Design, Implementation and Evaluation of a System to Create a Data Set Supporting Research in the EnergieBroker Platform

Master-Thesis

zur Erlangung des akademischen Grades

Master of Science – M. Sc.

Hochschule RheinMain

Fachbereich Design Informatik Medien

Master-Studiengang Informatik

vorgelegt von

Niklas Sauer

am

18. Januar 2022

Bearbeitungszeitraum	12.07.2021 – 18.01.2022
Matrikelnummer	1209772
Referent	Prof. Dr. Heinz Werntges
Korreferent	Prof. Dr. Robert Kaiser
Betreuer	Johannes Kaeppel

Erklärung gem. BBPO 4.1.5.4 (3)

Ich versichere, dass ich die Master-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

....

Ort, Datum Niklas Sauer

Hiermit erkläre ich mein Einverständnis mit den im Folgenden aufgeführten Verbreitungsformen dieser Master-Arbeit:

Verbreitungsform	**ja**	**nein**
Einstellung der Arbeit in die Hochschulbibliothek mit Datenträger		✗
Einstellung der Arbeit in die Hochschulbibliothek ohne Datenträger		✗
Veröffentlichung des Titels der Arbeit im Internet		✗
Veröffentlichung der Arbeit im Internet		✗

....

Ort, Datum Niklas Sauer

Abstract

Contents

Acronyms

ACID	Atomicity Consistency Isolation Durability
API	Application Programming Interface
ARM	Advanced RISC Machine
CD	Continuous Deployment
CI	Continuous Integration
CPU	Central Processing Unit
DIN	German Institute for Standardization (German: *Deutsche Institut für Normung*)
EBP	EnergieBroker Platform
EEG	Renewable Energy Sources Act (German: *Erneuerbare-Energien-Gesetz*)
FIT	Feed-In Tariff
GDPR	General Data Protection Regulation
GPS	Global Positioning System
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
IEC	International Electrotechnical Commission
IP	Internet Protocol
ISP	Internet Service Provider
JSON	JavaScript Object Notation
JWT	JSON Web Token
LAN	Local Area Network
MAC	Mandatory Access Control
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge
MaStR	Marktstammdatenregister
mTLS	Mutual TLS Authentication
NAT	Network Address Translation
NUTS	Nomenclature of Territorial Units for Statistics
OBIS	Object Identification System
OCI	Open Container Initiative
OS	Operating System
PaaS	Platform as a Service
PIN	Personal Identification Number
PVS	Photovoltaic System
REST	Representational State Transfer

ROI	Return on Investment
SML	Smart Message Language
SOA	Service-Oriented Architecture
SSH	Secure Shell Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
VCS	Version Control System
VDE	Association for Electrical, Electronic & Information Technologies (German: *Verband der Elektrotechnik Elektronik Informationstechnik*)
VM	Virtual Machine

List of Figures

List of Tables

1. Introduction

1.1. Motivation

Germany's Renewable Energy Sources Act (German: *Erneuerbare-Energien-Gesetz*) (EEG) is a powerful tool used to accelerate the energy transition and thus, represents a great part of the contributions Germany can make to slow down global climate change. Disappointingly though, one of the major initiatives set forth to achieve these goals is likely to become an obstacle in the future – namely, the 20-year limitation of the EEG's guaranteed Feed-In Tariff (FIT) for small-scale Photovoltaic Systems (PVSes) and wind turbines, beginning from the moment of their commissioning. As a result, operators of systems which have become ineligible for governmental aid will have to decide whether to:

(A) continue operations at a throttled rate, hereby withholding renewable capacities

(B) trade excess amounts of energy at public exchanges, resulting in a lower ROI [1]

(C) replace their systems to secure a new guaranteed FIT, leading to more e-waste

Neither of these options is satisfying or helpful to mitigate climate change. For exactly this reason, the EnergieBroker Platform (EBP) was called to life. It attempts to establish private, autonomous marketplaces for renewable energy at which even small amounts can be traded cheaply and hopefully, more profitably compared to regular energy exchanges [2].

To support the ongoing research in this undertaking, a data set shall be created which details to what extent private households equipped with PVSes:

- feed energy into the grid
- obtain energy from the grid
- store energy in a home battery (if present)

This thesis attempts to provide the means for establishing such a data set across a large number of households as part of an upcoming data collection survey.

[1] A lower ROI may be expected because the volumes in excess amounts of energy generated by small-scale installations will have to rival that of energy corporations who already benefit from the economies of scale.

[2] Please refer to [67] for an in-depth explanation of the EBP. An interactive demo, co-created by the author of this thesis, is available at https://energiebroker.cs.hs-rm.de

1.2. Goals and Scope

This thesis is concerned with the design, implementation and testing of a system that can be used to collect and store the data presented above. It does not cover the process of selecting households, nor does it attempt to estimate the costs for operating such a system or draw any conclusions from the collected data itself. More importantly, the legal framework underpinning the survey is expected to have been clarified in advance. In short, the goals and scope of this thesis are the:

- requirements engineering
- component design and mapping of requirements thereto
- deployment and maintenance plan design
- production-grade implementation
- field test

1.3. Thesis Overview

2. Theoretical Framework

2.1. Microservice Architecture

Popularized by companies such as Amazon, Netflix, Uber, LinkedIn and SoundCloud, the microservice architecture has emerged as a pattern to avoid the problems of conventional monolithic designs [66, p. 847] [71, p. 584]. This section provides an overview to the problems faced in monolithic applications, distinguishes the microservice architecture from a traditional Service-Oriented Architecture and lays out its core principles, while noting some of its newly introduced challenges.

2.1.1. The Monolith Problem

A monolith is a software application whose modules cannot be executed independently [25, p. 1]. Hence, a monolith is characterized by requiring to be deployed as a united solution [56, p. 24]. Based on this definition, a set of obstructive characteristics inherent to monolithic applications can be derived:

Maintainability

When developing an application with a single large codebase, it naturally becomes harder to maintain and comprehend [25, p. 2]. The latter is especially true for beginners, slowing down their productivity [56, p. 24]. Further, refactoring changes may touch many parts of the software which might lead to a situation in which refactoring is ignored because it becomes too risky [38, p. 35].

Dependencies and technology lock-in

Monoliths typically suffer from the "dependency hell" problem, where an application's modules depend on conflicting versions of a shared library [25, p. 2]. In cases where this has been solved, the likelihood of having to make many changes to update to a newer version of a library again increases with a growing codebase. Next to that, it becomes very difficult to change the technology stack, leading to a lock-in and forcing developers to use the same language in every problem domain [56, p. 24] [25, p. 2].

Deployment

Rolling out a new application version requires the complete set of services to be restarted, regardless of whether a service has been altered or not [25, p. 2]. Similarly, failure of one service leads to downtime across all services [1, p. 970]. The deployment can therefore be viewed as a single point of failure [71, p. 584]. Moreover, deployments are likely sub-optimal due to conflicting resource requirements (e.g. CPU vs. memory-intensive). Developers often have to compromise with a one-size fits all configuration [25, p. 2].

Scaling

By combining multiple services into a single process, scaling can lead to resource wastage since the whole application needs to be scaled up even if an increase in traffic stresses only a subset of modules [66, p. 850] [25, p. 2]. Less popular services consume unnecessary (idle) amounts of resources [71, p. 584]. Setting appropriate scaling thresholds also becomes more challenging because different components may have different resource requirements [56, p. 24].

Kalske, Mäkitalo, and Mikkonen, however, acknowledge that the monolith approach might be the correct choice if the codebase is relatively small or the need for fine-grained scaling has not come up [38, pp. 34, 36]. Villamizar et al. add that monoliths are faster to set up [71, p. 589]. Empirically, most organizations start with something big and slowly transition to a decomposed architecture when scaling problems arise [69, p. 113] [71, p. 590]. Yet, it can be argued that an organization should spend more time on the design upfront since it is easy to introduce tight coupling, hereby hindering future refactoring endeavors [38, p. 34].

2.1.2. Definition

A microservice-based application is one in which the core functionality has been decomposed into many small units that can be independently developed and deployed [40, p. 43] [1, p. 970]. Each unit, a *microservice*, is modeled around a single, clearly defined set of closely related functionalities that can be used independently over the network through a well-defined interface [3] [68, p. 56] [2, p. 176] [17, p. 30]. This definition implies that microservices:

- use independent codebases, thus can build on different technology stacks
- run in distinct processes, thus can fail and scale independently
- are decoupled but can be used as building blocks to form larger services

[3] Cerny, Donahoo, and Trnka draw a comparison to three Unix ideas: a program should fulfill only one task well, be able to work with other programs and use a universal interface [17, p. 31].

2.1.3. Decomposition Techniques

The task of splitting an application into services should happen along the lines of related processes that can be carried out in isolation and should not just arbitrarily distribute features across services [68, p. 61]. As in traditional software engineering, the term "cohesiveness" is used to indicate that a service only implements functionalities which are strongly related to the concern that the service is meant to model [25, p. 2]. Various techniques are cited to determine the breadth of concern:

Single responsibility principle
Defines a responsibility of a class as a reason to change and states that a class should only have one such reason [51, p. 36] [69, p. 116]. Is analogously applied to microservices. Leads to a large amount of services.

Y-axis of scale cube
Splits an application into distinct sets of related functions. Each set is implemented by a microservice. In a verb-based approach, sets consist of a single function which covers a specific use case, whereas the noun-based approach creates sets of functions responsible for all operations related to a particular entity [51, p. 36]. Leads to large and medium amount of services, respectively.

Domain-driven design
Refers to the application's problem space, i.e. the business, as the domain [29]. This domain consists of multiple subdomains (e.g. product catalog, order management). Each subdomain is represented by a microservice [8, p. 3]. Leads to a small amount of services.

Irrespective of the technique chosen, the overall goal should be to minimize later interface changes, i.e. to establish proper service contracts [56, p. 26].

2.1.4. Service Registry Pattern

The law of conservation of complexity states that the complexity of a large system does not vanish when the system is broken up into smaller pieces. Instead, the complexity is pushed to the interactions between these pieces [51, p. 38] [69, p. 114]. Applied to microservice-based applications, this means that developers need to deal with the challenges innate to distributed systems [56, p. 24] [71, p. 589]. One such challenge is the fact that services can no longer be invoked through language level method calls but rather only through the network. Moreover, given the need for scaling, clients are now required to make requests to a dynamically changing set of service instances. And since it is unfeasible to run these instances at fixed locations [4], a pattern known as the service registry is commonly employed [51, p. 37].

A service registry acts as a database of services, storing the various instances along their locations, i.e. the IP address and port number. Instances are added on startup and removed on shutdown. Keeping this in mind, two types of service discovery mechanisms are distinguished [40, p. 46]:

Client-side

To contact a service, clients obtain the locations of all service instances by querying the registry. The client then needs to perform a load balancing algorithm to decide which instance will be contacted.

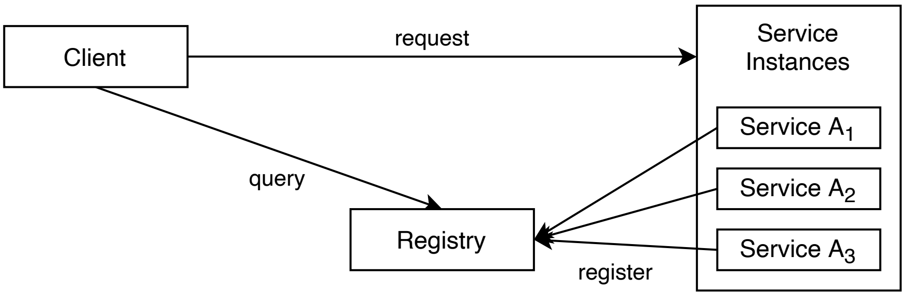

Figure 2.1.: Client-side service discovery [51, p. 37]

Server-side

To contact a service, clients make a request to the service's load balancer which runs at a well-known location. This load balancer queries the registry and forwards the request to an available instance.

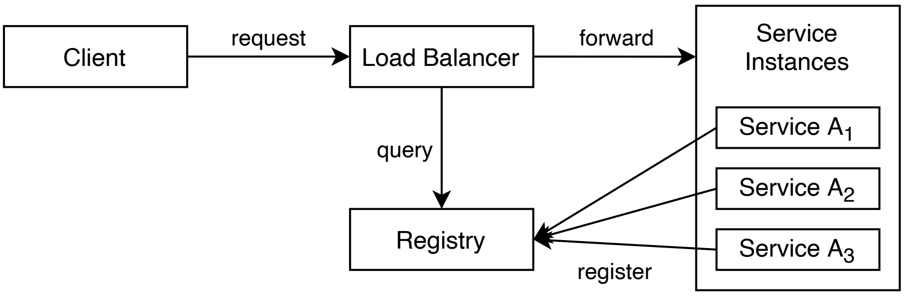

Figure 2.2.: Server-side service discovery [51, p. 37]

In any case, the service registry is a critical component and thus, must be highly available.

[4] In the event of a network partition, a standard recovery process would attempt to restart a service in the healthy partition, thus leading to a new network location for this service.

2.1.5. API Gateway Pattern

Depending on the decomposition technique chosen (see subsection 2.1.3), microservices might provide very fine-grained APIs. In turn, this means that clients may need to interact with multiple different services in order to carry out a high-level business process. To hide this complexity from clients and ensure consistent behavior, a pattern known as the API gateway is employed.

An API gateway represents the single entrypoint for all clients in which some requests are simply proxied while others fan out to and consume multiple services. In the latter case, gateways can be viewed as orchestrators and as such, typically do not have persistence layers [71, p. 585]. They may, however, cache responses. Another important task in orchestrating multiple microservices is managing distributed transactions [5], i.e. ensuring atomicity guarantees for a set of distributed resources [17, p. 32]. Finally, gateways may also deal with generic features such as authentication and authorization or implement the circuit breaker pattern to prevent a service failure from cascading to other services [38, p. 41] [51, p. 37].

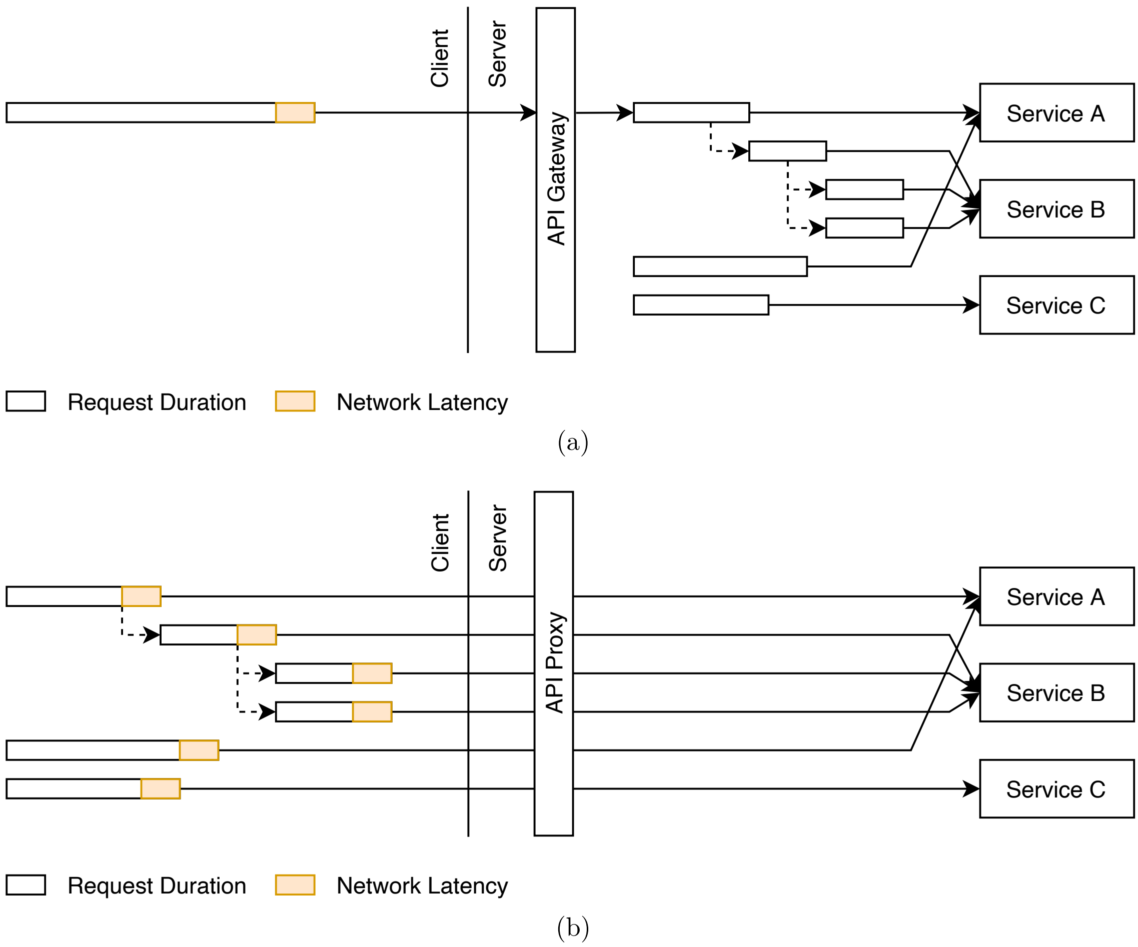

Figure 2.3.: Service consumption model with (a) and without (b) an API gateway [19]

[5] Distributed transactions are commonly implemented using the two-phase commit protocol. The no-ACID transaction type has also been proposed for this context, which is known as a compensation transaction [17, p. 32].

It shall be noted that gateways incur a performance penalty because they introduce an additional network hop [51, p. 37]. This is generally true for proxying gateways. Orchestrating gateways, on the other hand, have the potential to decrease latency since the various requests being collapsed and moved to the server-side will now originate from the target network (comp. Figure 2.3) [19]. In both cases, the performance degradation will heavily depend on the system's interconnectedness [25, p. 9].

2.1.6. Database-per-Service Pattern

To keep microservices loosely coupled, a pattern known as database-per-service is employed. This pattern calls for each microservice to have its own database, compared to sharing one across multiple services. Sharing is achieved by making the data accessible via the service's API [51, p. 36] [68, p. 59]. Messina et al. discern between three levels of pattern conformity [51, p. 37]:

Private tables
Each service has a set of tables private to that service.

Private schema
Each service has a database schema private to that service.

Private database
Each service has its own database.

While this pattern contributes to service intimacy, it comes at the cost of having to redefine data models and restate business rules across services [6] [17, p. 30].

2.1.7. Delineation from Service-Oriented Architecture

Historically, the complexity of monolithic applications (see subsection 2.1.1) has already been addressed using different Service-Oriented Architecture (SOA) approaches that also decompose a large system into many smaller services. Academia, however, is undecided as to whether microservices should be considered as a subset or superset of SOAs or whether they constitute a new, distinct idea [71, pp. 584–585] [17, p. 30]. The systematic mapping study conducted by Cerny, Donahoo, and Trnka in [17] spends a great deal on contrasting the two architectures. A short summary is given in the following paragraphs.

[6] In domain-driven design, the concept of a Bounded Context describes that services operate with business objects in a specific context and therefore, only need to model a subset of the global object's attributes [17, p. 30].

In both approaches, services cooperate to provide functionality for the overall system. However, the path to achieving this goal differs. This is most obvious when looking at the interaction patterns between the services involved. SOAs rely on orchestration, whereas microservice-based applications prefer choreography. The former expects a centralized business process to coordinate activities across services and combine the outcomes, whereas the latter expects individual services to collaborate based on their interface contracts. Orchestration differs from choreography with respect to where the logic that controls the interactions should reside. The two terms describe a centralized and decentralized approach hereof, respectively (comp. Figure 2.4).

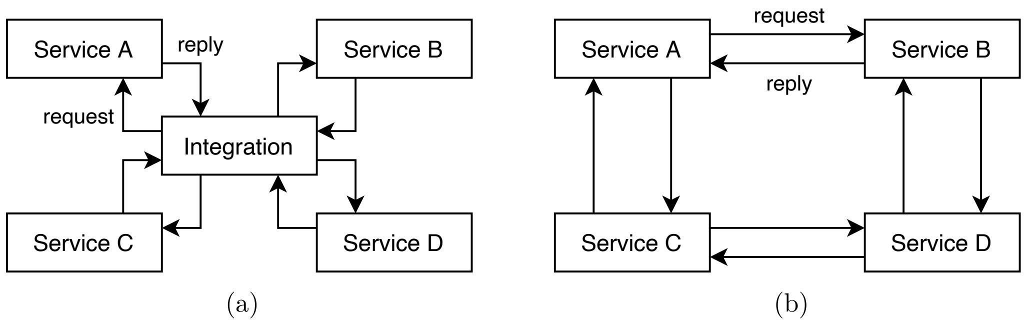

Figure 2.4.: Service orchestration (a) and choreography (b) [17, p. 30]

An important remark that Cerny, Donahoo, and Trnka make is that orchestration through an integration layer, such as a messaging bus, oftentimes leads to a situation in which the system parties, i.e. the services, agree on a standardized representation of the business objects they exchange. The system ends up with one kind of business object each. This is known as a canonical data model. The danger of it lies therein that a change in one of the business objects necessitates changes in all of the services that deal with this object. As a result, deployments in a SOA again happen in a monolithic fashion. In the microservice architecture, such a change can at most propagate to the API gateways (comp. subsection 2.1.5), though this is less likely because gateways integrate services based on their interfaces, not on their models.

Lastly, the most obvious and biggest drawback of SOAs and their centralized orchestration model must be stated. Using a centralized business process to integrate all services once again entails having a single point of failure.

2.2. OS-level Virtualization

Virtualization describes the act of creating a virtual version of something [16, p. 2]. In the context of computing, it specifically refers to hardware resources such as CPU, memory, storage or network devices which can be used to create complete virtual instances of computer systems [65, p. 21]. This section sheds light on the motivation behind virtualization, gives a brief overview of the traditional approach hereto and focuses on a more recent and lightweight alternative in the remainder.

2.2.1. The Need for Virtualization

Server consolidation attempts to maximize resource utilization, while also reducing costs through energy savings [7] [73, p. 233] [27, p. 2]. This is achieved by using fewer physical servers to host the same number of applications. However, without an isolation layer there are no guarantees that an application from one user will not interfere with that of another [73, p. 233]. Isolation and multi-tenancy are exactly those traits promised by virtualization technologies [65, p. 21]. Coupled with a software layer to provision resources on demand, virtualization yields the elastic multi-tenant model embodied in cloud computing [39, p. 203] [10, p. 81] [57, p. 24].

2.2.2. Comparison to Hardware Virtualization

In traditional hardware virtualization, a so-called hypervisor makes siloed slices of hardware available in the form of Virtual Machines (VMs) by emulating the underlying physical resources. Each of the VMs (*guests*) running on the physical hardware (*host*) comes with its own full-fledged OS. Two types of hypervisors can be discerned [50, p. 2] [27, p. 1] [55, pp. 386–387]:

Type 1 (bare-metal)

Operates directly on top of the host's hardware and requires no host OS.

Type 2 (hosted)

Operates as a software layer on top of the host's OS.

Although virtualization through emulation enjoys great popularity [8], it comes at the cost of efficiency. On the other hand, the overhead introduced by OS-level virtualization can be considered almost negligible [55, pp. 386, 392].

[7] In 2018, about 1% of the world's electric energy consumption stemmed from data center operations [4] [3]. If resources can be used more efficiently, carbon emissions could be lowered.

[8] Hardware acceleration for hypervisor-based virtualization has even been incorporated into commodity processors [73, p. 233].

Whereas hypervisor-based virtualization provides strong isolation guarantees between systems, OS-level virtualization only strives to isolate processes. Such an isolated process is known as a container [9]. Here, the isolation mechanisms are provided as kernel features which establish an abstract and protected view on the OS, making two containers unaware of each other or any of the other processes which run on the host [50, p. 2] [27, pp. 1–2].

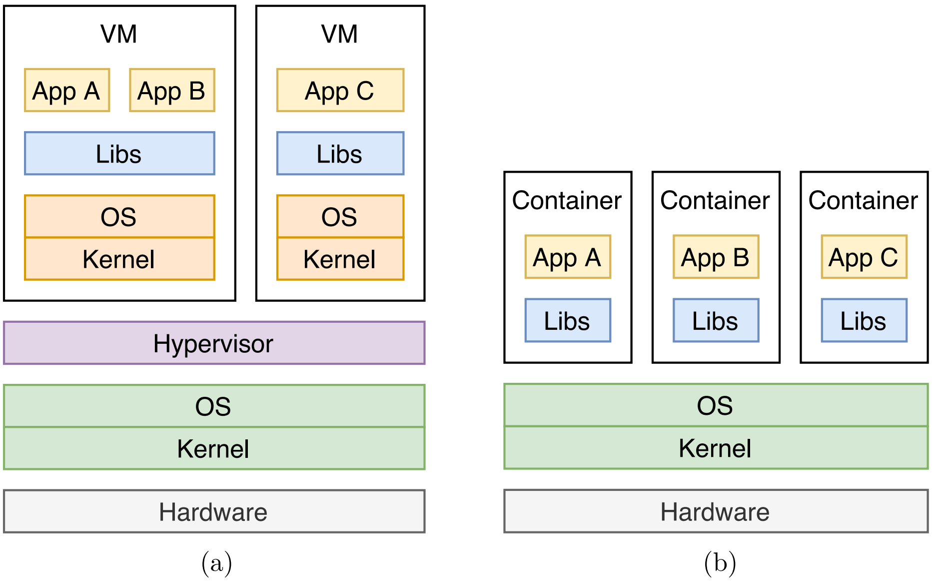

Figure 2.5.: Hardware (a) and OS-level (b) virtualization [55, p. 387]

Even though both technologies enable a safe multi-tenant model of hardware by confining parts of the application infrastructure, Pahl ascribes them to different use cases, arguing that VMs are concerned with hardware allocation and management, while containers are tools for delivering software. He further compares them to the concepts of IaaS and PaaS, respectively [57, p. 24]. Eder and Merkel see the potential that the two technologies would have in complementing each other since the resource footprint of containers is minimal and its security profile is still slightly worse than that of VMs [10] [27, p. 6] [50, p. 2].

Lastly, it shall be noted that the shared kernel approach of containers means that processes being isolated need to be compatible with the host's kernel and CPU architecture [55, p. 386] [27, p. 2]. In other words, it is not possible to, for example, run Windows containers on Linux hosts or deploy x86 containers on ARM because no emulation is taking place. On the flip side, the shared kernel approach allows kernel security patches to be applied without having to modify a container. This is not the case for a VM, where the underlying machine image would have to be

9 Containers are known as jails or zones in the FreeBSD and Solaris OSes, respectively [27, p. 2].

10 Kernels cannot prevent interference in low-level resources such as the CPU's L3 cache or memory bandwidth [13, p. 52]. A good example of combining both virtualization technologies is given by the Google Kubernetes Engine. Here, a cluster of hosts is created on the basis of VMs which then exclusively run software in the form of containers.

rebuilt first [27, p. 3]. Moreover, sharing a kernel allows container-based solutions to achieve a higher density of virtualized instances when compared to hypervisor-based solutions [55, p. 386] [39, p. 204]. Containers are also a magnitude faster to start and stop since they are essentially just processes that have to be spawned and terminated, whereas the OS of a VM needs to be fully booted and shut down [50, p. 2] [27, p. 2]. Similarly, being a process means that containers do not occupy any resources when they are not executing. VMs will idle until they are shut down [50, p. 2].

2.2.3. Linux Kernel Containment Features

As previously hinted at, containers rely on the host's kernel to sandbox processes from each other. In Linux, the set of containment features include but are not limited to [45]:

Chroots

`chroot()` (from *ch*ange *root*) changes the root directory of the calling process and all of its children [11]. This is used to restrict a container's view on the filesystem. However, it cannot be considered a security feature because there are multiple intentional escape hatches [27, p. 3].

Namespaces

Namespaces provide one or more processes with private and restricted views towards certain global system resources [55, p. 387]. Changes to resources within a namespace are only visible to processes that are members of that namespace. The namespace type (e.g. `ipc`, `net`, `pid` or `user`) indicates which kinds of resources are being isolated. As an example, this feature allows processes within a container to have identifiers that are already in use on the host system or within other containers [27, p. 3]. The network namespace can provide a container with its own network device and virtual IP address, whereas the user namespace would be used to ensure that a container's user database is separated from that of the host, which means that the container's root user privileges cannot be applied on the host system [50, p. 1].

Control groups

Control groups, usually referred to as cgroups, provide resource accounting and limiting for a set of processes [55, p. 387] [50, p. 1]. Even though they are not mandatory for process isolation, cgroups can ensure that one container cannot starve another (e.g. in a denial-of-service attack), and can equally help to realize cost-accounting multi-tenant environments [27, p. 4].

[11] Some people trace the inspiration for containers back to `chroot()`, which was originally introduced with Unix 7 in 1979 [10, p. 82].

Mandatory access control

Mandatory Access Control (MAC) describes a concept in which access to or operations on a particular resource is granted based on different authorization rules (*policies*). On Linux, the two prevalent implementations hereof are AppArmor and SELinux [12]. In the context of containers, MAC is useful in that it can restrict the process to its minimal requirements, thus further reducing the attack surface against the host and other containers [27, p. 4].

2.2.4. Docker

The concept of container-based virtualization has existed long before Docker came into view in 2013 [13] [14]. Yet, it was Docker, Inc. who made containers popular by creating a toolkit that is greater than the sum of its parts [50, p. 1]. The following descriptions present three areas of developer concern in which Docker shines and through which it has become a synonym for containers [15]:

Code packaging

A study found that less than 50% of software can be successfully built or installed. This is due to issues such as the "dependency hell" problem (see subsection 2.1.1), imprecise documentation or code rot. Executing code assumes the ability to create a compatible environment [11, p. 72]. Docker addresses this challenge by introducing the concept of container images. Such an image bundles an application with all of its dependencies up to, but excluding, the kernel [50, p. 1]. It is essentially the filesystem bundle made available to a process that is then isolated as a container (comp. Chroots on page 12). To further simplify things, Docker allows users to imperatively describe how such an image shall be built. This plain text file of steps to be taken is known as a Dockerfile and is ideally suited for use with a VCS, i.e. it can be checked in alongside a repository and can evolve with the application [11, p. 74]. As a final innovation in this area, Docker calculates the filesystem differences between each of the steps in a Dockerfile and treats each as a distinct layer of the image. This enables developers to both version and extend images [16] [50, p. 1].

[12] Security-Enhanced Linux (SELinux) was originally developed by the National Security Agency to address threats of tampering and enable the confinement of damage caused by malicious applications.

[13] `https://docker.com`

[14] In 1998, FreeBSD came up with an extended version of `chroot()`, called jails. This capability further improved with the release of Solaris' zones in 2004 [10, p. 82]. In Linux, kernel namespaces were discussed as early as 2006 [16, p. 1].

[15] Boettiger makes a good case of how Docker can also help to make research reproducible and more easily extendable [11, p. 71].

[16] At runtime, all image layers will be merged into a single representation of the filesystem. This is known as a union mount [57, p. 26].

Code portability

By packaging an application along with its dependencies into a single image, Docker paves the way for image-based deployments that offer the freedom of “develop once, deploy everywhere” [17] [39, p. 203]. In this context, a container can be regarded as a running instance of an image. However, it is unlikely that such a container by itself will run, if at all, across platforms in the same way due to differences in, for example, networking or storage [11, pp. 74–75]. This problem is known as runtime consistency [39, p. 203]. To enable consistent behavior, Docker includes a container runtime that abstracts many of the platform peculiarities [18] [11, p. 75].

Code reuse

As previously stated, Docker allows one image to extend another. To make this process more straightforward, Docker offers a public registry, the Docker Hub, to and from which users can up- and download versioned, binary copies of images. Eder notes that having such a social aspect in the area of virtualization was previously unheard of. At the same time, he warns that because images do not get updated automatically, there is a large amount of images containing security vulnerabilities. For instance, over 30% of official images, i.e. those created by official project developers, contain high priority vulnerabilities [27, pp. 6–7].

Driven by the fear of a lack of interoperability, Docker and other leaders in the container industry established the Open Container Initiative (OCI) in 2015 to standardize the aforementioned container image formats and runtimes [19] [65, p. 23].

2.3. Container Orchestration Platform

Given the operational benefits of microservice-based applications (see section 2.1), as well as containers used as a deployment target and medium (see section 2.2), it is not surprising that the industry is focused on simplifying the management of potentially hundreds of instances of containerized services across a cluster of hosts (*nodes*) [40, p. 44] [58, p. 1]. This section presents the immediate benefits of dealing with container workloads, lists the requirements of any such container orchestration platform and tries to convey its way of working by laying out a reference architecture.

[17] While image-based deployments can also be achieved with VMs, they are not as portable nor lightweight because they contain the complete toolchain for running an OS, including device drivers, the kernel and init system [39, p. 203] [27, p. 2].

[18] Of course, target platforms will still have to provide some sort of process isolation features (see subsection 2.2.3). Because this is not the case on macOS and Windows, Docker runs inside a Linux-based VM on those platforms [50, p. 5].

[19] `https://opencontainers.org`

2.3.1. The Shift to Container Workloads

In an article about the lessons Google has learned from creating and operating three container-management systems over more than a decade [20], Burns et al. make a strong case as to how containerization transforms data centers from being machine-oriented to being application-oriented [13, pp. 52–53]. Since a container is essentially just an isolated process, the identity of an instance being managed in a cluster now exactly lines up with the identity expected by developers, namely, the main process (and its children) of their applications. This shift of primary key has ripple effects throughout the cluster's infrastructure. For instance, load balancers no longer balance traffic across machines, but across instances of an application. Logs are automatically keyed by the application, whereas on machines logs are likely to be polluted by other applications or system operations. Finally, containers relieve developers and operations teams from worrying about machine-specific details. Together, this makes building, deploying, monitoring and debugging applications dramatically easier.

2.3.2. Basic Capabilities

A container orchestration platform can be broadly defined as a system that provides an enterprise-level framework for integrating and managing containers at scale [40, p. 44]. It is not only concerned with the runtime but also aids in the deploy and maintain phases of an application's lifecycle [15, p. 225]. For the purposes of automation, all cluster operations should be exposed and accessible via an API [15, p. 224] [57, p. 29]. Throughout literature, academia generally expects the following set of capabilities from a container orchestration platform:

Scheduling

Operators need to be able to deploy, i.e. schedule, containers onto hosts based on a variety of parameters. Typically, this encompasses a replication degree and placement constraints such as node affinity or resource requests [15, p. 225]. Deployments should also be able to take advantage of local inter-process communication by allowing containers to be co-located on the same node. Once scheduled, the platform should perform a readiness check to decide whether the container is capable of answering requests. Similarly, periodic health checks should be used to determine whether a container needs to be restarted or rescheduled onto a different host [37, p. 2].

[20] Google has contributed much of the Linux kernel's process isolation code [13, p. 50].

Scaling & availability

To cope with increases in traffic, horizontal scaling of applications is critical [37, p. 1]. In this context, an application refers to one or multiple containers that are managed as a single entity [40, p. 44] [58, p. 2]. Scaling may happen based on a static replication factor or through threshold-based autoscaling policies (e.g. CPU or memory utilization). Platforms should also allow operators to define more sophisticated policies via external plug-ins. Naturally, the load then needs to be distributed amongst container instances by means of a load balancer that utilizes the health check mentioned above to decide whether to include a container instance in its target set or not [15, pp. 225–226] [37, p. 2]. However, scaling by itself does not guarantee high availability or fault tolerance. This is especially true when dealing with a single node cluster. Instead, operators should follow the three principles of reliability engineering in which failures need to be detected, single points of failure eliminated and reliable crossovers established [40, p. 45]. Of this, redundancy, i.e. replication across physically separated hosts, remains the most important feature, though self-healing due to detecting failures early is just as key. A container platform must support and implement both techniques [2, p. 176] [13, p. 53].

Monitoring

Platforms should facilitate monitoring across two contexts. First, the cluster infrastructure needs to be observed for resource utilization and draining. Secondly, container activity must be made visible by aggregating logs and performance metrics, as well as allowing requests to be traced across containers [40, p. 47].

Next to these basic capabilities, platforms may further implement a slew of features across domains such as security, networking or maintenance. As an example, platforms could provide role-based access control for containers, allow configuring MAC (see subsection 2.2.3) on a per container basis, scan container images for vulnerabilities (see subsection 2.2.4), add support for the service registry pattern (see subsection 2.1.4) or enable cluster-wide backups.

2.3.3. Reference Architecture

In a survey on the state of the art in container orchestration technologies, Casalicchio describes how a container orchestrator may be implemented as an autonomic computing system based on the MAPE-K control loop [15, pp. 226-228]. Such systems manage themselves by dynamically adapting to changes in accordance to business policies and objectives. For compute clusters, this typically involves goals such as maximizing resource usage, reducing energy consumption or satisfying service-level agreements.

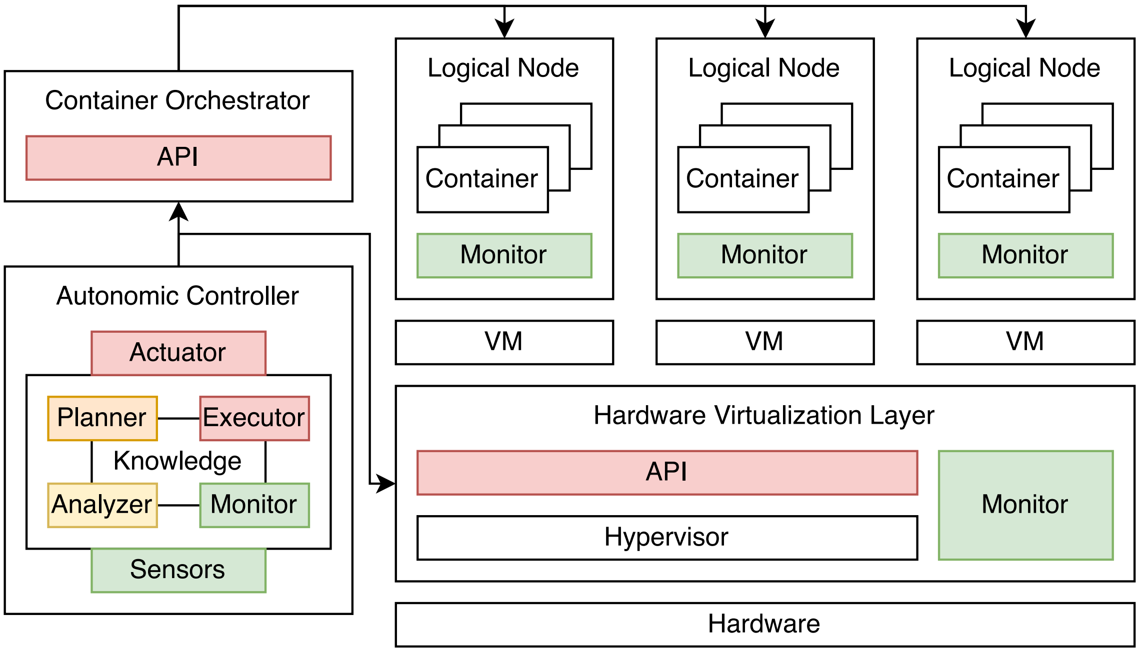

Figure 2.6.: Reference architecture for container orchestration platforms [15, p. 226]

Figure 2.6 illustrates the high-level architecture of container orchestration platforms that follow these self-adapting principles. The components in the MAPE-K cycle would collaborate as is described below:

Monitor

Collects details (e.g. node health or application load) from managed resources (e.g. VMs or containers) and performs preliminary aggregation, correlation and filtering to determine whether a symptom needs to be further analyzed.

Analyzer

Performs more complex data analysis and reasoning on reported symptoms, this time considering the shared knowledge base on the system's topology, policies, historical logs and metrics.

Planner

Devises a series of actions to be run against set of managed resources (e.g. spawn new container instances) in response to detected symptoms (e.g. high memory utilization).

Executor

Realizes adaptation plan using container and infrastructure management APIs.

3. Concept

3.1. Data Collection Survey

3.1.1. Overview

The research group around the EBP has called upon households to participate in an automated data collection survey that aims to establish a data set which details to what extent private households equipped with PVSes:

- feed energy into the grid
- obtain energy from the grid
- store energy in a home battery (if present)

Fundamentally, the task of creating such a data set boils down to:

(A) reading multiple electricity and generation meters [21]

(B) transmitting the observed data to a centralized data store

In order to do this in a fully-automated manner, the survey will require an end-to-end software-based solution. Having said that, such a solution will, however, still need to bridge between the physical world with its meters and the digital world where the observed data (*measurements*) shall be stored and processed. For this purpose, the solution may rely on an optocoupler (*measurement probe*) that is capable of converting the light signal (*meter interface*) of an electricity or generation meter into a serial bitstream [22]. Reversely, such an optocoupler may also be used to send light signals to the meter, allowing programmatic control of its menu.

With respect to Subtask (A), the survey intends to install one or more headless compute devices (*measurement devices*) in each participating household. These devices will be equipped with one or more measurement probes. The number of measurement devices and probes will vary depending on the number of meters to be read and whether they are physically separated from each other, i.e. whether

[21] This survey is particularly interested in meters which utilize the SML communication protocol to expose their internal measurement data via the optical interface standardized in IEC 62056-21, colloquially known as D0.

[22] Such a measurement probe is currently being developed and soldered by the supervisor of this thesis, Johannes Kaeppel.

one measurement device can be practically connected to multiple meters. Further, the survey is targeting a 15-minute interval in which measurements shall be made (*measurement cycle*).

Subtask (B) can be explained by the fact that the research group would like to analyze measurements as early as possible. More so, they see the potential to expand this survey with the ability to give households a preview of the transactions and revenue or savings that could be achieved if the household were to actively participate in the EBP (comp. [67, p. 63]). This rules out any solutions in which the measurements are stored on device and only processed when the survey ends, i.e. after the devices have been returned. Instead, measurements shall be transferred to a centralized data store on an hourly schedule, at minimum (*report cycle*).

3.1.2. Registration

Regardless of the criteria based on which households are selected (comp. section 1.2), participants will have to fill out a registration form that poses the following questions:

Contact details

- What is your name?
- What is your email address?

Location

- What is your address?

Metadata

- How many children live in your household?
- How many adults live in your household?
- How many of these adults have a job?
- Does your household have an electric vehicle charger?
- Does your household have a heat pump?
- Does your household have a home battery?

Photovoltaic systems

- What is the nominal power for each of your PVSes?
- What is the azimuth and tilt angle for each of your PVSes?
- What is the installation date for each of your PVSes?

Meters

- What is the model number of each of your electricity meters?
- What is the model number of each of your generation meters?
- Which of your PVSes is connected to which of your generation meters? [23]

The answers to this questionnaire (*registration details*) shall be stored alongside the measurements collected for that particular household. This allows the research group to, for example, correlate measurements with the geographic location of that household and thus, the weather conditions at the time of measurement.

3.1.3. Consent

Given the involvement of third parties, data capture, transmission and storage must be in compliance with local data protection laws (e.g. GDPR in Europe). Through consultation with a legal professional, the research group has determined that the solution shall track the participation status (*consent*) of each of the survey's subjects and couple that to the lifecycle of each measurement device, as well as to the data processing capabilities of the research group itself. Three states of participation shall be distinguished for any given household:

Consent not yet granted
The household has been signed up for participation in the data collection survey. It may have already received and connected one or multiple measurement devices. None of these devices shall make any measurements.

Consent granted
The household is actively participating in the data collection survey. During this period, the measurement devices shall make and transmit measurements. Collected measurements may be processed in a fully personalized manner, i.e. the research group may correlate measurements with the registration details.

[23] Technically, this question is not asked as part of the questionnaire but rather later when measurement devices are installed in that particular household.

Consent revoked

The household is no longer participating in the data collection survey. Measurement devices shall not make any further measurements and instead must be returned to the research group. Collected measurements may now only be processed in an anonymized manner (see subsection 3.1.4).

From these descriptions it is clear that the acts of granting and revoking a consent shall be considered one-time operations, i.e. participants shall not be able to reverse their decisions.

3.1.4. Anonymization

The GDPR stipulates that personally identifiable data must be removed upon user request [59, Art. 17, § 1]. Yet, to ensure that the collected measurements remain of value even after a participant has revoked his consent, the research group has come up with a concept to anonymize the registration details (comp. subsection 3.1.2), allowing continued processing and limited geographic correlation. This concept, again, has been established on the basis of acquired legal advice. It plans to:

(A) remove the contact details

(B) anonymize the location

(C) anonymize the Photovoltaic Systems

Obviously, Measure (A) is the most straightforward approach to obscuring a participant's identity. Unfortunately, the same technique cannot be applied to the location because measurements will almost, in all cases, have to be evaluated in the context of their geographic location (*geospatial analysis*). Therefore, Measure (B) is required. Similarly, Measure (C) is necessary because the specifications of a PVS could be matched against a public registry (e.g. MaStR in Germany [24]). The following descriptions shall further explain Measure (B) and Measure (C), respectively:

Anonymization of location

To protect a participant's identity but preserve some degree of locality among measurements, a household's location shall be anonymized by diluting it to a broader geographic region. This dilution shall happen in proportion to the population density per square kilometer (ρ_N) encountered for that particular household. To be specific, the household's location shall be replaced with a

[24] The Marktstammdatenregister (MaStR) contains the master data of all electricity and gas generating plants in Germany. It also tracks actors such as the plant operators or energy suppliers. Most of its records are viewable at `https://marktstammdatenregister.de`

randomly generated location that has a distance (d) of at least 0.3 km, and at most 20 km, from the original one. Given ρ_N, the exact bounds are defined as:

$$\begin{aligned} d_{\min} &= \max\left(\min(-1.321 \times \rho_N + 4264\,\mathrm{m}, 8000\,\mathrm{m}), 300\,\mathrm{m}\right) \\ d_{\max} &= \max\left(\min(-3.107 \times \rho_N + 10\,621\,\mathrm{m}, 20\,000\,\mathrm{m}), 1300\,\mathrm{m}\right) \end{aligned} \tag{3.1}$$

Anonymization of photovoltaic systems

To protect a participant's identity but preserve some context among measurements, each of a household's PVSes shall be anonymized by normalizing its specifications. To be specific, the nominal power of a PVS shall be rounded to the nearest integer and the installation date shall now only track the year. As for the azimuth and tilt angles, the research group has found that the German public registry only tracks these values vaguely, meaning that, in the context of this survey, no normalization is required thereof.

3.2. Functional Requirements

In accordance to section 3.1, the following functional requirements are defined as the bare minimum of features any implementation must provide in order to be used as the backbone of the data collection survey. These requirements are expressed from the perspective of an end-user who hopes to achieve different goals by using the system and are grouped by the various roles encountered. A visual representation in the form of a use case diagram is given in Figure B.1, located in the Appendix B on page 73.

Participant

FR-01 As a participant, I can grant my consent, so that the measurement devices installed in my household start collecting measurements.

FR-02 As a participant, I can revoke my consent, so that the measurement devices installed in my household stop collecting measurements.

FR-03 As a participant, I can revoke my consent, so that my registration details are anonymized.

FR-04 As a participant, I can view my registration details, so that I am able to notify the researchers of mistakes.

FR-05 As a participant, I can view my measurements, so that I am able to understand what kind of data is being captured, stored and processed.

Researcher

FR-06 As a researcher, I can add a new participant based on his registration details, so that measurements may be collected from a new household.

FR-07 As a researcher, I can modify the registration details of a participant, so that I am able to correct reported mistakes.

FR-08 As a researcher, I can view the consent of each participant, so that I know how many households are actively participating in the survey.

Administrator

FR-09 As an administrator, I can view the health status of each measurement device, so that I know when to investigate potential errors.

FR-10 As an administrator, I can remotely update measurement devices, so that new features may be added and bugs resolved without requiring user intervention.

FR-11 As an administrator, I can connect to individual measurement devices, so that I am able to service them beyond updates and debug errors at source.

3.3. Non-Functional Requirements

Whereas functional quality stresses conformance with the design specifications, structural quality addresses non-functional requirements such as security and maintainability [48, p. 2]. Together, both can be used to constitute the evaluation framework for measuring a system's quality, better known as Quality of Service (QoS). Liu, Ngu, and Zeng argue that it is not practical to come up with a standard model of attributes because QoS is a broad, context-dependent concept [46, p. 67]. Therefore, the following list of desired structural properties is, for the most part, based on the goal of supporting the long-term development and operation of this system.

Security

NFR-01 The system shall be protected from unauthorized access. Systems not affiliated with this project shall not be allowed to retrieve or modify any information. More specifically, the individual features outlined above shall only be accessible to their designated user.

Scalability

NFR-02 The system shall be able to handle varying levels of load without notable performance degradation. It shall be designed to scale both vertically and horizontally without having to exchange major components beforehand.

Availability

NFR-03 The system shall be highly available, so that measurements can be made, transmitted and stored around the clock.

Maintainability

NFR-04 The system shall be well-documented, include usage examples and other explanatory materials that help researchers, administrators and developers get onboarded quickly.

NFR-05 The system shall use tools such as a static type checker, linter and formatter to induce best practices, encourage self-descriptiveness and enforce a common legible style across all code contributed.

Testability

NFR-06 The system shall employ a rigid test suite to ensure that as few regressions as possible are introduced accidentally. Coverage reports shall highlight the places lacking in tests, so that these can be addressed in the future.

Portability

NFR-07 The system shall be compatible with a broad range of execution environments, including cloud-agnostic deployments, and shall only limit the choice in hosts if specialized hard- or software is required. Further, the system shall be self-contained and where necessary, explicitly state host-dependent features.

4. Design & Implementation

4.1. Modeling as an Edge Cloud Computing Problem

Driven by applications with strict time-bound requirements such as the Internet of Things (IoT), artificial intelligence or stream data analytics, a new computing model known as edge computing has emerged in recent years that pushes computations closer to the devices (*edge*) where data is being generated [74, p. 373] [5, p. 118]. This model does not only have the potential to improve application latency and thus, user-experience, but may also strengthen security and minimize bandwidth consumption by not having to transfer all data to centralized infrastructure (*cloud*) [25] [34, p. 295]. Edge computing works complementary to the cloud, placing services at the most efficient and logical place between the producers and consumers of data [5, p. 122].

Applied to the problem at hand, one can easily see how a system supporting the data collection survey outlined in section 3.1 resembles that of a large distributed IoT application which leverages edge computing. Multiple geographically dispersed measurement devices gather meter data, analyze it for relevance (*edge computing*), and then transfer the interesting measurements to a centralized data store for further in-depth processing (*cloud computing* [26]). Therefore, the following sections will discuss the proposed solution's architecture in the context of edge and cloud realms, respectively.

4.2. Component Design

Striving for a component-based design represents one of the most important practices in software engineering because it facilitates developers in maintaining a complex system by decomposing it into parts that are easier to conceive, understand and program (comp. The Monolith Problem on page 3). At the same time, the process of dismantling a system should not happen arbitrarily but rather attempt to take the application's domain structure into account to reduce the likelihood of having to remedy large parts afterwards (comp. Decomposition Techniques on page 5).

[25] This centralized infrastructure may either follow an on-premise, off-premise or hybrid approach.

[26] In this context, cloud computing does not necessarily imply the elasticity and self-service characteristics typically referred to (comp. cloud computing in glossary).

Accordingly, Figure 4.1 presents a microservice-based approach (see subsection 2.1.2) to modeling the system in question. It also gives a first indication as to how the components, i.e. microservices, will interact to achieve the desired behavior.

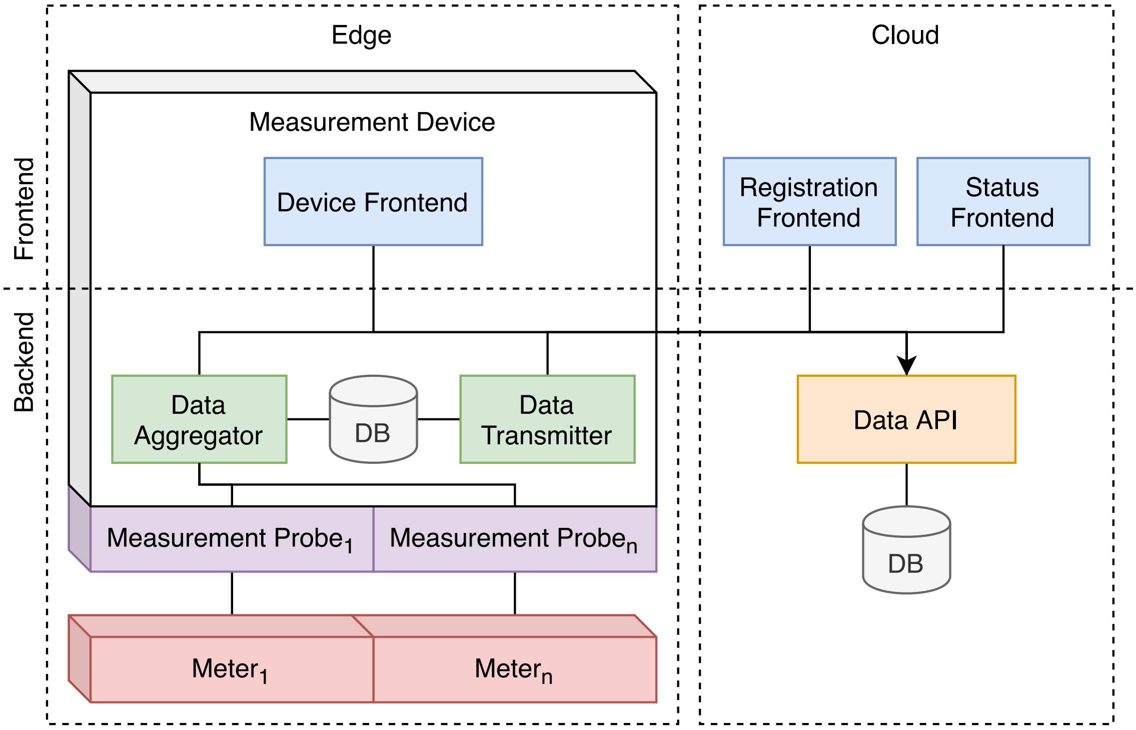

Figure 4.1.: High-level component design

4.2.1. Edge

The edge of this system is composed by the set of measurement devices that are to be installed across the various participating households. Each measurement device will be fitted with the following set of microservices:

Data Aggregator

The Data Aggregator periodically (every 15 minutes) makes measurements for each of the electricity or generation meters connected to this device. Each connection is established on the basis of a dedicated physical measurement probe. Measurements will only be taken if the participant associated with this measurement device has granted his consent. As soon as the participant revokes his consent, measurements will stop indefinitely (comp. subsection 3.1.3). For each measurement made, the Data Aggregator will extract the set of data points that are of interest to the survey and store those, if not empty, as a new single entry in a database along with the measurement date and identifier of the meter from which it originated. The interaction model with the database can be classified as append-only.

Realizes: Subtask (A) on page 19

Data Transmitter

The Data Transmitter periodically (every 60 minutes, at minimum) transfers all of the measurements stored in the Data Aggregator's database to the cloud. Transfers are skipped if the participant associated with this measurement device has revoked or not yet granted his consent. Upon successful transfer, measurements will be deleted to avoid duplicate uploads in the future. Although this sharing of databases violates the database-per-service pattern (comp. subsection 2.1.6), it is still preferred due to the setup's simplicity. Otherwise, the Data Aggregator would have to offer a network interface to retrieve and delete or modify measurements. This overhead is not warranted given the fact that the Data Aggregator exclusively interacts with the database in an append-only mode.

Realizes: Subtask (B) on page 19

Device Frontend

The Device Frontend provides the participant associated with this device with a web-based user interface for granting and revoking his consent, viewing his registration details, as well as the measurements that have been made in and transferred from his household. Additionally, it will list the measurement devices which are in possession of that household and states to which meters and thus, PVSes, each device is connected to. All of these details and actions are retrieved from and performed through the cloud. It shall be noted that a (desired) consequence of locating this component on the edge, rather than in the cloud, is that participants will no longer have access to this interface once they return their devices to the research group.

Realizes: FR-01, FR-02, FR-03, FR-04, FR-05

4.2.2. Cloud

The cloud of this system is composed by the following set of microservices:

Data API

The Data API acts as the single source of truth for each participant's registration details, consent, measurement devices and the measurements themselves. It offers a network interface to retrieve and modify these details, grant and revoke consent, and most importantly, add and retrieve measurements to and from a central data store that is accessible to the research group. Further, it takes care of anonymizing a participant's registration details upon revocation of his consent (see subsection 3.1.4) and tracks a measurement device's version and last date of activity. It may be argued that this component has a monolithic character due to the breadth of methods offered. Yet, the number of methods is not crucial to this judgement, but rather the degree to which they are related (comp. subsection 2.1.2 and subsection 2.1.3).

Enables: FR-01, FR-02, FR-03, FR-04, FR-05, FR-06, FR-07, FR-08, FR-09

Registration Frontend

The Registration Frontend will allow researchers to add new participants, modify their registration details and check whether the consent of a participant has been granted or revoked. All of these details and actions are retrieved from and performed through the Data API.

Realizes: FR-06, FR-07, FR-08

Status Frontend

The Status Frontend provides administrators with a web-based user interface for viewing the health status of each measurement device. Specifically, it will list details such as the device's version, date of the last measurement and moment of activity. For convenience, it will also state to which household and participant a device belongs and whether the participant has granted or revoked his consent. All of these details are retrieved from the Data API.

Realizes: FR-09

4.3. Component Specification

Having given a high-level overview of the components comprising the system, this section will formalize core aspects of the system's two most important microservices.

4.3.1. Data API

Data Model

Given the possibility that one human may wish to act as the contact person of two households which participate in the data collection survey (e.g. his own household and that of his parents), this component will model all data around a logical `Household` entity rather than that of a `User`. However, the contact details provided as part of the registration (see subsection 3.1.2) still indeed semantically belong to an individual user, i.e. a participant. These considerations lead to the initial entity-relationship model given in Figure 4.2.

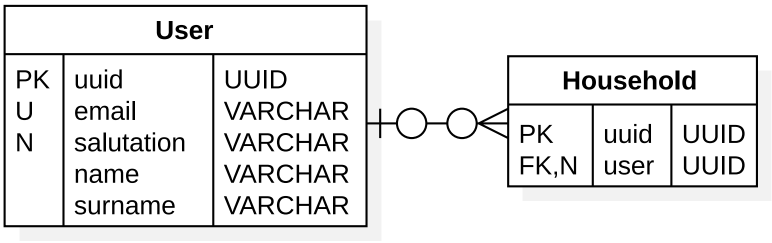

Figure 4.2.: Modeling of participants and households

Next, the answers to the location and metadata subjects of the registration form may be modeled as entities of the same names as shown in Figure 4.3.

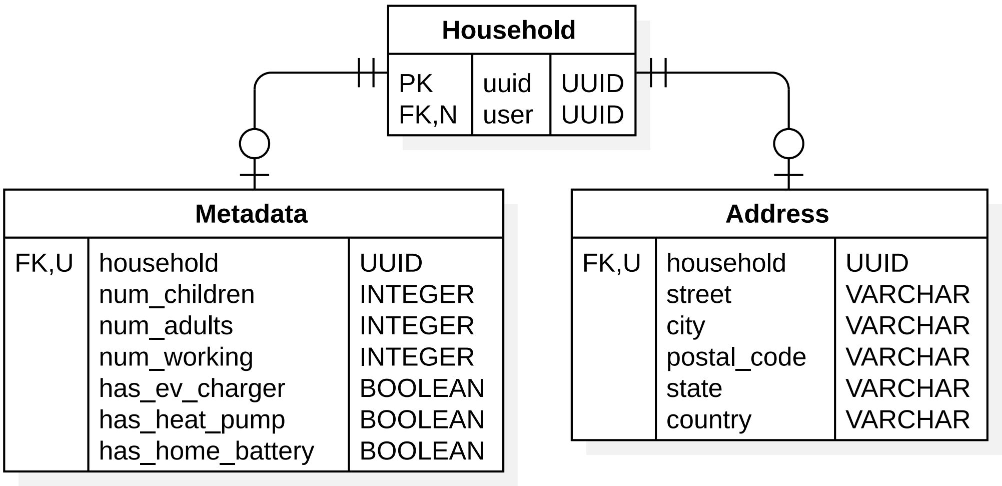

Figure 4.3.: Modeling of a household's metadata and address

Here, the decision to model the relationship as foreign keys with a unique constraint on the `Metadata` and `Address` entities shall be noted. This will lead to a `0..1` cardinality, meaning that this data does not necessarily have to be supplied for an individual household, even if its presence will be enforced on the application layer. The same result may be achieved through nullable foreign keys on the `Household` entity, although that approach would necessitate a primary key on the `Metadata` and `Address` entities which makes less sense conceptually since these details shall not exist on their own.

Before the remaining registration details are modeled, two types of constants shall be introduced with Table 4.1. These constants will be used to differentiate electricity and generation meters, as well as to specify the model number of a meter itself. The latter is needed because the steps to take measurements from a meter will likely vary across models.

Constant	Values
`MeterType`	`ELECTRICITY`, `PV-GENERATION`
`MeterModelNumber`	`MT681`, `MT175`, `EHZ`, `DD3`, `EASY`

Table 4.1.: Meter type and model number constants

Based on these constants, Figure 4.4 models the relationship between a household's meters and PVSes. It also associates each meter with a measurement device to indicate how these objects will be physically connected to each other later. As mentioned in the component design, this `MeasurementDevice` entity will need to track the device's version and date of last activity. Recalling the fact that microservices are deployed independently (comp. subsection 2.1.2), the device's version will need to be split across multiple attributes, each tracking a different component running on the edge.

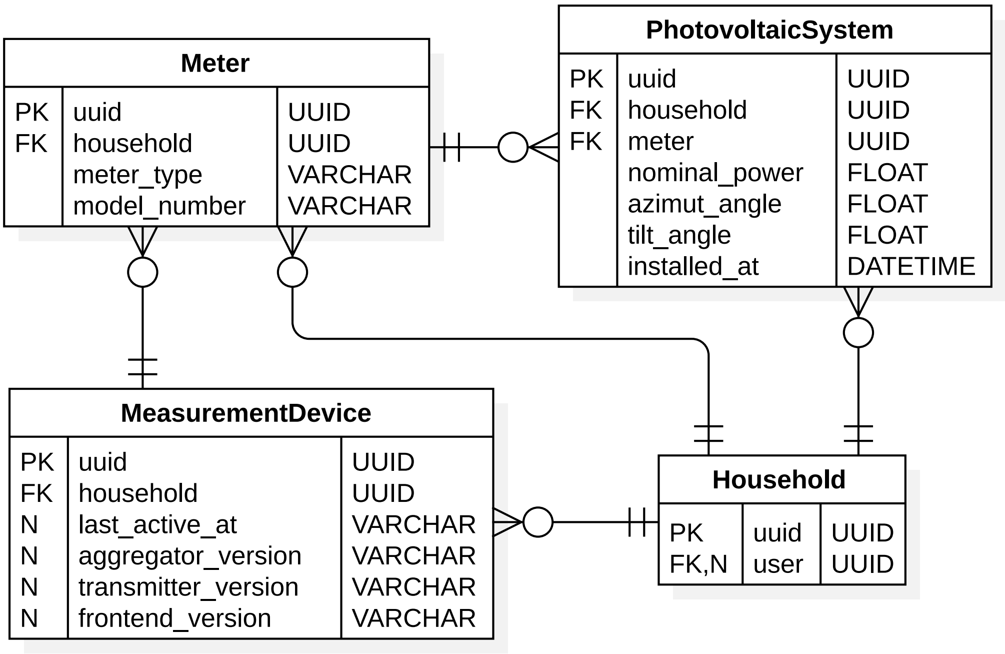

Figure 4.4.: Modeling of a household's meters, PVSes and measurement devices

Now, only runtime-generated data is missing. Beginning with a participant's consent, Figure 4.5 details how the participation status of a household may be modeled. Again, this consent is associated with the `Household` entity rather than the `User` entity because one household may wish to stop sharing data earlier than the other.

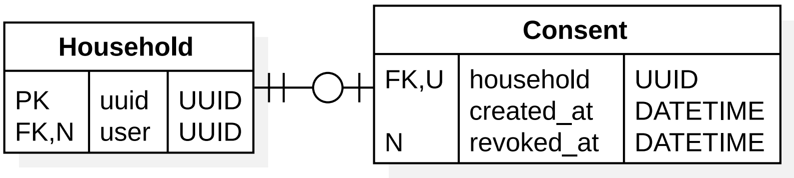

Figure 4.5.: Modeling of a participant's consent

Finally, the measurements taken from a particular meter are modeled in Figure 4.6.

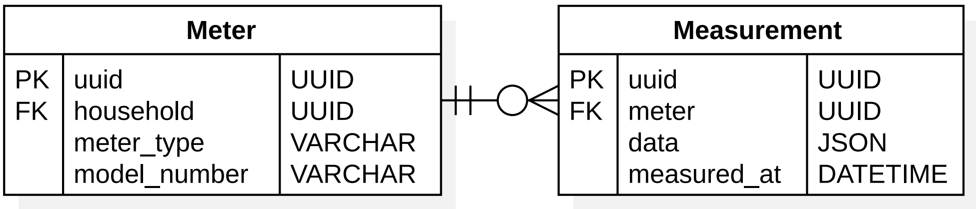

Figure 4.6.: Modeling of a meter's measurements

Of special note is the JSON `data` attribute on the `Measurement` entity which will contain the actual data points that have been observed for a meter at a given point in time. In turn, this means that measurements will be stored in a denormalized manner. The shape and amount of data will vary across measurements. Several factors contribute to this modeling decision:

- electricity and generation meters inherently produce different data
- different meter models may expose different data
- the data points of interest may vary across households and regions

As a result, the responsibility to ensure a consistent representation of measurement data is pushed to the application layer.

Anonymization Considerations In light of the data protection requirements formulated in subsection 3.1.4, the data model shall be briefly verified for compatibility with the anonymization strategy presented therein.

Measure (A) is supported through the nullable foreign `User` entity key modeled on the `Household` entity which allows households to exist without participants and thus, without any contact details (comp. Figure 4.2). On the other hand, Measure (C) can be easily realized by modifying the attributes of a particular PVS instance. However, because the `PhotovoltaicSystem` entity tracks the installation date as a `DATETIME`, this attribute will have to be normalized by setting the date to, for example, the first day of the year, rather than removing the date entirely and only keeping the year (comp. Figure 4.4).

Lastly, even though Measure (C) could be implemented based on the current data model, the following change should be made to further simplify the component's realization. It may be argued that because the process of generating a random location within a larger area is a geospatial operation, i.e. an algorithm which deals with geographic coordinate inputs [53], the component performing such an operation should also maintain geographic coordinates for the models on which it is operating. Currently, however, the `Household` entity is only associated with an `Address` entity that resembles a postal address. Therefore, an additional `Location` entity has been introduced in Figure 4.7.

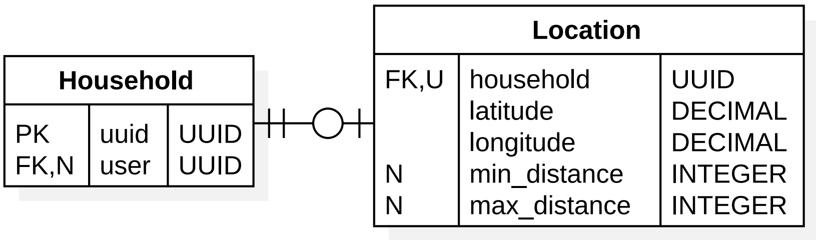

Figure 4.7.: Modeling of a household's location

This location should be created simultaneously with the household's address but will then eventually outlive it once the anonymization has taken place. To create the initial location, the geographic coordinates which, in this case, are modeled as the latitude and longitude in the Global Positioning System (GPS), may be retrieved based on the household's address (*geocoding*).

Service Interface

While the microservice pattern does not dictate any specific kind of network interface for exposing a service's methods (comp. subsection 2.1.2), REST APIs have become ubiquitous and represent the de-facto industry standard approach [27] [62, p. 3]. For that reason, Table 4.2 breaks down the service's functionalities into a set of HTTP-based REST endpoints and maps each endpoint to the requirements satisfied thereof. A complete documentation based on the OpenAPI specification format is available as part of this thesis' hand-in [28].

[27] A full discussion on the principles of REST is given in the Appendix A on page 68.
[28] `https://swagger.io/specification`

#	Method	Path	Requirements
1	GET	`/account`	FR-04
2	GET	`/household`	FR-04
3	POST	`/household/consent`	FR-01
4	DELETE	`/household/consent`	FR-02, FR-03
5	GET	`/household/measurement_device`	
6	GET	`/household/meter`	FR-04
7	GET	`/household/meter/:uuid/measurement`	FR-05
8	POST	`/household/meter/measurement`	Subtask (A)
9	GET	`/household/pv_system`	FR-04
10	GET	`/management/household`	FR-08, FR-09

Table 4.2.: Data API endpoints mapped to requirements

Access Control

Since this microservice will be publicly exposed for consumption by the edge clients, as well as the Registration Frontend and Status Frontend (comp. Figure 4.1), appropriate measures must be put into place to protect the component from unauthorized access. In terms of access control, authentication and authorization are key to these concerns.

Authentication

The component will need to implement some form of authentication in order to verify the identify of the user, i.e. the participant, researcher or administrator, for whom access to a service is requested. Verification typically happens on the basis of something that the requesting party should know (e.g. password), posses (e.g. access card) or incarnate (e.g. biometrics).

Edge clients Edge clients will need to posses a digital certificate and present it with every request. Such certificates are generally used to prove the ownership of a public key by carrying an appropriate signature. Moreover, they can be extended with additional metadata that is then also attested by the signature [35, p. 37]. Based on these cryptographic guarantees, the component will authenticate requests by matching the metadata encoded in a certificate against the stored participant registration details. In particular, it will expect each certificate to encode an email address that belongs to a participant. This lookup scheme is supported by the fact that the `email` attribute in the `User` entity has been modeled with a unique constraint (comp. Figure 4.2), ensuring that any given email address can, at most, identify one participant. Of course, the signature of the presented certificate must also be checked to ensure that it comes from a public key which is exclusively owned by this component or some trusted party associated with the data collection survey. Otherwise, requesting parties could self-attest their identity.

The main benefit of this authentication method is the ability to preload edge devices with certificates [29] [30]. In theory, this allows a measurement device to transmit data as soon as it is connected to a meter.

Registration Frontend and Status Frontend These frontends will need to know a shared secret and present it with every request. The secret will collectively identify the research or administrator group, meaning that individual researchers and administrators are not discerned. Such differentiation is not needed as per the requirements presented in section 3.2.

Authorization

Once users have been identified and authenticated, the component will need to determine whether they are authorized to use the requested service method. Therefore, Table 4.3 details which user group shall be permitted to which of the service's endpoints. Table 4.4 further specifies which set of permissions (*scope*) a user must have to access a particular service endpoint.

User Group	Endpoints
Participants	Endpoint 1 – Endpoint 9
Administrators	Endpoint 10

Table 4.3.: Data API endpoints mapped to user groups

[29] The patent presented in [28] describes a method for binding digital certificates to one or more devices. In case of a breach, access for individual devices, rather than users, can be revoked.

[30] Ammar, Russello, and Crispo warn that embedding a key in a device can represent a security risk, if not regularly rotated, since, eventually, all cryptographic algorithms will be broken [6, p. 23].

Scope	Endpoints
`account`	Endpoint 1
`household`	Endpoint 2, Endpoint 5, Endpoint 6, Endpoint 9
`consent`	Endpoint 3, Endpoint 4
`measurement-read`	Endpoint 7
`measurement-write`	Endpoint 8

Table 4.4.: Data API endpoints mapped to permission scopes

> Scopes support the principle of least privilege. For instance, the Data Transmitter only needs the `measurement-write` scope to perform its primary task of uploading measurements (in addition to the `consent` scope to check whether uploads may take place), whereas the Device Frontend requires the `measurement-read` scope to retrieve individual measurements for display. These scopes may be granted based on the user's group or other metadata which is encoded in a certificate [14, p. 1].

With regard to shielding data of one participant from that of another, the service interface has already been designed from the perspective of the currently authenticated user (comp. Endpoint 1 – Endpoint 9 in Table 4.2). Hence, no additional access control measures are needed.

4.3.2. Data Aggregator

Data Model

In true microservice fashion, this component will only need to model a single entity which is precisely that of a measurement taken from a meter. Its attributes match those of the `Measurement` entity introduced in Figure 4.6, except that the `meter` attribute cannot be modeled as a foreign key. This difference is shown in Figure 4.8.

Measurement		
PK	uuid	UUID
	meter	UUID
	data	JSON
	measured_at	DATETIME

Figure 4.8.: Modeling of an edge client's measurements

Recalling the fact that the application layer is responsible for ensuring a consistent representation of measurement data, a short explanation shall be given as to how this will be achieved. Since the data collection survey is particularly interested in meters which utilize the Smart Message Language (SML) communication protocol to expose internal measurement data (see subsection 3.1.1), its representation of measured values (e.g. counter readings or specific amounts of energy) is decisive for this question. The SML protocol uses the Object Identification System (OBIS) to denote data items in metering equipment. Its codes have been standardized in IEC 62056-6-1 [22]. For this survey, the OBIS codes listed in Table 4.5 are most relevant [31].

Hex Code	Short Code	Description
`0100010800ff`	`1.8.0`	Positive active energy total (in kW h)
`0100020800ff`	`2.8.0`	Negative active energy total (in kW h)
`0100010700ff`	`1.7.0`	Positive active instantaneous power (in kW)
`0100020700ff`	`2.7.0`	Negative active instantaneous power (in kW)
`0100100700ff`	`16.7.0`	Sum active instantaneous power (in kW)

Table 4.5.: Relevant OBIS codes in metering equipment

Given these codes, the component will store the relevant measurement data as a JSON array of objects that map each observed OBIS code to the value observed for that code. However, because the OBIS short code is not uniquely identifiable [32], the hexadecimal representation shall be used.

Control Flow

Although the component's overall control flow has already been discussed in subsection 4.2.1 on page 28, one crucial aspect has not been mentioned so far – namely, that the electricity and generation meters installed in a household may be factory locked. Such meters may not expose all of their internal measurement data. Therefore, it will be necessary to monitor the SML stream for the expected OBIS codes (see Table 4.5), and attempt an unlock if the condition appears to persist over a longer period of time. This revised control flow is depicted in Figure 4.9.

[31] A description of common OBIS codes is available at `https://promotic.eu/en/pmdoc/Subsystems/Comm/PmDrivers/IEC62056_OBIS.htm`

[32] OBIS short codes do not carry the physical medium and channel from which data was generated [22].

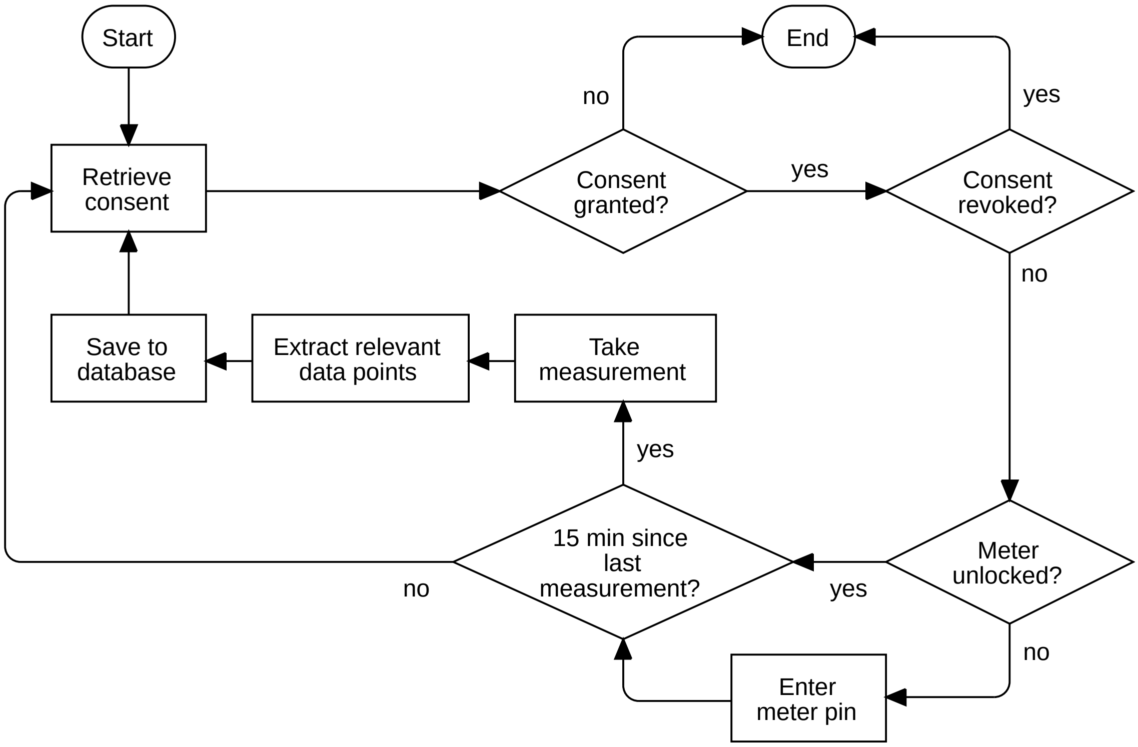

Figure 4.9.: Control flow of the Data Aggregator

Unlocking a meter involves entering a PIN code. This code will, in the most cases, have to be obtained from the household's energy supplier. The actual PIN input may then happen physically in advance to this survey or programmatically at runtime through the meter's optical interface (comp. subsection 3.1.1).

4.4. Component Realization

4.4.1. Certificate-based Authentication

The certificate-based authentication scheme outlined in subsubsection 4.3.1 on page 35 is already standardized as an extension to the TLS protocol, known as Mutual TLS Authentication (mTLS).

Whereas TLS is traditionally used to ensure the secrecy and integrity of data exchanged in a TCP connection, as well as to authenticate the server with which a client is communicating, mTLS offers a way for clients to authenticate themselves to servers, if requested, without having to disclose a shared secret [60, p. 1]. For that purpose, the TLS handshake has been extended with an optional `CertificateRequest` message. This message includes a list of Distinguished Names (DNs) of Certificate Authorities (CAs) that the server trusts, i.e. a list of parties from which certificates must originate. Based on this list, clients can select one or more certificates and respond to the server's authentication request with a `Certificate` message.

These certificates must follow the X.509 standard as defined by the International Telecommunication Union. Finally, the client will have to prove access to the private key corresponding to the public key referenced in the certificate. This is done by calculating and signing a hash of all previous handshake messages which is then sent to the server for verification in a `CertificateVerify` message. It is exactly this randomness that prevents a signature and thus, the certificate, from being exploited in and being susceptible to phishing and replay attacks. Even if an attacker obtains the signature in a man-in-the-middle attack, he cannot impersonate the victim [60, p. 2].

mTLS is suitable to this application context because the service interface of the Data API is based on HTTP (comp. subsubsection 4.3.1) which, in turn, leverages TCP. To successfully use this technology, the system will have to:

- establish its own CA or trust a third party
- issue X.509 certificates that encode the participant's email address and, optionally, the required scopes (comp. subsubsection 4.3.1)
- embed a certificate in each measurement device
- verify the certificate presented with each request [33]

For these tasks, an extensive amount of infrastructure, tools and expertise can be reused [35, p. 37].

Device Frontend Considerations It shall be noted that JavaScript lacks APIs to comfortably interact with a browser's client certificate store. For instance, it is not possible to implement a logout functionality that clears the current session's certificate choice [60, p. 8]. Such functionality, however, is required in client-side web applications, such as the Device Frontend, which should be protected from unauthorized access. Moreover, using certificate-based authentication in such applications assumes that users have access to certificates. In this application context, however, certificates are only embedded in measurement devices. And since these measurement devices are headless (comp. subsection 3.1.1), participants will be required to use another, separate device to access the Device Frontend. This problem can be solved by either:

[33] Parsovs recommends using the TLS session resumption feature to avoid an additional network round-trip and private key operation on repeated requests [60, p. 3].

- sharing the certificate with the participant, so that he can add it to each of the devices used to access the frontend
- funneling each of the frontend's requests through another component (*proxy*) running on the measurement device

For the sake of user experience, as well as security concerns, the latter option should be preferred [34].

4.4.2. Anonymization of Location

The task of anonymizing a household's location in compliance with subsection 3.1.4 can be split into two main steps:

(A) calculate distance bounds d_{min} and d_{max} according to Equation 3.1

(B) generate random location that has a distance (d) from the household's location which satisfies $d_{\text{min}} \leq d \leq d_{\text{max}}$

Beginning with Step (A), an algorithm will, for any given household, first need to lookup the population density per square kilometer (ρ_N). Of course, this density will greatly depend on the chosen granularity (e.g. state, province or region). For Europe, the Nomenclature of Territorial Units for Statistics (NUTS) has gained acceptance. This geocode standard subdivides countries on three hierarchy levels, though each division happens in agreement with the respective member state, meaning that a level in one country may refer to a concept which is different from that in another. In Germany, the levels 1–3 refer to states, government regions and districts, respectively [20, p. 6]. For this application, NUTS 3 will be used.

Now, in order to lookup the population density on this level, the actual NUTS 3 code will have to be determined for the given household. Recalling the fact that each household is associated with a GPS position (comp. Figure 4.7), an algorithm may calculate this code by mapping the position onto a map that draws in the individual subdivisions. In geospatial analysis, this is done using a polygon that represents the geographic bounds of a particular area. A common data format for storing such polygons and other spatial features, i.e. points and lines, is known as a *shapefile* [53]. For Germany, the Federal Agency for Cartography and Geodesy (German: *Bundesamt für Kartographie und Geodäsie*) maintains various such shapefiles for different kinds of subdivisions, including NUTS [35]. Once the code is known, a database, such as

[34] Participants would otherwise become responsible for configuring certificates in their browsers. This may represent a hurdle in itself. Further, no guarantees can be made about their browsers' security. In the worst case, certificates may leak and be used to impersonate a participant.

[35] https://gdz.bkg.bund.de/index.php/default/digitale-geodaten.html

that provided by the Federal Statistical Office (German: *Statistisches Bundesamt*) [36], can be used to retrieve the population density.

Continuing with Step (B), the problem of generating a random location (p) that has a distance (d) from the household's location (p_0) which satisfies $d_{\min} \leq d \leq d_{\max}$ may be understood as illustrated in Figure 4.10.

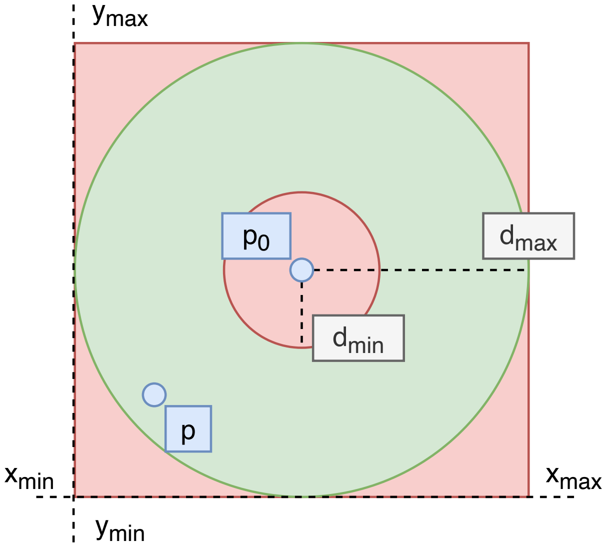

Figure 4.10.: Random location bounds for anonymization

Here, the distance bounds $d_{\min}$ and $d_{\max}$ are used as the radii for circles drawn around a center p_0. Points generated within the green area should be accepted, whereas points in the red areas are either too close or too far away and thus, should be rejected. Again, this problem can be solved with two geographic polygons that represent the inner and outer circles. For the most precise result, each polygon would consist of 360 points. In order to then generate a random location p, an algorithm can take the square bounds ($x_{\min}, x_{\max}, y_{\min}, y_{\max}$) of the outer circle and generate two random coordinates (x, y). The resulting point is then checked against both circles as stated above. The process should be repeated until a valid point and thus, location, has been found. This technique is known as *rejection sampling*.

4.4.3. Meter Interaction

Optical Interface

So far, the probe used to take measurements from a meter, as well as to control its menu, has only been described vaguely. To understand its construction and way of working, the direct local data exchange standard in electricity metering must first be presented.

[36] https://destatis.de/DE/Themen/Laender-Regionen/Regionales/Gemeindeverzeichnis/Administrativ/04-kreise.html

IEC 62056-21 standardizes an optical interface and several protocol modes that permit both reading and programming of meters [37] [21, p. 15]. This optical interface is realized using an infrared receiver and transmitter which interpret and generate light signals according to predefined limiting values (*optocoupler*). In other words, the receiver will, depending on the signal's radiation strength, interpret a light signal as an `ON`- (binary `0`) or `OFF`-condition (binary `1`), whereas the transmitter will, depending on the desired binary state, generate a light signal of appropriate radiation strength [21, pp. 30–31].

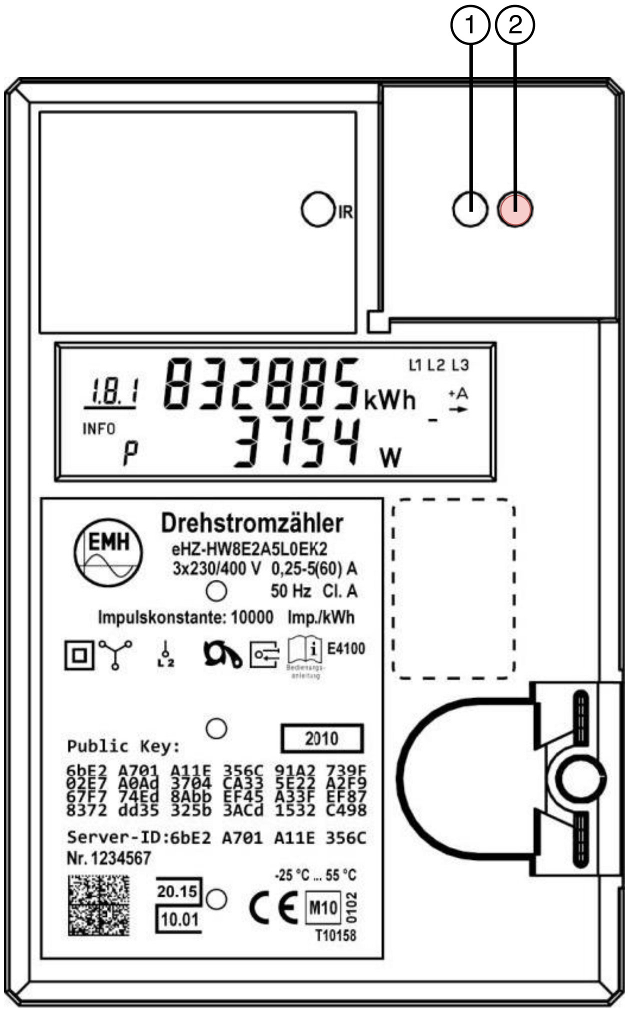

Figure 4.11.: Optical interface of a meter, consisting of an infrared receiver (1) and transmitter (2) [61, p. 7]

Given this description, it is clear that the measurement probe will function analogously, meaning that it also consists of an infrared receiver and transmitter but has these aligned vice-versa to those in the meter. In IEC 62056-21, such a component is known as a reading head [38] [21, p. 29].

[37] In addition to this galvanically isolated interface, IEC 62056-21 also standardizes an electrical coupling method for use in more permanent setups, or when more than one meter needs to be read at one site [21, p. 15].

[38] To ensure broad compatibility of reading heads, the dimensions of the optical port, i.e. the diameter of and distance between the receiver and transmitter, have also been standardized as part of IEC 62056-21 [21, p. 25].

While in theory this setup would allow for an arbitrary data exchange, consumer-facing meters are, by default, configured to operate in protocol mode D which only permits data readout [61, p. 7] [21, p. 35]. The infrared receiver, in this mode, is only used to navigate a meter's menu, including entering a PIN to enable a more detailed readout (see subsubsection 4.3.2). This is likely the reason why a meter's optical port is colloquially referred to as the D0 interface, where 0 stands for the channel number used by electricity metering [21, p. 119].

Data Readout

Once the measurement probe described above is connected to a measurement device (e.g. via USB), the Data Aggregator will have access to a continuous stream of bits that represent parts of the SML telegrams successively emitted by a meter. Naturally, these bits will have to be buffered until a complete telegram can be reconstructed. The exact format, as well as the different types of SML messages available, are standardized in DIN VDE 0418-63-9 [70]. Therefore, this section will not go into further detail, even more so because this task will likely be delegated to a third-party library [39]. However, it shall be noted that multiple different (sub-)versions of the SML communication protocol exist. This must be taken into account by the Data Aggregator.

PIN Input

The process of entering a meter's PIN has, unfortunately, not been standardized as rigorously. In fact, during the course of this project, it has been observed that the steps, i.e. the signal lengths and pauses required in-between, vary from model to model and sometimes, even for the same model. Further, some models protect the detailed data readout by hiding it behind an additional menu setting. Therefore, it is safe to say that this aspect of the system represents the greatest barrier to compatibility and inclusion of a household in the data collection survey.

[39] `https://github.com/spacemanspiff2007/SmlLib`

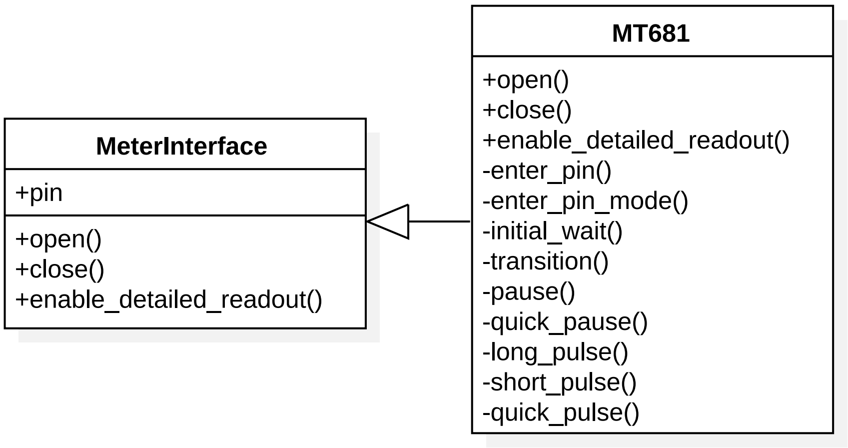

Figure 4.12.: Abstract meter interface to enable a more detailed data readout

To hide this complexity and thus, streamline application code, the interaction (e.g. `enable_detailed_readout()`) with a specific meter model (e.g. `MT681`) shall happen on the basis of a generic `MeterInterface` class. This class may be subclassed to create specialized versions that encapsulate the timing details for each particular meter model (comp. Figure 4.12). At runtime, the factory pattern may then be used to obtain an appropriate instance.

4.5. Deployment Model

In light of the lengthy discussion on the benefits of containerized applications in subsection 2.2.4 and subsection 2.3.1, it should come as no surprise that this system will fully embrace containers as the deployment target and medium of choice. This choice is also a major enabler to the system's portability since containers are not bound to a specific hardware platform. They only need to be compatible with the host's kernel and CPU architecture (comp. subsection 2.2.2). On the other hand, to simplify the management of these containers, the system will rely on a container orchestration platform as was presented in section 2.3.

4.5.1. Target Hardware

A brief overview of the currently targeted hardware and thus, CPU architectures, shall be given in the following [40]:

Edge

Each measurement device shall be realized on the basis of a Raspberry Pi 4 Model B with 4 GB of memory. This single-board computer is equipped with a 64-bit quad-core Cortex-A72 processor that is based on version 8 of the ARM instruction set architecture (`armv8`) [63] [7]. In this price range, most devices run on ARM. Therefore, this will be the primary target architecture going forward [41].

Cloud

The set of services representing the cloud shall be deployed using the compute infrastructure provided by the university. The exact configuration matters less in this case because, here, resources are provisioned on the basis of VMs which, on request, may be reconfigured for more memory or CPU cores. More importantly, all VMs will run on the 64-bit version of the x86 instruction set architecture (`x86-64`). This architecture has, traditionally, always been used for server workloads which is why it will be the primary target going forward [42].

It shall be noted that the development of the components comprising this system will take place in an x86-based environment. Therefore, so-called multi-architecture builds will be necessary to enable local testing. Such functionality has recently been added to Docker [26]. Technically, this is achieved through emulation by a hypervisor [43].

4.5.2. Container Builds and Registry

As previously hinted at, this project will leverage Docker to build container images. Portability of build artifacts, again, is ensured by the fact that Docker builds OCI-compliant images (comp. subsection 2.2.4). Now, to use these images in an actual

[40] The Linux kernel versions have been disregarded because container runtimes, such as the one used by Docker, only requires version 3.10 which is an almost 9-year old release that reached its end of life in late 2017 [36] [44].

[41] Usage of the ARM architecture generally also leads to more power-efficient designs, which is a valuable trait considering that measurement devices will run on a household's own electricity [54, p. 8837].

[42] Amazon Web Services has only recently launched ARM-based versions of its compute instance families. According to a Bloomberg report, Microsoft is also designing its own chips for use in its cloud computing offerings and consumer devices [41].

[43] While Docker ensures that a build is performed in the context of a specific CPU architecture, it is up to the developer to ensure that libraries and application code are compatible with that particular architecture.

deployment, the build artifacts will have to be distributed to the execution hosts. In non air-gapped systems such as this one, a container image registry is typically used for this task. However, because the licensing model of this application has not yet been decided upon, a private registry, or rather repository, shall be used, i.e. one to which only a limited audience has access to. Finally, to make this process automatic and reproducible for any given revision of a component's code, a continuous delivery pipeline will be employed as is described in the Appendix A on page 72.

4.5.3. Container Orchestrator

To manage and deploy the aforementioned containers, this project will rely on Kubernetes [44], a state-of-the-art container orchestration platform that exhibits and goes well beyond the minimum set of characteristics pressed for in subsection 2.3.2. Originally developed at Google, Kubernetes builds upon more than a decade and a half of experience running containerized production workloads at scale [13, p. 50]. It is also the primary solution recommended and hosted by the Cloud Native Computing Foundation [58, p. 13].

Resource Considerations Even though the currently targeted edge client hardware is powerful in comparison to its size, care should be taken in the choice of container orchestrator. Optimizing this choice for resource usage will allow less well-equipped hardware to be used in the future. Moreover, whereas an orchestrator running in the cloud typically attempts to maximize resource utilization, single-node edge orchestrators will not have to move and migrate containers as frequently and thus, can be built to use minimal resources [33, p. 2]. For this purpose, lightweight distributions of Kubernetes have emerged which promise savings by modifying and reorganizing some of the core components [12, pp. 65–66]. Based on the findings in [12] and [33], this project will use the K3s distribution of Kubernetes to manage and deploy containers on the edge [45].

4.6. Maintenance and Support Plan

A major objective of this thesis is the design of a maintenance and support plan through which the system can be effortlessly operated, upgraded and serviced over the course of the data collection survey (comp. section 1.2). This is especially important considering that measurement devices will be located in various geographic regions,

[44] `https://kubernetes.io`
[45] `https://k3s.io`

making physical access to these devices impossible or impractical at the very least. More so, it cannot be expected that the participants of the survey will have enough technical knowledge to aid in support beyond rebooting a particular device. Therefore, this section will discuss three approaches relevant to this stage of the application lifecycle.

4.6.1. Remote Updates

While it is easy to see how the initial setup of a measurement device and deployment of the services running on it may happen on the basis of a custom system image, it is less clear how these components will be updated once the device has been installed in a household. For this, a high-level understanding of a deployment in Kubernetes is needed.

At the heart, Kubernetes works in a declarative manner, meaning that users describe a desired cluster state based on which control loops then watch the cluster and make or request changes that move the current state closer to this desired state (comp. MAPE-K cycle in subsection 2.3.3) [23]. For instance, so-called replication controllers will monitor a service's replica count and attempt to create or destroy additional instances if the actual count does not match the user-provided specification. Now, given the fact that Kubernetes uses configuration files to formulate such a desired cluster state [46], a workflow can be designed that allows components to be updated remotely without requiring a participant's input. Of course, this workflow may also be used to update the cloud's components.

In this workflow, a VCS repository will contain the complete set of configuration files needed to describe a particular revision of the system's deployment. Then, a so-called operator, i.e. an extension to Kubernetes, will continuously monitor the repository for changes and instruct the orchestrator to use the most recent revision as the cluster's desired state [47].

[46] Kubernetes also allows for the possibility to use imperative commands for managing a cluster. However, because these commands effectively only alter the previous desired state, i.e. that specified through the series of commands executed prior, this method is not listed explicitly [47].

[47] `https://fluxcd.io`

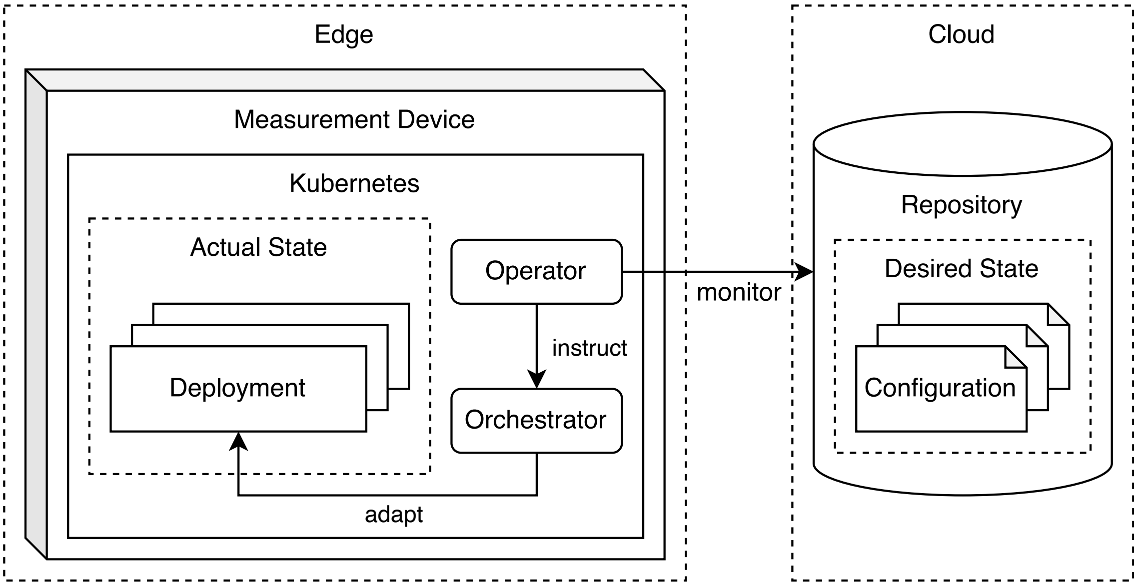

Figure 4.13.: Workflow to achieve remote component updates

This workflow inherently decouples the update process from the execution hardware, allowing individual devices to fail intermittently but then self-heal by applying the latest configuration revision. Decoupling these steps also helps in scaling the system since administrators will not have to interact with each individual measurement device in order to perform an update.

Two additional advantages of this approach shall be noted. Firstly, the workflow prepares for the possibility to practice Continuous Deployment (see Appendix A on page 72). This practice can be easily attained by extending the pipeline mentioned in subsection 4.5.2 to update the cluster configuration files whenever a new container image has been built. Secondly, by storing the cluster's configuration in a VCS repository, users outside of the operations team can be empowered to do their own operations, lowering the bar for self-service systems [48] [43, p. 38].

Realizes: FR-10

4.6.2. Remote Access

For debugging purposes and other maintenance activities, it will be necessary to remotely connect to measurement devices via SSH. However, because these devices will be situated on a network whose firewall only allows outbound connections, i.e. the household's LAN, direct access (*forward connection*) will not be possible. Besides, it is unlikely that the household will have a publicly reachable IP address [49]. Such

48 The article in [43] goes into great detail on the benefits of this VCS-driven workflow and culture, known as GitOps.

49 ISPs use NAT to route traffic from the internet to their customers who have been assigned private IP addresses. The reason why customers have been given these addresses is due to

(static) public IPs are usually reserved for commercial internet plans. Instead, a *reverse connection* approach will be used as is shown in Figure 4.14.

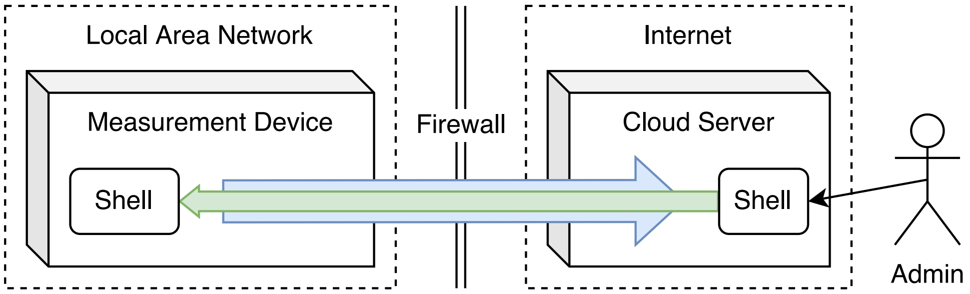

Figure 4.14.: Reverse SSH tunnel to allow remote access

Here, each measurement device will establish a long-lived TCP/IP connection (*tunnel*) with one of the system's cloud servers which are situated on the public internet and to which administrators will have access. Once an administrator connects to that particular cloud server, he will be able to start a regular SSH session with the remote measurement device by specifying the port of the previously opened tunnel in his connection request (*reverse SSH tunneling*).

Obviously, a measurement device will have to ensure that its tunnel remains active over the course of the study. On the other hand, each measurement device needs to connect to the cloud server using a different port number. Such automatic retry and port negotiation functionality is commonly found in open-source solutions [50].

Realizes: FR-11

4.6.3. Monitoring

Although the solutions presented in the previous sections will enable administrators to upgrade and service individual measurement devices, no method has been laid out by which potential defects can be recognized proactively. This is important as participants will likely not monitor their devices for failure, hereby incurring a chance of missed measurements. This section concludes the system's design by outlining two more solutions that address these concerns.

Meter Incidents

It is clear that the system's effectiveness depends on the cyber-physical interaction with a meter such as the one described in subsection 4.4.3. Yet, because this

the limited amount of public IPv4 addresses. IPv6 may change this situation with its 128-bit (versus 32-bit) address space.

50 https://github.com/fatedier/frp

interaction happens on the basis of an optical interface, it cannot be guaranteed that the communication will be stable at all times. For instance, small alterations in the alignment of transmitters and receivers may cause interruptions. Furthermore, the physical connection of the measurement probe to the measurement device may become faulty or unresponsive. Lastly, recalling the fact that the timing details to navigate a meter's menu vary from model to model and sometimes, even for the same model, it might not be possible to enable the detailed data readout of a meter. In all of these cases, administrators shall be made aware of the situation, so that they can proactively connect to a particular measurement device and attempt to resolve the issue (*incident*) [51]. For this reason, Table 4.6 lists three additional endpoints of the Data API that will be used by the Data Aggregator to track these incidents on a per-meter basis. To distinguish between the different types of incidents, Table 4.7 introduces another constant.

#	Method	Path
11	`GET`	`/household/meter/:uuid/incident`
12	`POST`	`/household/meter/:uuid/incident/report`
13	`POST`	`/household/meter/:uuid/incident/resolve`

Table 4.6.: Data API endpoints to track meter incidents

Constant	Values
`MeterIncidentType`	`SERIAL-PORT-UNAVAILABLE` `OBIS-CODES-UNAVAILABLE`

Table 4.7.: Meter incident constants

The control flow of this monitoring measure is rather simple. Upon initialization, the Data Aggregator will fetch the list of incidents previously reported for each of the meters connected to the device. If that specific type of incident has already been reported or resolved, respectively, the Data API will take no action. If, however, a previously resolved incident is reported again, the case will be re-opened. This effectively means that there will be no history of incidents, although that can be easily changed by simply appending to the list of incidents instead of modifying existing instances.

[51] Given the geographic distance of a household, participants may also be asked to re-align the measurement probe or unplug the measurement device from its power source.

Figure 4.15 showcases how incidents will be modeled on sides of the Data API.

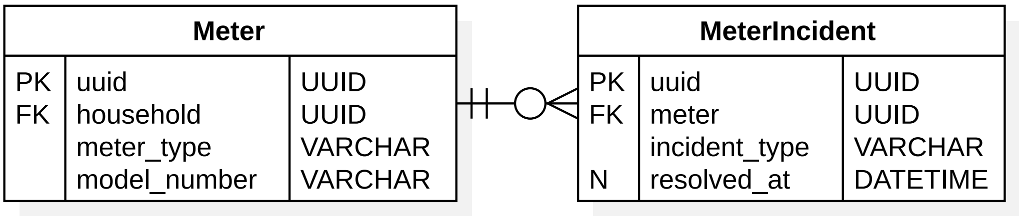

Figure 4.15.: Modeling of a meter's incidents

Error Logging

As a best practice in any application, this project will leverage a third-party error tracking software that remotely collects errors [52], including their stack traces, so that production code can be continuously monitored and improved. This solution will be integrated into each of the system's custom built components, regardless of whether they run on the edge or cloud. To name a few practical benefits, such error tracking platforms will allow developers and administrators to:

- receive alerts on new errors in real-time
- automatically group similar errors
- correlate errors with a component's version and other user metadata

[52] https://rollbar.com

5. Evaluation

This thesis has carefully laid out a blueprint for constructing, deploying and operating a system that can be used as the backbone to the data collection survey presented in section 3.1. Based on this blueprint, a production-grade implementation has been prepared in parallel and provided to the research group for both initial testing and actual use in the survey. To quantitatively asses the quality and readiness of its design, the measurements collected as part of a field test shall be evaluated against the core functional requirements of the system, which are (comp. subsection 3.1.1):

(A) taking measurements from multiple electricity and generation meters in a 15-minute interval (*measurement cycle*)

(B) transmitting the observed measurements to a centralized data store in a 60-minute interval (*report cycle*)

Because the contents of a measurement are already verified at the time of their creation (comp. Data Aggregator on page 28), only the interval constraints listed above need to be checked for across the collected data set. At the same time, these interval constraints define the exact number of measurements and reports that should have been made for any given meter in a particular observation timeframe. It is simply the timeframe's duration in minutes divided by the respective interval length [53]. In the following, *target fulfillment* denotes the degree to which this is satisfied. Finally, (B) implies that, for any given meter, each report carries exactly four measurements.

The data set for which these expectations are tested captures the past three weeks of system operations, i.e. exactly the timeframe since the last major component revision. It covers 10 meters across 8 distinct households and 3 meter models. As such, the body of analysis can be calculated as 20,160 measurements. Table 5.1 and Table 5.2 present the combined statistical results. A per-meter break-down is given in the Appendix C on pages 74 and 75.

[53] This calculation assumes that the time of the first measurement always aligns with the start of the observation timeframe. However, because measurement devices start independently from each other, the actual calculation should ensure that the difference in time between the observation timeframe's start and that of the first measurement is subtracted accordingly.

	$\overline{\mathbf{X}}$	$\widetilde{\mathbf{X}}$	$\boldsymbol{\sigma}$
Measurement cycle	14.80	15.01	1.18
Report cycle	60.00	60.00	0.11
Measurements per report	4.05	4.00	0.24

Table 5.1.: Measurement and report cycle statistics over past three weeks of operations

Measurement target fulfillment	101.23 %
Report target fulfillment	100.00 %

Table 5.2.: Measurement and report target fulfillment over past three weeks of operations

The results are very promising as they suggest a near perfect level of system availability. This is especially true on sides of the Data Transmitter and Data API since a missed, failed or rejected report would result in those measurements being transmitted upon the next scheduled iteration, leading to a lower report target fulfillment, which currently shows no decimal places before rounding. The slight variance in report cycles can be explained by the fact that the Data Transmitter has been deployed as a Kubernetes `CronJob` whose status is only checked every 10 seconds [24].

On the other hand, the results indicate that, on average, each meter has taken more measurements than necessary. This can be traced back to an overlook in the design of the Data Aggregator which, upon failure, currently does not consider the date of the last measurement taken. Instead, it will immediately take another measurement, effectively ignoring the measurement cycle constraint between restarts and thus, producing a higher than expected count in measurements. This higher count also directly correlates with the higher than expected mean in measurements per report.

While the these statistics in itself are promising, the system's quality shall further be evaluated in the context of the non-functional requirements set fourth in section 3.3:

Security (NFR-01)

Subsection 4.3.1 introduced an extensive role-based access control model for the Data API that supports the principle of least privilege, allowing individual components to be restricted to the set of permissions needed to fulfill their tasks. As a bonus, the Data API is prepared for an ecosystem of community solutions that want to share data through means other than the measurement devices designed herein. For this purpose, it additionally implements a JWT-based authentication solution [54]. Beyond that, all communications, including the

[54] `https://jwt.io/introduction`

long-lived maintenance and support tunnel (see subsection 4.6.2), are secured via TLS and authenticated with proper credentials.

Scalability (NFR-02)

While the edge of this system is limited to vertical scaling, the cloud is prepared to scale horizontally. This is facilitated by the use of a container orchestration platform which makes the management of a multi-node cluster largely opaque to the administrator (comp. subsection 2.3.2). For long-running tasks, such as future data analytics, the current deployment also includes a distributed task queue [55].

Availability (NFR-03)

To achieve a high availability, the system relies on both the cluster's self-healing capabilities, as well as more traditional replication techniques. The effectiveness and stability of the solution have largely been proven through the statistics presented at the beginning of this chapter.

Maintainability (NFR-04, NFR-05)

In support of future development, all components have been implemented using a static type checker, linter and formatter. These tools enforce best practices and a common legible style across all code contributed. Dependencies have been carefully chosen for a high degree of developer support. Furthermore, these dependencies can be automatically updated when new releases are available, ensuring that critical security patches can distributed in a timely manner [56].

Testability (NFR-06)

A total of 114 tests have been devised and implemented for the Data API, each of which consist of several sub-conditions to be tested for. These tests are run on each new component revision (see Continuous Integration on page 71) [57], helping to detect regressions as soon as possible.

Portability (NFR-07)

The choice of containers as the deployment target and medium of choice largely enables effortless compatibility with a wide range of execution environments (comp. subsection 2.2.2). More so, the project includes multi-architecture builds to be compatible with the two most popular CPU architectures, i.e. `x86` and `ARM` (see subsection 4.5.1). Besides that, Kubernetes is also offered as a managed service by all major cloud providers, allowing the cloud realm of this system to be hosted on a pay-per-use model [58].

[55] https://docs.celeryproject.org/en/stable/
[56] https://github.blog/2020-06-01-keep-all-your-packages-up-to-date-with-dependabot/
[57] https://github.com/features/actions
[58] https://cloud.google.com/kubernetes-engine

6. Summary

Bibliography

[1] Leila Abdollahi Vayghan et al. "Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 970–973. DOI: 10.1109/CLOUD.2018.00148.

[2] Leila Abdollahi Vayghan et al. "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*. 2019, pp. 176–185. DOI: 10.1109/QRS.2019.00034.

[3] Energy Information Administration. *Net consumption of electricity worldwide in select years from 1980 to 2018*. Statista. 2021.

[4] International Energy Agency. *Energy demand in data centers worldwide from 2015 to 2021, by type*. Statista. 2020.

[5] Muhammad Alam et al. "Orchestration of Microservices for IoT Using Docker and Edge Computing". In: *IEEE Communications Magazine* 56.9 (2018), pp. 118–123. DOI: 10.1109/MCOM.2018.1701233.

[6] Mahmoud Ammar, Giovanni Russello, and Bruno Crispo. "Internet of Things: A survey on the security of IoT frameworks". In: *Journal of Information Security and Applications* 38 (2018), pp. 8–27. DOI: 10.1016/j.jisa.2017.11.002.

[7] *Armv8-A Instruction Set Architecture*. 1.1. Arm Limited. 2020.

[8] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52. DOI: 10.1109/MS.2016.64.

[9] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/.

[10] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[11] Carl Boettiger. "An Introduction to Docker for Reproducible Research". In: *SIGOPS Oper. Syst. Rev.* 49.1 (2015), pp. 71–79. DOI: 10.1145/2723872.2723882.

[12] Sebastian Böhm and Guido Wirtz. "Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes". In: *Conference: 13th Central European Workshop on Services and their Composition*. 2021.

[13] Brendan Burns et al. “Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade”. In: *Queue* 14.1 (2016), pp. 70–93. DOI: 10.1145/2898442.2898444.

[14] Alan B Butt, Paul B Hillyard, and Jin Su. *Certificate-based authentication system for heterogeneous environments*. US Patent 6,754,829. 2004.

[15] Emiliano Casalicchio. “Container Orchestration: A Survey”. In: *Systems Modeling: Methodologies and Tools*. Ed. by Antonio Puliafito and Kishor S. Trivedi. Cham: Springer International Publishing, 2019, pp. 221–235. DOI: 10.1007/978-3-319-92378-9_14.

[16] Antonio Celesti et al. “Exploring Container Virtualization in IoT Clouds”. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2016, pp. 1–6. DOI: 10.1109/SMARTCOMP.2016.7501691.

[17] Tomas Cerny, Michael J. Donahoo, and Michal Trnka. “Contextual Understanding of Microservice Architecture: Current and Future Directions”. In: *SIGAPP Appl. Comput. Rev.* 17.4 (2018), pp. 29–45. DOI: 10.1145/3183628.3183631.

[18] Lianping Chen. “Continuous Delivery: Huge Benefits, but Challenges Too”. In: *IEEE Software* 32.2 (2015), pp. 50–54. DOI: 10.1109/MS.2015.27.

[19] Ben Christensen. *Optimizing the Netflix API*. 2013. URL: https://netflixtechblog.com/optimizing-the-netflix-api-5c9ac715cf19.

[20] European Commission and Eurostat. *Statistical regions in the European Union and partner countries : NUTS and statistical regions 2021 : 2020 edition*. Publications Office, 2020. DOI: 10.2785/72829.

[21] International Electrotechnical Commission. *Electricity metering - Data exchange for meter reading, tariff and load control - Part 21: Direct local data exchange*. 2002.

[22] International Electrotechnical Commission. *Electricity metering data exchange - The DLMS/COSEM suite - Part 6-1: Object Identification System (OBIS)*. 2017.

[23] *Controllers*. URL: https://kubernetes.io/docs/concepts/architecture/controller/.

[24] *CronJob*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/.

[25] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and Ulterior Software Engineering*. Ed. by Manuel Mazzara and Bertrand Meyer. Cham: Springer International Publishing, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12.

[26] Jeremie Drouet. *Multi-arch build and images, the simple way*. 2020. URL: https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/.

[27] Michael Eder. "Hypervisor-vs. container-based virtualization". In: *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)* 1 (2016).

[28] Craig S Etchegoyen and Dono Harjanto. *Device-bound certificate authentication*. US Patent 8,438,394. 2013.

[29] Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321125215.

[30] Roy T Fielding. *REST APIs must be hypertext-driven*. 2008. URL: https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven.

[31] Roy Thomas Fielding and Richard N. Taylor. "Architectural Styles and the Design of Network-Based Software Architectures". PhD thesis. 2000. ISBN: 0599871180.

[32] Martin Fowler. *Continuous integration*. 2006. URL: https://www.martinfowler.com/articles/continuousIntegration.html.

[33] Tom Goethals, Filip De Turck, and Bruno Volckaert. "FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices". In: *Internet of Vehicles. Technologies and Services Toward Smart Cities: 6th International Conference, IOV 2019, Kaohsiung, Taiwan, November 18–21, 2019, Proceedings*. Kaohsiung, Taiwan: Springer-Verlag, 2019, pp. 174–189. DOI: 10.1007/978-3-030-38651-1_16.

[34] Saiful Hoque et al. "Towards Container Orchestration in Fog Computing Infrastructures". In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. 2017, pp. 294–299. DOI: 10.1109/COMPSAC.2017.248.

[35] René Hummen et al. "Towards Viable Certificate-Based Authentication for the Internet of Things". In: *Proceedings of the 2nd ACM Workshop on Hot Topics on Wireless Network Security and Privacy*. HotWiSec '13. Budapest, Hungary: Association for Computing Machinery, 2013, pp. 37–42. DOI: 10.1145/2463183.2463193.

[36] *Install Docker Engine from binaries*. URL: https://docs.docker.com/engine/install/binaries/.

[37] Isam Mashhour Al Jawarneh et al. “Container Orchestration Engines: A Thorough Functional and Performance Comparison”. In: *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*. 2019, pp. 1–6. DOI: 10.1109/ICC.2019.8762053.

[38] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. “Challenges When Moving from Monolith to Microservice Architecture”. In: *Current Trends in Web Engineering*. Ed. by Irene Garrigós and Manuel Wimmer. Cham: Springer International Publishing, 2018, pp. 32–47. DOI: 10.1007/978-3-319-74433-9_3.

[39] Hui Kang, Michael Le, and Shu Tao. “Container and Microservice Driven Design for Cloud Infrastructure DevOps”. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. 2016, pp. 202–211. DOI: 10.1109/IC2E.2016.26.

[40] Asif Khan. “Key Characteristics of a Container Orchestration Platform to Enable a Modern Application”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 42–48. DOI: 10.1109/MCC.2017.4250933.

[41] Ian King and Dina Bass. *Microsoft Designing Its Own Chips for Servers, Surface PCs*. 2020. URL: https://www.bloomberg.com/news/articles/2020-12-18/microsoft-is-designing-its-own-chips-for-servers-surface-pcs.

[42] Marko Leppänen et al. “The highways and country roads to continuous deployment”. In: *IEEE Software* 32.2 (2015), pp. 64–72. DOI: 10.1109/MS.2015.50.

[43] Thomas A. Limoncelli. “GitOps: A Path to More Self-Service IT: IaC + PR = GitOps”. In: *Queue* 16.3 (2018), pp. 13–26. DOI: 10.1145/3236386.3237207.

[44] *Linux 3.10.108 (EOL)*. 2017. URL: https://lkml.org/lkml/2017/11/4/178.

[45] *Linux man pages online*. URL: https://man7.org/linux/man-pages/.

[46] Yutu Liu, Anne H. Ngu, and Liang Z. Zeng. “QoS Computation and Policing in Dynamic Web Service Selection”. In: *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*. WWW Alt. ’04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 66–73. DOI: 10.1145/1013367.1013379.

[47] *Managing Kubernetes Objects Using Imperative Commands*. URL: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/imperative-command/.

[48] A. L. Martínez-Ortiz et al. “A Quality Model for Web Components”. In: *Proceedings of the 18th International Conference on Information Integration and Web-Based Applications and Services*. iiWAS ’16. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 430–432. DOI: 10.1145/3011141.3011203.

[49] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology. 2011. DOI: 10.6028/NIST.SP.800-145.

[50] Dirk Merkel. “Docker: Lightweight Linux Containers for Consistent Development and Deployment”. In: *Linux J.* 2014.239 (2014).

[51] Antonio Messina et al. “A Simplified Database Pattern for the Microservice Architecture”. In: *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*. 2016, pp. 35–40. DOI: 10.13140/RG.2.1.3529.3681.

[52] Mathias Meyer. “Continuous Integration and Its Tools”. In: *IEEE Software* 31.3 (2014), pp. 14–16. DOI: 10.1109/MS.2014.58.

[53] Jacqueline W. Mills. “Geospatial Analysis: A Comprehensive Guide to Principles, Techniques, and Software Tools, Second Edition - by Michael J. de Smith, Michael F. Goodchild, and Paul A. Longley”. In: *Transactions in GIS* 12.5 (2008), pp. 645–647. DOI: 10.1111/j.1467-9671.2008.01122.x.

[54] Roberto Morabito. “Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation”. In: *IEEE Access* 5 (2017), pp. 8835–8850. DOI: 10.1109/ACCESS.2017.2704444.

[55] Roberto Morabito, Jimmy Kjällman, and Miika Komu. “Hypervisors vs. Lightweight Virtualization: A Performance Comparison”. In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[56] Dmitry Namiot and Manfred sneps sneppe. “On Micro-services Architecture”. In: *Interenational Journal of Open Information Technologies* 2 (2014), pp. 24–27.

[57] Claus Pahl. “Containerization and the PaaS Cloud”. In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31. DOI: 10.1109/MCC.2015.51.

[58] Claus Pahl et al. “Cloud Container Technologies: A State-of-the-Art Review”. In: *IEEE Transactions on Cloud Computing* 7.3 (2019), pp. 677–692. DOI: 10.1109/TCC.2017.2702586.

[59] European Parliament and Council of the European Union. “Regulation on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”. In: *Official Journal of the European Union* 59 (2016), pp. 1–88.

[60] Arnis Parsovs. “Practical Issues with TLS Client Certificate Authentication”. In: *Network and Distributed System Security Symposium*. 2014. DOI: ndss.2014.23036.

[61] *Produkthandbuch Elektronischer Haushaltszähler eHZ*. 09/2011. Nordwestdeutsche Zählerrevision. Heideweg 33, 49196 Bad Laer.

[62] RapidAPI. *Developer Survey 2020 Report*. 2021.

[63] *Raspberry Pi 4 Model B specifications*. URL: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[64] Tony Savor et al. "Continuous Deployment at Facebook and OANDA". In: *Proceedings of the 38th International Conference on Software Engineering Companion*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 21–30. DOI: 10.1145/2889160.2889223.

[65] Vitor Silva, Marite Kirikova, and Gundars Alksnis. "Containers for Virtualization: An Overview". In: *Applied Computer Systems* 23 (2018), pp. 21–27. DOI: 10.2478/acss-2018-0003.

[66] Vindeep Singh and Sateesh K Peddoju. "Container-based microservice architecture for cloud applications". In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017, pp. 847–852. DOI: 10.1109/CCAA.2017.8229914.

[67] Patrick Stoy. "Konzeption und Entwicklung standardisierter Schnittstellen zur automatisierten regionalen Vermarktung kleiner Mengen an erneuerbaren Energien". MA thesis. Hochschule RheinMain, 2019.

[68] Davide Taibi and Valentina Lenarduzzi. "On the Definition of Microservice Bad Smells". In: *IEEE Software* 35.3 (2018), pp. 56–62. DOI: 10.1109/MS.2018.2141031.

[69] Johannes Thönes. "Microservices". In: *IEEE Software* 32.1 (2015), pp. 116–116. DOI: 10.1109/MS.2015.11.

[70] Deutsche Kommission Elektrotechnik Elektronik Informationstechnik in DIN und VDE. *DIN VDE 0418-63-9: Messeinrichtungen und -systeme - Teil 63-9: Intelligentes Kommunikationsprotokoll für Elektrizitätszähler (SML)*. 2022.

[71] Mario Villamizar et al. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *2015 10th Computing Colombian Conference (10CCC)*. 2015, pp. 583–590. DOI: 10.1109/ColumbianCC.2015.7333476.

[72] Manish Virmani. "Understanding DevOps & bridging the gap from continuous integration to continuous delivery". In: *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*. 2015, pp. 78–82. DOI: 10.1109/INTECH.2015.7173368.

[73] Miguel G. Xavier et al. "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments". In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 2013, pp. 233–240. DOI: 10.1109/PDP.2013.41.

[74] Ying Xiong et al. "Extend Cloud to Edge with KubeEdge". In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 373–377. DOI: 10.1109/SEC.2018.00048.

Glossary

cloud computing

Describes a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [49, p. 2].

code rot

Describes the slow deterioration of software quality over time due to, for example, changes in the execution environment, compatibility between the software's components or unnoticed bugs.

node affinity

Set of rules by which a scheduler can determine which hosts to select as the deployment targets.

scale cube

Describes a three-dimensional model for scaling an application. X-axis scaling refers to running multiple instances of an application behind a load balancer. Y-axis scaling splits an application into multiple, distinct services. Z-axis scaling partitions the data to be processed across a set of application instances [51, p. 36].

Appendices

A. Theoretical Framework

A.1. Representational State Transfer

Coined and theorized by Fielding and Taylor, Representational State Transfer (REST) stands for an architectural style of distributed hypermedia systems that was motivated by the need to create a model for how the modern web should work, thereby serving as the guiding framework for web protocol standards such as HTTP and URI [31, pp. 76, 107].

Like any software architecture, this named set of constraints intends to outline how a well-designed network-based application behaves, i.e. how the system is partitioned and how components communicate [31, p. xvi], rather than deciding on the protocol selection or focusing on implementation details and syntax [31, pp. 86, 109].

A.1.1. Concepts

Derived from several other pre-existing architectures [31, p. 76], REST is best explained as a combination of the following architectural styles and constraints:

Client-server

The principle of separation of concerns allows components to evolve independently. By splitting the user interface engine concerns (*client*) from data storage concerns (*server*), portability and scalability improve [31, p. 78].

Stateless

Each interaction (*request*) between client and server must contain enough information to be processed in isolation (*self-descriptiveness*) [31, pp. 78–79]. In turn, this enables parallel processing and allows intermediaries (*proxy*) to view and understand a request without accessing server context (*session state*). Server-side scalability also improves as no resources have to be dedicated to storing state in-between requests [31, pp. 79, 93]. By nature, this has the design trade-off of increasing repetitive-data and thus, decreasing network performance.

Cache

Data returned by a server (*response*) may be implicitly or explicitly labeled as cacheable or non-cacheable. In some cases, the cache may entirely eliminate interactions, again improving efficiency, scalability, and user-perceived performance, though caution must be taken since reliability issues may arise through stale data [31, pp. 79–80].

Uniform interface

While a standardized component interaction interface simplifies the overall system architecture, it disregards the efficiency improvements that can be gained from an application-level decision of such [31, pp. 81–82]. The exact communication constraints are discussed in the next section.

A.1.2. Component Interaction

The key abstraction in REST is a *resource* which refers to any kind of information that can be named, such as a [31, p. 88]:

- document or image
- temporal service (e.g. the current weather)
- collection of resources
- non-virtual object (e.g. a student)

A resource's value may be static or variable (e.g. document vs. current weather), though in any case the semantics, i.e. what distinguishes one resource from another, must remain the same [31, p. 89].

From here on, Fielding and Taylor explains that REST components perform actions on a resource by exchanging stateless messages composed of [31, pp. 90–91]:

Resource identifier

Unique identifier used to address that particular resource.

Representation

Depending on the *control data*, the representation may capture the current or intended state of a requested resource, value of a different resource (e.g. user input) or an error condition.

Representation metadata

Describes the supplied representation.

Resource metadata

Information about the resource that is not specific to the supplied representation.

Control data

Defines the message's purpose such as the action requested or meaning of the response. Can also be used to parameterize requests or override default behavior.

It shall be noted that a message originating from a client component may include fields different from those delivered by a server component in response [31, pp. 93–94]. This discrepancy is displayed in Table A.1.

Field	Request Message	Response Message
Resource Identifier	x	-
Representation	optional	optional
Resource Metadata	-	optional
Control Data	x	x

Table A.1.: In- and out-parameters of a REST component interaction

The final constraint to component interaction is a concept called Hypermedia as the Engine of Application State [31, p. 82]. Even though Fielding and Taylor's dissertation did not fully elaborate on this aspect, he re-emphasized its importance in the coming years and would not accept an API's *RESTful* labelling if absent [30]. In essence, an application's control state must be included in the resource representation returned by a server [31, p. 102], thereby explicitly stating the actions a client may perform on that resource at that specific point in time. Of course, the list of permitted actions could change in response to any actions that were taken previously.

A.2. Continuous Integration and Deployment

Agile software development welcomes changing requirements, even late in the development. This, combined with the proclaimed goal of wanting to satisfy customers through early and continuous delivery of software has lead organizations to adopt practices known as Continuous Integration and Continuous Deployment (CI/CD) [9] [64, p. 22] [72, p. 78]. This section explains each practice, as well as the value it adds to the software development lifecycle.

A.2.1. Continuous Integration

Fowler, a well-known software developer, public speaker and co-author of the *Manifesto for Agile Software Development*, defines Continuous Integration (CI) as a software development practice where members of a team integrate their work frequently, and at least daily. Each such integration (*build*) should be verified by an automated compile and test suite. Only if all of these steps (*pipeline*) succeed, can the overall build considered to be good, with the goal being that integration errors are detected as quickly as possible [32, pp. 1, 3]. Early detection allows developers to [32, pp. 7, 11–12]:

- build off a shared stable base [59]
- predict how long the integration will take
- resolve issues more easily since the changes are recent and few
- prevent cumulative failures (where one bug appears as the result of another)

However, the degree of these benefits is directly tied to the depth of the test suite and the degree of similarity between the environment in which the results were generated and the one to which the software will ultimately be deployed. Every difference introduces the chance that what happens during a test will not happen in production [32, pp. 9, 12].

Although CI requires no particular tooling, many organizations leverage a so-called CI server to monitor code repositories for changes. With each commit, the server will checkout the source code, initiate a build and publicly display the integration status. It is exactly this automatism that differentiates CI from traditional builds which are performed on a timed schedule. The latter will, by definition, always delay detection of errors [32, pp. 7, 10].

Lastly, Fowler emphasizes that build pipelines must balance the breadth of bug finding techniques (e.g. static code analysis) with the need for speed, i.e. how long it will take to run a build [60]. More in-depth tests may, for example, be moved to a secondary test suite that is not run on every commit. Builds could also be configured to only run against modified components [32, pp. 5, 8].

59 Meyer draws a comparison to Toyota's factory floor. Here, every worker can halt the production line if something breaks or holds them up. Failed integrations should echo a similar behavior and encourage developers to resolve issues promptly as a favor to others [52, p. 15].

60 CI originated as one of the twelve original practices of Extreme Programming (XP). Another practice of XP advocates keeping build times under ten minutes [32, pp. 2, 8].

A.2.2. Continuous Deployment

Whereas CI purely focuses on the integration of changes, Continuous Deployment (CD) extends the practice by automatically deploying newly integrated changes to production [42, p. 64] [64, p. 21]. Consequently, deployment processes are no longer manual, nor involve human approval, but instead happen through repeatedly tested automation. This removes a major source of error, giving developers one less reason to stress on release day [61] [72, pp. 79–80] [18, p. 53]. With regard to other perceived benefits, Leppänen et al. have surveyed 15 companies across various domains and sizes and found that CD [42, pp. 66–67]:

- improves productivity and customer satisfaction
- reinforces developers' sense of accomplishment
- enables stakeholders to stay informed
- prevents a disconnect between the development and operations teams

Admittedly, most organizations will, however, settle on a less continuous process for reasons such as industry regulations (e.g. automotive software), distribution channels (e.g. review process of application store) or pure customer preference [42, pp. 68–69]. As an example, web-based applications will be more suitable for CD than off-the-shelve software because updates can happen in a largely transparent manner to the user [62] [64, p. 22]. In such situations where immediate deployments are not exercised but a company is still ready to reliably release its software on demand, literature refers to the same acronym as Continuous Delivery [18, p. 50].

Irrespective of the level practiced, the pipelines used for these purposes (comp. subsection A.2.1) will likely be extended with more tests (e.g. system and performance), as well as methods to roll back releases on failures [18, pp. 52–53]. CD can of course also be combined with advanced deployment strategies such as using a secondary environment for beta testing (*staging*), switching traffic between two identical environments as testing completes (*blue / green*) or releasing changes off peak or out of user reach to test scalability and performance (*dark launch*) [64, p. 23].

[61] Ironically, developers prefer more frequent releases when given the choice [64, p. 21].

[62] Hewlett-Packard (HP) even practices CD with its printer firmware. The author of this thesis was formerly employed at HP and remembers this fact being shared as if it were a secret family recipe.

B. Concept

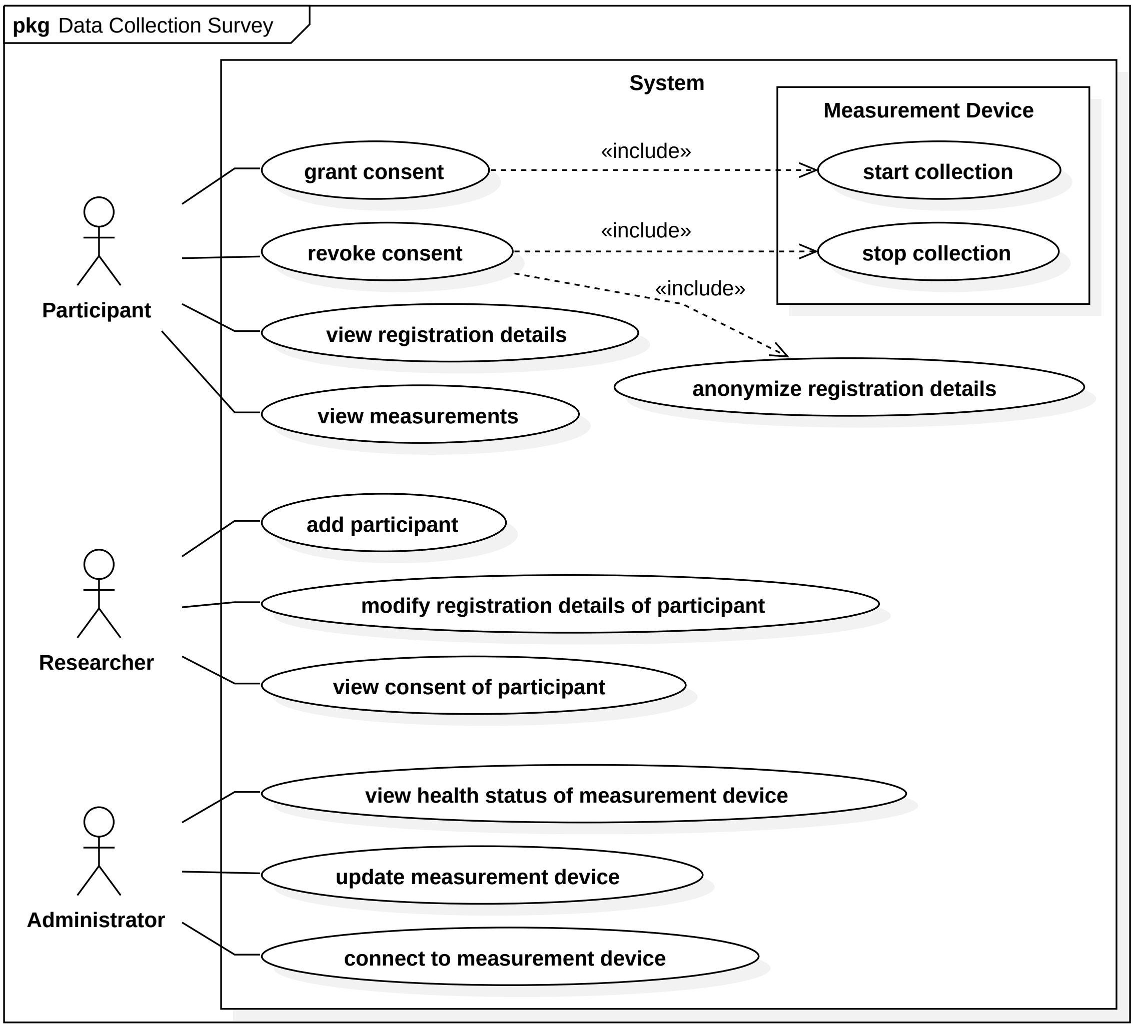

Figure B.1.: System use cases

C. Evaluation

#	Model	MTF (%)	$\overline{\textbf{MC}}$	$\widetilde{\textbf{MC}}$	$\boldsymbol{\sigma}$
1	MT681	100.05	14.97	15.00	0.53
2	EHZ	99.70	15.03	15.05	0.33
3	MT175	99.95	14.99	15.00	0.48
4	EHZ	99.90	15.01	15.02	0.02
5	MT175	103.18	14.52	15.00	2.21
6	MT175	103.23	14.51	15.00	2.22
7	EHZ	100.15	14.96	15.02	0.79
8	EHZ	100.60	14.89	15.02	1.02
9	EHZ	103.57	14.45	15.00	2.34
10	MT175	101.98	14.69	15.00	1.82
		101.23	**14.80**	**15.01**	**1.18**

Table C.1.: Measurement target fulfillment (MTF) and measurement cycle (MC) statistics over past three weeks of operations, grouped by measurement device [63]

[63] The mean across all statistics can be taken in this case because the sample size is expected to be the same for each measurement device.

#	**RTF** (%)	$\overline{\mathbf{RC}}$	$\widetilde{\mathbf{RC}}$	$\boldsymbol{\sigma}$	$\overline{\mathbf{MPR}}$	$\widetilde{\mathbf{MPR}}$	$\boldsymbol{\sigma}$
1	100.00	60.00	60.00	0.00	4.00	4.00	0.12
2	100.00	60.00	60.00	0.00	3.99	4.00	0.15
3	100.00	60.00	60.00	0.00	4.00	4.00	0.13
4	100.00	60.00	60.00	0.00	3.99	4.00	0.11
5	100.00	60.00	60.00	0.38	4.12	4.00	0.39
6	100.00	60.00	60.00	0.38	4.13	4.00	0.38
7	100.00	60.00	60.00	0.38	4.01	4.00	0.18
8	100.00	60.00	60.00	0.00	4.02	4.00	0.22
9	100.00	60.00	60.00	0.00	4.14	4.00	0.42
10	100.00	60.00	60.00	0.00	4.08	4.00	0.34
	100.00	**60.00**	**60.00**	**0.11**	**4.05**	**4.00**	**0.24**

Table C.2.: Report target fulfillment (RTF), report cycle (RC) and measurements per report (MPR) statistics over past three weeks of operations, grouped by measurement device [64]

[64] The mean across all statistics can be taken in this case because the sample size is expected to be the same for each measurement device.

D. List of CD Contents

⊢ abstract.pdf ⇒ *Summary in English and German*

⊢ **latex.zip/** ⇒ *LaTeX files for this thesis*
  – exposé.pdf
  – thesis.pdf
  – bibliography.bib
  ⊢ **ads/** ⇒ *Front- and backmatter*
  ⊢ **content/** ⇒ *Main part*
  ⊢ **resources/** ⇒ *Draw.io and StarUML files of figures created for this thesis*

⊢ **specification/** ⇒ *OpenAPI specification of REST APIs*
  – data-api-spec.json
  – data-api-proxy-spec.json

⊢ **code/** ⇒ *Source code of components listed in section 4.2*
  – kubernetes-config.zip
  ⊢ **edge/**
    – data-aggregator-transmitter.zip
    – device-frontend.zip
    – data-api-proxy.zip
  ⊢ **cloud/**
    – data-api-registration-frontend.zip
    – status-frontend.zip

⊢ **evaluation/**
  – report.py ⇒ *Script used to calculate system availability*
  – output.txt