_floatingip_v2" "float-worker" {
50   depends_on = ["openstack_networking_router_interface_v2.kube"]
51   region    = "${var.region}"
52   pool      = "${var.pool}"
53 }
54 resource "openstack_compute_secgroup_v2" "kube" {
55   name          = "kube"
56   region       = "${var.region}"
57   description   = "Security group for the kube instances"
58
59   rule {
60     from_port = 1
61     to_port   = 65535
62     ip_protocol = "tcp"
63     cidr       = "0.0.0.0/0"
64   }
65
66   rule {
67     from_port = 1
68     to_port   = 65535
69     ip_protocol = "udp"
70     cidr       = "0.0.0.0/0"
71   }
72
73   rule {
74     ip_protocol = "icmp"
75     from_port   = "-1"
```

```

76     to_port      = "-1"
77     cidr         = "0.0.0.0/0"
78   }
79 }
80
81 resource "openstack_compute_floatingip_associate_v2" "kube_floating_master" {
82   floating_ip = "${openstack_networking_floatingip_v2.float.*.address[0]}"
83   instance_id = "${openstack_compute_instance_v2.kube-master.id}"
84 }
85
86 resource "openstack_compute_instance_v2" "kube-master" {
87   name          = "kube-master"
88   region        = "${var.region}"
89   image_id      = "${var.image_id}"
90   flavor_name   = "${var.master_flavor}"
91   key_pair      = "${openstack_compute_keypair_v2.kube.name}"
92   security_groups = ["${openstack_compute_secgroup_v2.kube.name}"]
93
94   network {
95     uuid = "${openstack_networking_network_v2.kube.id}"
96   }
97 }
98
99 resource "null_resource" "cluster" {
100   depends_on=["openstack_networking_floatingip_v2.float", "
101             openstack_compute_instance_v2.kube-master" ]
102
103   provisioner "remote-exec" {
104     connection {
105       user          = "${var.ssh_user_name}"
106       private_key   = "${file("${var.ssh_key_file}")}"
107       host          = "${openstack_networking_floatingip_v2.float.*.address[0]}"
108     }
109     inline = [
110       "docker run -d --name rancher-master --restart=unless-stopped -p
111         8080:80 -p 8443:443 -v /host/rancher:/var/lib/rancher rancher/
112         rancher:latest"
113     ]
114   }
115 }

```

## A Appendix

```
111 }
112 }
113
114 resource "openstack_compute_floatingip_associate_v2" "kube_floating_worker" {
115     floating_ip = "${openstack_networking_floatingip_v2.float-worker.address}"
116     instance_id = "${openstack_compute_instance_v2.rancher-worker-ingress.id}"
117 }
118
119 resource "openstack_compute_instance_v2" "rancher-worker-ingress" {
120     name          = "rancher-worker-ingress-${count.index}"
121     count         = "${var.worker_ingress_count}"
122     region       = "${var.region}"
123     image_id     = "${var.image_id}"
124     flavor_name  = "${var.worker_flavor}"
125     key_pair     = "${openstack_compute_keypair_v2.kube.name}"
126     security_groups = ["${openstack_compute_secgroup_v2.kube.name}"]
127     depends_on   = ["openstack_compute_instance_v2.kube-master"]
128
129     network {
130         uuid = "${openstack_networking_network_v2.kube.id}"
131     }
132 }
133
134 resource "null_resource" "worker-ingress" {
135     depends_on=["openstack_compute_instance_v2.rancher-worker-ingress"]
136     triggers {
137         cluster_instance = "${openstack_compute_instance_v2.rancher-worker-
138             ingress.id}"
139     }
140     provisioner "remote-exec" {
141
142         connection {
143             type          = "ssh"
144             user          = "${var.ssh_user_name}"
145             private_key   = "${file("${var.ssh_key_file}")}"
146             host          = "${openstack_compute_instance_v2.rancher-worker-ingress.
147                 access_ip_v4}"
148             bastion_host = "${openstack_networking_floatingip_v2.float.*.address[0]}"
149         }
150     }
151 }
```



```

147 }
148 inline = [
149     "echo 'Starting to deploy ingress-worker!'",
150     "sudo docker run -d --name rancher-agent --privileged --restart=unless-
        stopped --net=host -v /etc/kubernetes:/etc/kubernetes -v /var/run:/
        var/run rancher/rancher-agent:v2.0.8 --server https://${
        openstack_networking_floatingip_v2.float.*.address[0]}:8443 --token
        ${var.rancher_token} --ca-checksum ${var.rancher_ca_checksum} --
        etcd --controlplane --worker --internal-address ${
        openstack_compute_instance_v2.rancher-worker-ingress.access_ip_v4}
        --address ${openstack_networking_floatingip_v2.float-worker.address
        } --label app=ingress"
151 ]
152 }
153 }
154
155 resource "openstack_compute_instance_v2" "rancher-worker" {
156     name          = "rancher-worker-${count.index}"
157     count         = "${var.worker_count}"
158     region       = "${var.region}"
159     image_id     = "${var.image_id}"
160     flavor_name  = "${var.worker_flavor}"
161     key_pair     = "${openstack_compute_keypair_v2.kube.name}"
162     security_groups = ["${openstack_compute_secgroup_v2.kube.name}"]
163     depends_on   = ["openstack_compute_instance_v2.kube-master"]
164
165     network {
166         uuid = "${openstack_networking_network_v2.kube.id}"
167     }
168 }
169
170 resource "null_resource" "worker" {
171     depends_on=["openstack_compute_instance_v2.rancher-worker"]
172     count      = "${var.worker_count}"
173     triggers {
174         cluster_instance = "${openstack_compute_instance_v2.rancher-worker.*.
            id[count.index]}"
175     }

```

## A Appendix

```
176 provisioner "remote-exec" {
177
178     connection {
179         type          = "ssh"
180         user          = "${var.ssh_user_name}"
181         private_key   = "${file("${var.ssh_key_file}")}"
182         host = "${openstack_compute_instance_v2.rancher-worker.*.access_ip_v4[
                count.index]}"
183         bastion_host = "${openstack_networking_floatingip_v2.float.*.address[0]}"
184     }
185     inline = [
186         "echo 'Starting to deploy'",
187         "sudo docker run -d --name rancher-agent --privileged --restart=unless-
                stopped --net=host -v /etc/kubernetes:/etc/kubernetes -v /var/run:/
                var/run rancher/rancher-agent:v2.0.8 --server https://${
                openstack_networking_floatingip_v2.float.*.address[0]}:8443 --token
                ${var.rancher_token} --ca-checksum ${var.rancher_ca_checksum} --
                etcd --controlplane --worker --internal-address ${
                openstack_compute_instance_v2.rancher-worker.*.access_ip_v4[count.
                index]}"
188     ]
189 }
190 }
```

Listing A.3: A Kubernetes deployment description for the measurement-access-service

```
1 apiVersion: apps/v1beta2
2 kind: Deployment
3 metadata:
4   labels:
5     workload.user.cattle.io/workloadselector: deployment-default-nynm-
      measurements-access
6   name: nynm-measurements-access
7   selfLink: /apis/apps/v1beta2/namespaces/default/deployments/nynm-
      measurements-access
8 spec:
9   progressDeadlineSeconds: 600
10  replicas: 1
11  revisionHistoryLimit: 10
12  selector:
13    matchLabels:
14      workload.user.cattle.io/workloadselector: deployment-default-nynm-
        measurements-access
15  strategy:
16    rollingUpdate:
17      maxSurge: 1
18      maxUnavailable: 0
19    type: RollingUpdate
20  template:
21    metadata:
22      creationTimestamp: null
23    spec:
24      affinity: {}
25      containers:
26      - env:
27        - name: MONGODB_HOST
28          value: mongodb
29        - name: MONGODB_PORT
30          value: "27017"
31        image: registry.gitlab.com/noise-map-system/nynm-measurements/
          accessservice
32        imagePullPolicy: Always
33        name: nynm-measurements-access
34        resources: {}
35        securityContext:
36          allowPrivilegeEscalation: false
37          privileged: false
38          readOnlyRootFilesystem: false
39          runAsNonRoot: false
40        stdin: true
41        terminationMessagePath: /dev/termination-log
42        terminationMessagePolicy: File
43        tty: true
44        dnsPolicy: ClusterFirst
45        imagePullSecrets:
46        - name: gitlab
47        restartPolicy: Always
48        schedulerName: default-scheduler
49        securityContext: {}
50        terminationGracePeriodSeconds: 30
```

Listing A.4: Example pipeline specification for the measurement accessservice

```

1 stages:
2   - build
3   - bake
4   - deploy
5 build:
6   image: maven:3-jdk-8
7   stage: build
8   script:
9     - mvn clean install
10  artifacts: #which are passed to the next stage of the pipeline
11    paths:
12      - ./target
13 bake-master:
14   image: docker:latest
15   services:
16     - docker:dind
17   stage: bake
18   dependencies:
19     - build
20   script:
21     - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
22       $CI_REGISTRY
23     - docker build --pull -t "$CI_REGISTRY_IMAGE" .
24     - docker push "$CI_REGISTRY_IMAGE"
25   only:
26     - master
27 bake-tag:
28   image: docker:latest
29   services:
30     - docker:dind
31   stage: bake
32   dependencies:
33     - build
34   script:
35     - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
36       $CI_REGISTRY
37     - docker build --pull -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" .
38     - docker tag "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" "$CI_REGISTRY_IMAGE
39       :$CI_COMMIT_TAG"
40     - docker push "$CI_REGISTRY_IMAGE:$CI_COMMIT_TAG"
41   only:
42     - tags
43 deploy:
44   stage: deploy
45   image: roffe/kubectl
46   only:
47     - tags
48   when: manual
49   script:
50     - echo $KUBE_CONFIG_INTEGRATION | base64 -d > kubeconfig
51     - export SERVICE=nym-measurements-access
52     - kubectl --kubeconfig kubeconfig set image deployments $SERVICE $SERVICE
53       =$CI_REGISTRY_IMAGE:$CI_COMMIT_TAG
54     - kubectl --kubeconfig kubeconfig rollout status deployments $SERVICE

```

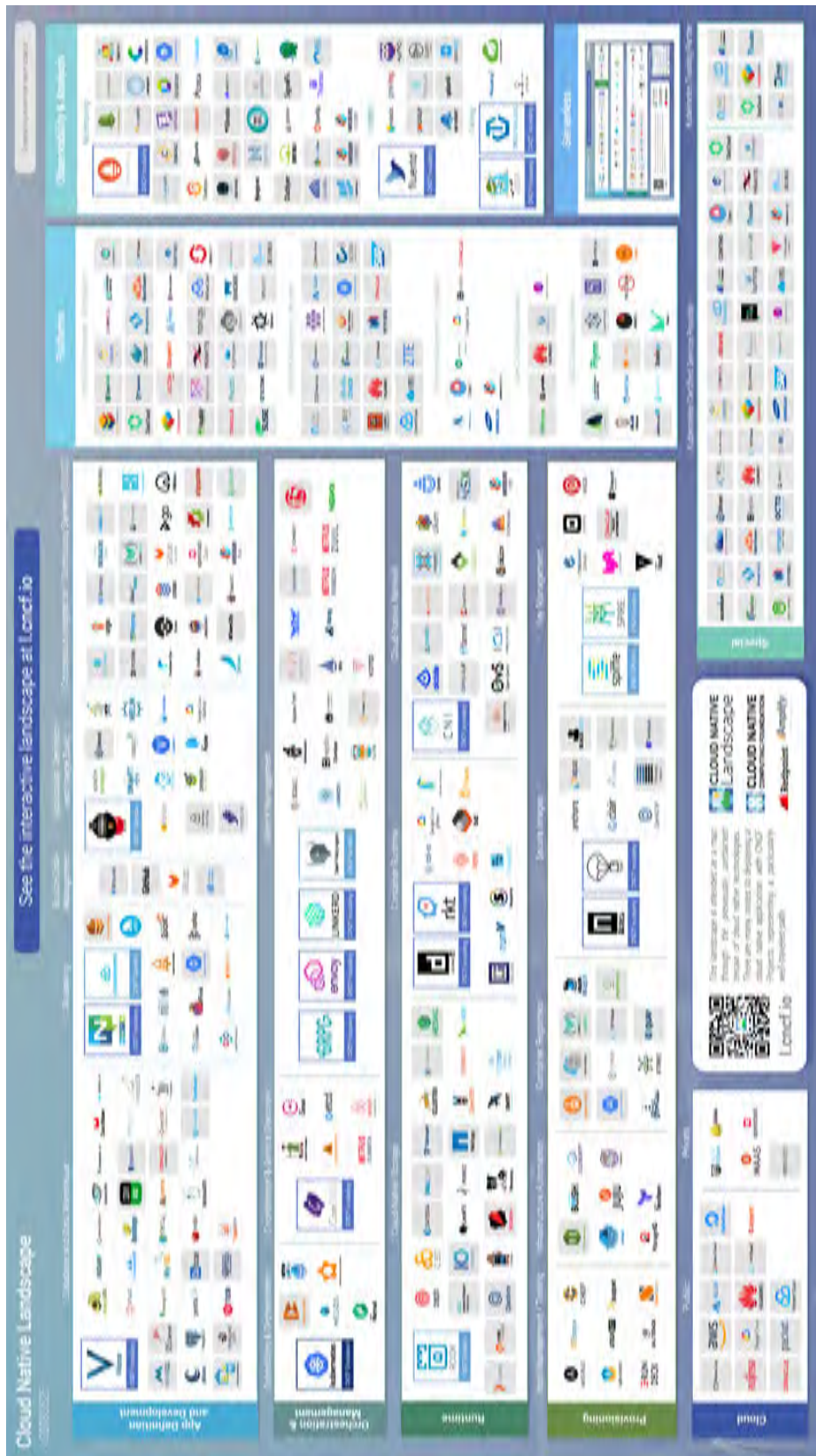





Figure A. 1: Static Cloud Native Landscape by [5]



# List of Figures

2.1	MCS-Spectrum for Crowdsensing and personal-sensing represented as a 2D-spectrum. . . . .	8
2.2	Typical self-made sensor for measuring mass concentration in the air. Image by [41]  . . . . .	11
2.3	Example for different types of geographic relations. . . . .	14
2.4	Example DGGs based on partitioning of an icosahedron. Image by [73]  . . . . .	18
3.1	The functionality of a typical professional Sound Level Meter (SLM). Image by [3]  . . . . .	33
3.2	Example mobile application for measuring and viewing noise data by [42].	36
4.1	Graphical representation of the overall architecture. This work provides detail about orange components, the blue components are only described briefly for completeness. . . . .	53
4.2	Internal architecture of Kubernetes. Based on a drawing from the official documentation [44]. . . . .	56
4.3	Example flow of information through Kubernetes for creating a pod. Based on a diagram in [9]. . . . .	58
4.4	Structure of the distributed commit log of an “Example Topic” and the corresponding Producer and Consumer dependencies. Image is based on multiple drawings in [56]. . . . .	64
4.5	Conceptual internal architecture of the Measurement Context showing different phases for data processing and access. . . . .	70
4.6	Plots of two regular Points and their differently sized H3 hexagons. . . . .	75
4.7	Plot of a H3 index, splitting the city of Philadelphia in the middle. . . . .	76
4.8	Authentication and Authorization in a monolithic vs distributed environment.	80
4.9	Example of a typical information flow using distributed authentication. . .	83
4.10	Overall architectural design of the Communication Context. . . . .	90

*List of Figures*

5.1	Diagram showing the integration infrastructure deployed by Terraform. . . . .	93
5.2	Diagram showing the data-model of the Measurement Bounded Context. . . . .	102
5.3	Concrete internal implementation of the Measurement Context shown all major steps of the data-flow. . . . .	104
6.1	Prototypic web frontend showing a visualization of single test measurements. . . . .	116
6.2	Prototypic web frontend showing a visualization of aggregations in H3 resolution 8 calculated by the stream processing pipeline. . . . .	116
6.3	Scalable MongoDB cluster architecture for accessing data shards. Based on [53]. . . . .	119
A.1	Static Cloud Native Landscape by [5] . . . . .	157



# List of Tables

2.1	Precision values for numbers of decimal digits at latitude and longitude values for coordinates. . . . .	16
3.1	Functional Requirements for the Noisemap-Project. . . . .	40
3.2	The qualities types of Non-Functional Requirements after [79]. . . . .	42
3.3	Internal Non-Functional Requirements for the Noisemap-Project. . . . .	43
3.4	External Non-Functional Requirements for the Noisemap-Project. . . . .	44
3.5	Four environments related to indoor-outdoor contexts based on [78] . . .	46
4.1	Key business capabilities of the backend mapped to bounded contexts . .	51
4.2	Geometric Types supported by GeoJSON. . . . .	71
5.1	Current topics used in the noise stream-processing . . . . .	106
6.1	Discussion of the design in relation to functional requirements of the Noisemap-Project. . . . .	122
6.2	Discussion of the design in relation to non-functional requirements of the Noisemap-Project. . . . .	125

Name: Ferdinand Birk

Matriculation number: 890638

**Honesty disclaimer**

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, .....

Ferdinand Birk