Hochschule **RheinMain**

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

Master Thesis

for the

Master of Science

in Computer Science

at RheinMain University of Applied Sciences

by

Niklas Sauer

January 11th, 2022

Project Duration	6 Months
Student ID	1209772
First Reviewer	Prof. Dr. Heinz Werntges
Second Reviewer	Prof. Dr. Robert Kaiser
Supervisor	Johannes Kaeppel

Abstract

Contents

Acronyms

EBP	EnergieBroker Platform
EB	EnergieBroker
EEG	Renewable Energy Sources Act (German: *Erneuerbare-Energien-Gesetz*)
FIT	Feed-In Tariff
EV	Electric Vehicle
HATEOAS	Hypermedia as the Engine of Application State
PVS	Photovoltaic System
REST	Representational State Transfer
SOA	Service-Oriented Architecture
CPU	Central Processing Unit
API	Application Programming Interface
IP	Internet Protocol
SRP	Single Responsibility Principle
DDD	Domain-Driven Design
ACID	Atomicity Consistency Isolation Durability
NIST	National Institute of Standards and Technology
VM	Virtual Machine
OS	Operating System
IaaS	Infrastructure as a Service
PaaS	Platform as a Service
MAC	Mandatory Access Control
DoS	Denial of Service
SELinux	Security-Enhanced Linux
NSA	National Security Agency
VCS	Version Control System
OCI	Open Container Initiative
MAPE-K	Monitor-Analyze-Plan-Execute over a shared Knowledge
SLA	Service-Level Agreement
CI	Continuous Integration
CD	Continuous Deployment
CI/CD	Continuous Integration and Continuous Deployment
XP	Extreme Programming
ROI	Return on Investment
GDPR	General Data Protection Regulation

MaStR	Marktstammdatenregister
QoS	Quality of Service
IoT	Internet of Things
AI	Artificial Intelligence

List of Figures

List of Tables

Listings

1. Introduction

1.1. Motivation

Germany's Renewable Energy Sources Act (German: *Erneuerbare-Energien-Gesetz*) (EEG) is a powerful tool used to push ahead the energy transition and thus, parts of the contributions Germany can make to slow down global climate change. Disappointingly though, one of the major drivers 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, starting 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 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 [36] 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 beforehand. 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
- production-grade implementation
- deployment and maintenance plan design
- field test

1.3. Thesis Overview

2. Theoretical Framework

2.1. Microservice Architecture

Popularized by companies like Amazon, Netflix, Uber, LinkedIn and SoundCloud, the microservice architecture has emerged as a pattern to avoid the problems of conventional monolithic designs [35, p. 847] [41, 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 the newly introduced challenges.

2.1.1. The Monolith Problem

A monolith is a software application whose modules cannot be executed independently [14, p. 1]. Hence, a monolith is characterized by requiring to be deployed as a united solution [13, 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 [14, p. 2]. The latter is especially true for beginners, slowing down their productivity [13, 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 [20, 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 [14, p. 2]. In cases where this is 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 [13, p. 24] [14, 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 [14, p. 2]. Similarly, failure of one service leads to downtime across all services [39, p. 970]. The deployment can therefore be viewed as a single point of failure [41, 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 [14, 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 [35, p. 850] [14, p. 2]. Less popular services consume unnecessary (idle) amounts of resources [41, p. 584]. Setting appropriate scaling thresholds also becomes more challenging because different components may have different resource requirements [13, 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 [20, pp. 34, 36]. Villamizar et al. add that monoliths are faster to set up [41, p. 589]. Empirically, most organizations start with something big and slowly transition to a decomposed architecture when scaling problems arise [38, p. 113] [41, 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 [20, 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 [22, p. 43] [39, 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] [37, p. 56] [40, p. 176] [10, 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 [10, p. 31].

2.1.3. Decomposition Techniques

Spitting an application into services should happen along the lines of related processes that can be carried out in isolation. It is not about arbitrarily distributing features across services [37, p. 61]. As in traditional software engineering, the term cohesiveness is used to indicate that a service implements only functionalities strongly related to the concern that it is meant to model [14, 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 [27, p. 36] [38, 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 that covers a specific use case, whereas the noun-based approach creates sets of functions responsible for all operations related to a particular entity [27, 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. This domain consists of multiple subdomains (e.g. product catalog, order management). Each subdomain is represented by a microservice [3, p. 3]. Leads to a small amount of services.

add citation

Irrespective of the technique chosen, the overall goal should be to minimize later interface changes, i.e. to establish proper service contracts [13, 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 [27, p. 38] [38, p. 114]. Applied to microservice-based applications, this means that developers need to deal with the challenges innate to distributed systems [13, p. 24] [41, 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 [27, 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 [22, 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.

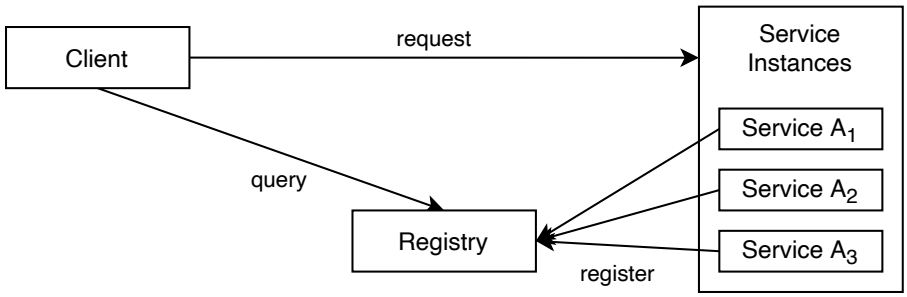


Figure 2.1.: Client-side service discovery [27, p. 37]

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.

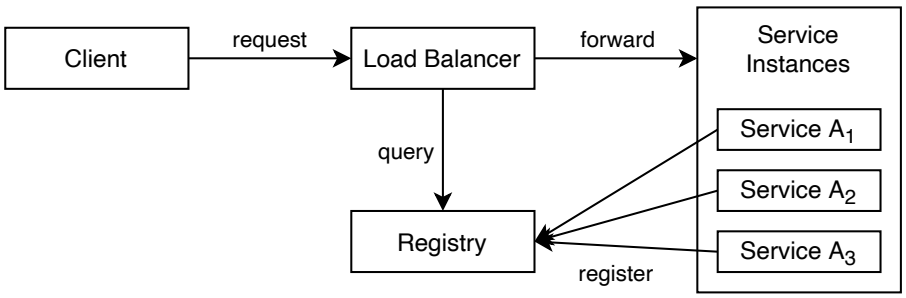


Figure 2.2.: Server-side service discovery [27, p. 37]

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

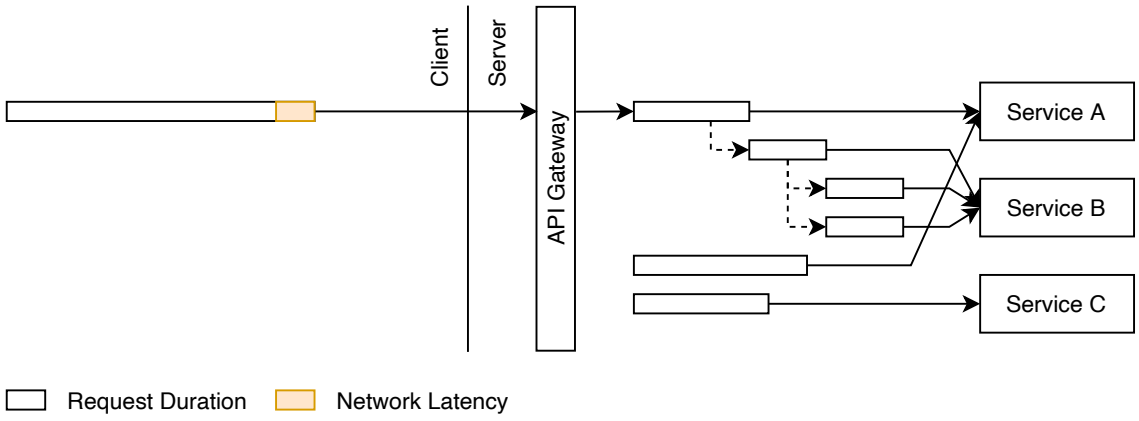
[4] In the event of a network partition, a standard recovery process would attempt to restart a service in the healthy partition, thus leading to a new network location for this service.

2.1.5. API Gateway Pattern

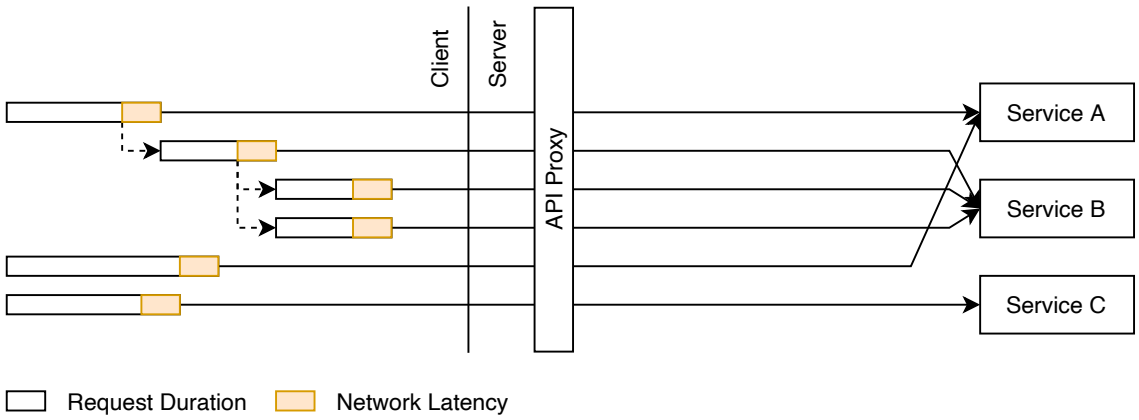
Depending on the decomposition technique chosen (see subsection 2.1.3), microservices might provide very fine-grained APIs. In turn, this means that clients may need to interact with multiple different services to carry out a high-level business process. To hide this complexity from clients and ensure 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 [41, 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 [10, 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 [20, p. 41] [27, p. 37].

add cita tion



(a)



(b)

Figure 2.3.: Service consumption model with (a) and without (b) API gateway [12]

[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 [10, p. 32].

It shall be noted that gateways incur a performance penalty because they introduce an additional network hop [27, 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 now already originate from the target network (comp. Figure 2.3) [12]. In both cases, the performance degradation will heavily depend on the system's interconnectedness [14, 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 [27, p. 36] [37, p. 59]. Messina et al. discern between three levels of pattern conformity [27, 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] [10, 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 whether microservices should be considered as a subset or superset of SOAs or whether it constitutes a new, distinct idea [41, pp. 584–585] [10, p. 30]. The systematic mapping study conducted by Cerny, Donahoo, and Trnka in [10] spends a great deal on contrasting the two architectures. A short summary is given in the following.

[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 [10, p. 30].

In both approaches, services cooperate to provide functionality for the overall system. However, the path to achieving this goal is different. 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).

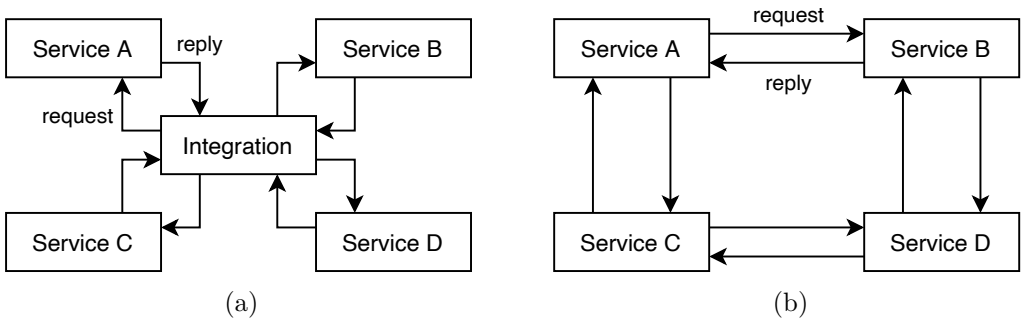


Figure 2.4.: Service orchestration (a) and choreography (b) [10, p. 30]

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 modal. The danger being 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, albeit obvious, the biggest drawback of SOAs and their centralized orchestration model must be stated. Having a centralized business process integrate all services again entails having a single point of failure.

2.2. OS-level Virtualization

Virtualization describes the act of creating a virtual version of something [9, p. 2]. In the context of computing, it specifically refers to hardware resources such as CPU, memory, storage or network devices to create complete virtual instances of computer

systems [34, 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] [43, p. 233] [15, p. 2]. This is achieved by using fewer physical servers to host the same number of applications. But without an isolation layer there are no guarantees that an application from one user will not interfere with that of another [43, p. 233]. Isolation and multi-tenancy are exactly those traits promised by virtualization technologies [34, p. 21]. Coupled with a software layer to provision resources on demand, virtualization yields the elastic multi-tenant model embodied in cloud computing [21, p. 203] [5, p. 81] [30, p. 24].

update footnote figure a citation

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 [26, p. 2] [15, p. 1] [29, pp. 386–387]:

Type 1 (bare-metal)
Operates directly on top of the host's hardware, not requiring a 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 [29, pp. 386, 392].

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 that establish an abstract and protected view on the OS, making two

[7] About 10% of the world's energy consumption stems from data center operations [33, p. 1]. 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 [43, p. 233].
[9] Containers are known as jails or zones in the FreeBSD and Solaris OSes, respectively [15, p. 2].

containers unaware of each other or any of the other processes running on the host [26, p. 2] [15, pp. 1–2].

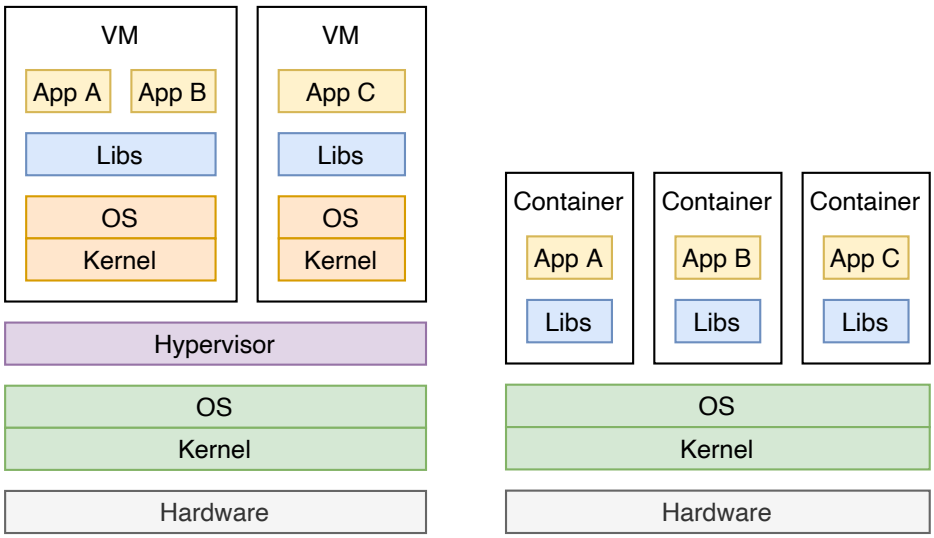


Figure 2.5.: Hardware (a) and OS-level (b) virtualization [29, p. 387]

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 about hardware allocation and management, while containers are tools for delivering software. He further compares them to the concepts of IaaS and PaaS, respectively [30, p. 24]. Eder and Merkel et al. see the potential for the two technologies to complement each other since the resource footprint of containers is minimal and its security profile is still slightly worse than that of VMs [10] [15, p. 6] [26, p. 2].

add cita
tion

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 [29, p. 386] [15, 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 rebuild first [15, p. 3]. Moreover, sharing a kernel allows container-based solutions to achieve a higher density of virtualized instances when compared to hypervisor-based solutions [29, p. 386] [21, 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,

[10] Kernels cannot prevent interference in low-level resources such as the CPU's L3 cache or memory bandwidth [7, 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 runs software in the form of containers.

whereas the OS of a VM needs to be fully booted and shut down [26, p. 2] [15, 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 [26, 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:

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 [15, p. 3].

update citation

Namespaces

Namespaces provide one or more processes with private and restricted views towards certain global system resources [29, 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 [15, 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 [26, p. 1].

update citation

Control groups

Control groups, usually referred to as cgroups, provide resource accounting and limiting for a set of processes [29, p. 387] [26, p. 1]. Even though they are not mandatory for process isolation, cgroups can ensure that one container cannot starve another (e.g. in a DoS attack), as well as to help realize cost-accounting multi-tenant environments [15, p. 4].

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

[11] Some people trace the inspiration for containers back to `chroot()`, which was originally introduced with Unix 7 in 1979 [5, p. 82].

AppArmor and SELinux [12]. In the context of containers, MAC is useful as that it can restrict the process to its minimal requirements, thus further reducing the attack surface against the host and other containers [15, p. 4].

add citation

2.2.4. Docker

The concept of container-based virtualization has existed long before Docker came into view in 2013 [13]. Yet, it was Docker, Inc. who made containers popular by creating a toolkit that is greater than the sum of its parts [26, p. 1]. The following presents three areas of developer concern in which Docker shines and through which it has become a synonym for containers [14]:

add citation

add citation

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 [6, p. 72]. Docker addresses this challenge with the concept of container images. Such an image bundles an application with all of its dependencies up to, but excluding, the kernel [26, 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 along a repository and evolve with the application [6, 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 [15] [26, p. 1].

add to glossary

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

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

[13] 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 [5, p. 82]. In Linux, kernel namespaces were discussed as early as 2006 [9, p. 1].

[14] Boettiger makes a good case how Docker can also help to make research reproducible and more easily extendable [6, p. 71].

[15] At runtime, all image layers will be merged into a single representation of the filesystem. This is known as a union mount [30, p. 26].

"develop once, deploy everywhere" [16] [21, 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 in the same way, if at all, across platforms due to differences in, for example, networking or storage [6, pp. 74–75]. This problem is known as runtime consistency [21, p. 203]. To enable consistent behavior, Docker ships with a container runtime that abstracts many of the platform peculiarities [17] [6, 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 unheard of before. 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 [15, pp. 6–7].

Driven by the risk of lack of interoperability, Docker and other leaders in the container industry established the Open Container Initiative (OCI) in 2015 to standardize aforementioned container image formats and runtimes [34, p. 23].

update citation

2.3. Container Orchestration Platform

Given the operational benefits of microservice-based applications (see section 2.1), as well containers 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*) [22, p. 44] [31, p. 1]. This section presents the immediate benefits to dealing with container workloads, lists the requirements to any such container orchestration platform and tries to convey its way of working by laying out a reference architecture.

[16] 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 [21, p. 203] [15, p. 2].

[17] 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 [26, p. 5].

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 [18], Burns et al. make a strong case how containerization transforms data centers from being machine oriented to being application oriented [7, 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. On the other hand, 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 [22, p. 44]. It is not only concerned with the runtime but also aids in the deploy and maintain phases of an application's lifecycle [8, p. 225]. For the purposes of automation, all cluster operations should be exposed and accessible via an API [8, p. 224] [30, p. 29]. Throughout literature, the following set of capabilities is generally expected 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 like node affinity or resource requests [8, 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 to answer requests. Similarly, periodic health checks should be used to determine whether a container needs to be restarted or rescheduled onto a different host [1, p. 2].

Scaling & Availability

To cope with increases in traffic, horizontal scaling of applications is critical [1,

[18] Google has contributed much of the Linux kernel's process isolation code [7, p. 50].

p. 1]. In this context, an application refers to one or multiple containers that are managed as a single entity [22, p. 44] [31, 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 [8, pp. 225–226] [1, 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 [22, p. 45]. Of this, redundancy, i.e. replication across physically separated hosts, remains the most important feature, though self-healing by detecting failures is just as key. A container platform must support and implement both techniques [40, p. 176] [7, 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 white-box tracing of requests [22, 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 Mandatory Access Control (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 [8, 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 SLAs.

add cita
tion

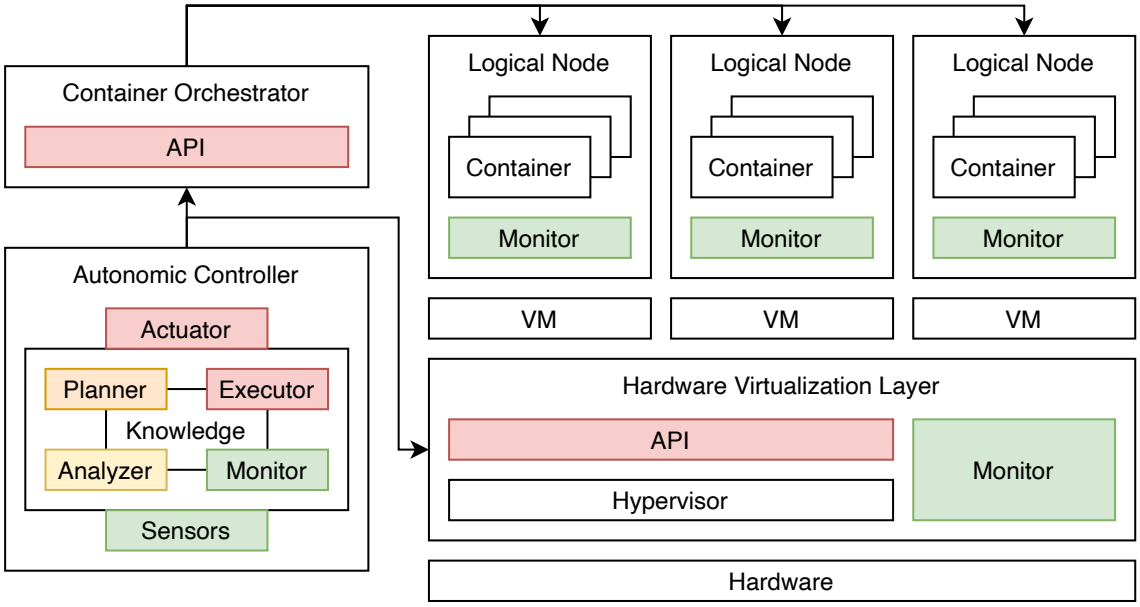


Figure 2.6.: Reference architecture for container orchestration platforms [8, p. 226]

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 following:

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 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

(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 a phototransistor optocoupler (*measurement probe*) that is capable of converting the light signal (*measurement interface*) of an electricity or generation meter into a serial bitstream.

With respect to Subtask (A), the survey intends to install one or multiple hardware devices (*measurement devices*) in each participating household. These devices will be equipped with one or multiple measurement probes. The number of measurement devices and probes will vary depending on the number of meters to be read out and whether they are physically separated from each other, i.e. whether 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 be actively participating in the EBP (comp. [36]). This rules out any solutions in which the measurements are stored on device and only processed when the survey ends and thus, devices have been returned. Instead, measurements shall be transferred to a centralized data store on an hourly schedule, at minimum (*report cycle*).

add pag number

3.1.2. Registration

Even though the survey hopes to cover a diverse set of households, it currently does not intend for the public to sign up for participation at free will. Instead, the survey will follow an invite-only model. Regardless of the criteria based on which households are selected (comp. section 1.2), participants will have to fill out a registration form that covers the following subjects:

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 adults work in your household?
- Does your household have an EV 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?

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*) for each of the study's subjects and couple that to the lifecycle of each measurement device, as well as the data processing capabilities of the research group itself. Three states of participation shall be distinguished for any given household:

Consent not 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.

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 on user request. 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 legal advice sought after. It plans to: add citation

(A) remove the contact details

(B) anonymize the location

(C) anonymize the Photovoltaic Systems

(D) keep the metadata and meters

Obviously, (A) is the most straightforward measure 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. Therefore, (B) is required. Similarly, (C) is necessary because the specifications of a PVS could be matched against a public registry (e.g. MaStR in Germany [19]). All measures combined allow for (D). The following descriptions shall explain the anonymization strategies of (B) and (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

[19] 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. Many of its contents are publicly viewable at `https://www.marktstammdatenregister.de`

household. To be specific, the household's location shall be replaced with a 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:

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

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 study, 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 43.

Participant

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

FR-02 As a participant, I can revoke my consent, so that the measurement devices installed in my household stop to collect 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 [25, 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 [24, 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 Edge Cloud Computing Problem

Driven by applications with strict time-bound requirements such as the Internet of Things (IoT), Artificial Intelligence (AI) 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 [44, p. 373] [2, 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*) [20] [19, 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 [2, 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* [21]). 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).

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

[21] 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.

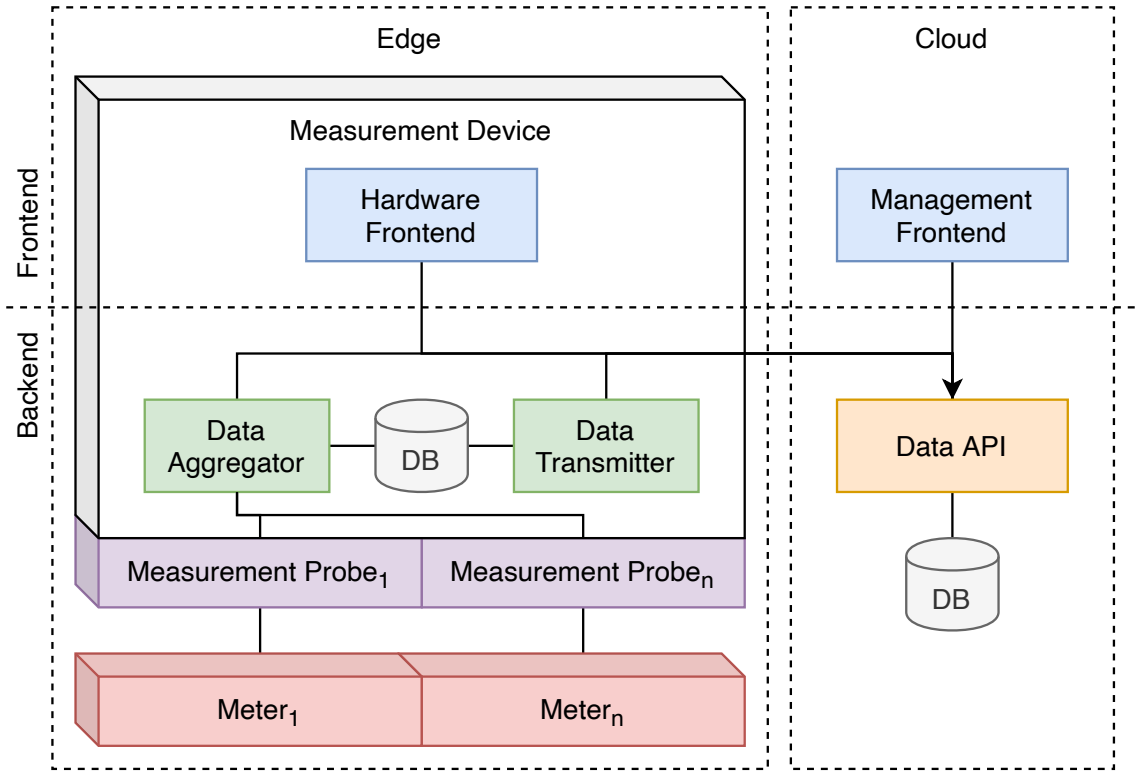


Figure 4.1.: High-level component design

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 min) 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 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 18

Data Transmitter

The data transmitter periodically (every 60 min, 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 either be deleted or marked as reported in order to avoid duplicate uploads. 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 aggregator exclusively interacts with the database in an append-only mode.

Realizes: Subtask (B) on page 18

Hardware Frontend

The hardware 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. All of these details and actions are retrieved from and performed through the cloud.

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

4.2.2. Cloud

The set of microservices running in the cloud can be described as following:

Data API

The data API acts as the single source of truth for each participant's registration details, consent and measurements. It offers a network interface to retrieve and modify these details, grant and revoke the 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 the health status of each measurement device. It may be argued that this component itself has a monolithic character due to the breadth of services it offers. Yet, the number of services is not crucial to this opinion, but rather the degree of their relatedness (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

Management Frontend

The management frontend provides administrators with a web-based user interface for viewing the health status of each measurement device. Specifically, it will state to which participant a measurement device belongs, indicate whether that participant has granted or revoked his consent, and list more technical details such as the device's version and date and time of activity. All of these details are retrieved from the data API.

Realizes: FR-09

4.3. Component Specification

4.4. Component Mapping

4.5. Deployment Model

4.6. Maintenance and Support Plan

5. Conclusion

6. Summary

7. Outlook

Bibliography

[1] 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)*. IEEE. 2019, pp. 1–6.

[2] Muhammad Alam et al. “Orchestration of microservices for iot using docker and edge computing”. In: *IEEE Communications Magazine* 56.9 (2018), pp. 118–123.

[3] 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.

[4] Kent Beck et al. “Manifesto for agile software development”. In: (2001).

[5] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[6] Carl Boettiger. “An introduction to Docker for reproducible research”. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 71–79.

[7] Brendan Burns et al. “Borg, omega, and kubernetes”. In: *Communications of the ACM* 59.5 (2016), pp. 50–57.

[8] Emiliano Casalicchio. “Container orchestration: A survey”. In: *Systems Modeling: Methodologies and Tools* (2019), pp. 221–235.

[9] Antonio Celesti et al. “Exploring container virtualization in IoT clouds”. In: *2016 IEEE international conference on Smart Computing (SMARTCOMP)*. IEEE. 2016, pp. 1–6.

[10] Tomas Cerny, Michael J Donahoo, and Michal Trnka. “Contextual understanding of microservice architecture: current and future directions”. In: *ACM SIGAPP Applied Computing Review* 17.4 (2018), pp. 29–45.

[11] Lianping Chen. “Continuous delivery: Huge benefits, but challenges too”. In: *IEEE software* 32.2 (2015), pp. 50–54.

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

[13] Namiot Dmitry and Sneps-Sneppe Manfred. “On micro-services architecture”. In: *International Journal of Open Information Technologies* 2.9 (2014).

[14] Nicola Dragoni et al. “Microservices: yesterday, today, and tomorrow”. In: *Present and ulterior software engineering* (2017), pp. 195–216.

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

[16] Roy T Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. University of California, Irvine Irvine, 2000.

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

[18] Martin Fowler. *Continuous integration*. 2006.

[19] Saiful Hoque et al. “Towards container orchestration in fog computing infrastructures”. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. 2017, pp. 294–299.

[20] Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. “Challenges when moving from monolith to microservice architecture”. In: *International Conference on Web Engineering*. Springer. 2017, pp. 32–47.

[21] 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)*. IEEE. 2016, pp. 202–211.

[22] Asif Khan. “Key characteristics of a container orchestration platform to enable a modern application”. In: *IEEE cloud Computing* 4.5 (2017), pp. 42–48.

[23] Marko Leppänen et al. “The highways and country roads to continuous deployment”. In: *Ieee software* 32.2 (2015), pp. 64–72.

[24] 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*. WWW Alt. '04. New York, New York, USA: ACM Press, 2004, p. 66. ISBN: 1581139128.

[25] 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*. iiWAS '16. New York, NY, USA: ACM, 2016, pp. 430–432. ISBN: 9781450348072.

[26] Dirk Merkel et al. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[27] 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.

[28] Mathias Meyer. “Continuous integration and its tools”. In: *IEEE software* 31.3 (2014), pp. 14–16.

[29] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. lightweight virtualization: a performance comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393.

[30] Claus Pahl. "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[31] Claus Pahl et al. "Cloud container technologies: a state-of-the-art review". In: *IEEE Transactions on Cloud Computing* 7.3 (2017), pp. 677–692.

[32] Tony Savor et al. "Continuous deployment at Facebook and OANDA". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE. 2016, pp. 21–30.

[33] Mathijs Jeroen Scheepers. "Virtualization and containerization of application infrastructure: A comparison". In: *21st twente student conference on IT*. Vol. 21. 2014.

[34] Vitor Goncalves da Silva, Marite Kirikova, and Gundars Alksnis. "Containers for Virtualization: An Overview." In: *Appl. Comput. Syst.* 23.1 (2018), pp. 21–27.

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

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

[37] Davide Taibi and Valentina Lenarduzzi. "On the definition of microservice bad smells". In: *IEEE software* 35.3 (2018), pp. 56–62.

[38] Johannes Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[39] 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)*. IEEE. 2018, pp. 970–973.

[40] 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)*. IEEE. 2019, pp. 176–185.

[41] 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)*. IEEE. 2015, pp. 583–590.

[42] Manish Virmani. "Understanding DevOps & bridging the gap from continuous integration to continuous delivery". In: *Fifth international conference on the innovative computing technology (intech 2015)*. IEEE. 2015, pp. 78–82.

[43] 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*. IEEE. 2013, pp. 233–240.

[44] Ying Xiong et al. “Extend cloud to edge with KubeEdge”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2018, pp. 373–377.

Glossary

cloud computing

National Institute of Standards and Technology (NIST) internationally accepted definition of cloud computing calls for resource pooling where provider's computing resources are pooled to serve multiple consumers using multi-tenant model with different physical and virtual resources dynamically assigned and reassigned according to consumer demand [5, p. 81].

node affinity

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

add citation

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 [27, p. 36].

white-box tracing

add explanation

.

Appendices

A. Theoretical Framework

A.1. Representational State Transfer (REST)

Coined and theorized by Fielding, 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 [16, 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 [16, p. xvi], rather than deciding on the protocol selection or focusing on implementation details and syntax [16, pp. 86, 109].

A.1.1. Concepts

Derived from several other pre-existing architectures [16, 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 [16, p. 78].

Stateless

Each interaction (*request*) between client and server must contain enough information to be processed in isolation (*self-descriptiveness*) [16, 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 store state in-between requests [16, 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 [16, 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 [16, pp. 81–82]. The exact communication constraints are discussed in subsection A.1.2.

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 [16, p. 88]:

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

A resource's value may be static or variable (document vs. current weather), though in any case the semantics, i.e. what distinguishes one resource from another, must remain the same [16, p. 89].

From here on, Fielding explain that REST components perform actions on a resource by exchanging stateless messages composed of [16, 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 different fields than that delivered by a server component in response [16, 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 (HATEOAS) [16, p. 82]. Even though Fielding'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 [17]. In essence, an application's control state must be included in the resource representation returned by a server [16, 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 previous one taken.

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) [4] [32, p. 22] [42, 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. The goal being that integration errors are detected as quickly as possible [18, pp. 1, 3]. Early detection allows developers to [18, pp. 7, 11–12]:

- build off a shared stable base [22]
- predict how long the integration will take
- resolve issues more easily since the changes are recent and few
- prevent cumulative failures (where one bug shows as the result of another)

However, the degree of these benefits is directly tied to the depth of the test suite and similarity between the environment in which the results were generated and that where the software will ultimately be deployed to. Every difference introduces a chance that what happens under a test will not happen in production [18, 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 [18, 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 [23]. 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 [18, pp. 5, 8].

add citation

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 [23, p. 64] [32, p. 21]. Consequently, deployment processes are no longer

[22] 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 [28, p. 15].

[23] CI originated as one of the twelve original practices of Extreme Programming (XP). Another practice of XP advocates for keeping build times under ten minutes [18, pp. 2, 8].

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 [24] [42, pp. 79–80] [11, p. 53]. With regard to other perceived benefits, Leppänen et al. have surveyed 15 companies across various domains and sizes and found that CD [23, 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 [23, pp. 68–69]. As an example, web-based applications will be more suitable for CD than off-the-shelve software because updates can happen largely transparent to the user [25] [32, p. 22]. In such situations where immediate deployments not are exercised but a company is still ready to reliably release its software on demand, literature refers to the same acronym as Continuous Delivery [11, 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 [11, 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*) [32, p. 23].

[24] Ironically, developers prefer more frequent releases when given the choice [32, p. 21].

[25] 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

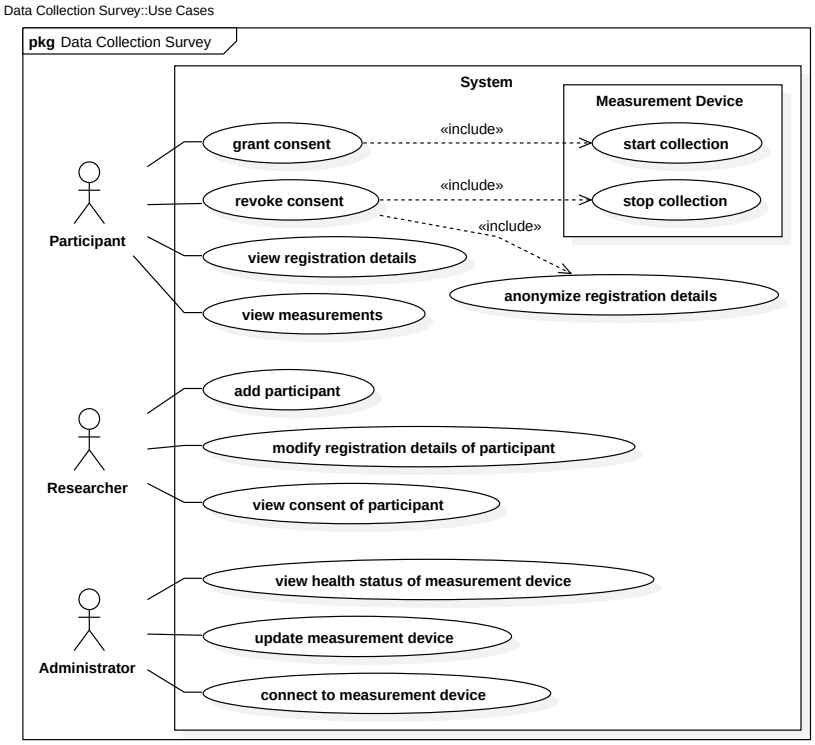


Figure B.1.: System use cases