Architectural Quality Attributes for the Microservices of CaRE

Thomas Schirgi

April 2021

Contents

List of Tables

List of Figures

Nomenclature

CD	Continuous Delivery
CI	Continuous Integration
ESB	Enterprise Service Bus
IoT	Internet of Things
MSA	Microservice Architecture
ORM	Object Relational Mapper
OS	Operating System
QA	Quality Attribute
SLA	Service Level Agreement
SOA	Service Oriented Architecture
TLS	Transport Layer Security
VM	Virtual Machine

1 CaRE Project

With products and individual items that are becoming more and more complex, it is logistics wise (travel restrictions during Covid,...) and for cost reasons sometimes not possible to dispatch specialists to site. As a result, maintenance personnel often has no previous experience with a product to be serviced. Nevertheless knowledge is essential for most maintenance and repair works.

The CaRE (Custom Assistance for Remote Employees) is an Siemens Project, where it is researched how to assist employees during their work at large power transformers. Those transformers can be seen as singletons due to their various different types (like Generator Transformers, Phase Shifters, Shunt Reactors,....). Since several subsystems are required in order to provide the necessary information to the users, the backend system was designed as a Microservice Architecture. With this approach the system should remain scaleable and maintainable in many dimensions.

2 Microservices

Since Microservices (MSA) were first mentioned by Fowler and Lewis in 2014 [Fowler, 2020] this type of architecture seems to be a paradigm change in software development. This can be seen in the Hype Cycle for Application and Integration Infrastructure published by Gartner in 2019 [Permikangas, 2020]. Many of IT Companies - like the FANG companies - deliver their services based on Microservices. In comparison to that a widely used architecture is a monolith - were all functionality of an application has to be deployed together. (see [Newman, 2019]).

2.1 Differences between Monolithic, Microservices and Service Oriented Architecture

Basically the main features of Service Oriented Architecture (SOA) and Microservices are pretty the same. Both are relying on (hybrid) cloud environments to provide and execute applications. Both are splitting huge and complex application into smaller parts and can combine one or more services which are needed in order to create and use applications. Microservices are basically defined as "independent deployable" modules and can be seen as an extension to service-oriented architecture (see [Zimmermann, 2016])

In the table below the main differences are stated:

	Microservices	SOA	Monolitic
Architecture	Host Services, operating individually	Provide Resources, commonly used by services	Single Architecture with Single Development Stack
Common use of Components	Components not shared but Shared Libraries (connectors, etc.)	Components shared	Components in terms of shared libraries
Granularity	Differentiated Services	Bigger, strong modular Services	Single Unit
Data Storage	Own Data Storage	Commonly used Data Storage	Mostly one data Storage
Governance	Collaboration between teams necessary	common Governance Protocols, Team overarching	Governance not necessary
Size and Scope	Better for small, web-based Applications	Better for big integrations	Size varies, Scope dependent
Communication	API Layer	Enterprise Service Bus	Typically: Presentation Layer - Business Logic Layer - Data Access Layer
Coupling and Cohesion	Limited Context for the Coupling	Resources are shared	No - Resources are shared
Remote-Services	REST or Java Message Service	SOAP or AMQP	Independent - eventually proprietary
Deployment	Fast and Easy Deployment	Lack of Flexibility at Deployment	Lack of Flexibility and Deployment

Table 1: Differences between Microservices, SOA and Monolithics (see [Talend.com, 2020, Katkoori, 2019])

In Figure 1 the difference is shown in a graphical manner. It can be seen that the main difference between MSA and SOA and Monolithic Architecture is, that it Microservices are typically conducted via multiple databases within MSA.

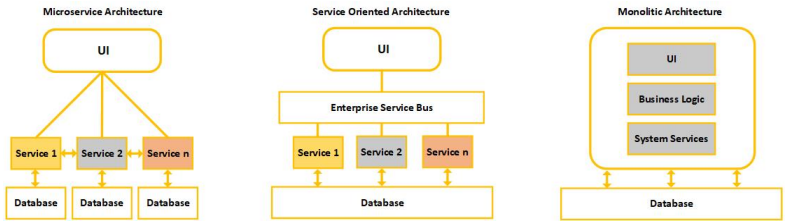

Figure 1: Differences between Microservices, SOA and Monolithics, for typical, respective Architectures

The definition and the differences against service-oriented architecture leads to several advantages and disadvantages of Microservices. (see [Sam, 2015, Eberhard, 2018]).

2.2 Advantages and Disadvantages of Microservices

Microservices have both a lot of advantages and disadvantages. In the following section some of the most important pros and cons are discussed:

2.2.1 Scalability

Each of the Microservices can be scaled individually. In comparison to monolithic architectures - where multiple instances are hard do implement - it is possible to implement Microservices in a fine-grained structure with less services. Furthermore it is possible to deploy multiple instances of services for load balancing reasons. With this scalability it is also possible to exchange services easy. (see [Sam, 2015, Nemer, 2019, Newman, 2019])

2.2.2 Technology independent

For different purposes different technologies can be used. It is possible to select the most suitable tool for each task. Each Microservice can easily be migrated to other - even new - technologies. These technology independence can led from different database systems, different interfaces up to different programming languages. (see [Sam, 2015, Nemer, 2019])

2.2.3 Resilient

Assuming, that the network is reliable, Microservices are basically more resilient than monolithic approaches. If one service fails, the other services may remain up and running and failures are not cascading so that an app crashes. Even if other Microservice need to compensate the failed service, values can be cached and used in case of failures. (see [Sam, 2015, Nemer, 2019])

2.2.4 Deployment

Microservices also lead to some advantages for development teams and deployment of services itself. With smaller services it is easier for new developers to learn the structure. Each Microservice can be deployed independently, as needed, enabling continuous improvement and faster updates. Specific Microservices can be assigned to specific development teams, which allows them to focus solely on one service or feature - also technology wise. This means teams can work autonomously without worrying what's going on with the rest of the app.(see [Sam, 2015, Nemer, 2019])

2.2.5 Complexity

The communication between services can be complex. With a huge amount of services debugging can be challenging if each service has its own set of logs. Additionally testing might be hard since Microservices might be distributed over machines and not running on one single computer. (see [Nemer, 2019, Newman, 2019, Wittmer, 2019])

2.2.6 Controllability

With a huge amount of Microservices relying on each other, simple changes of APIs are dangerous. These changes need to be backward compatible, otherwise each of the Microservices relying on the changed one needs to be changed as well. (see [Nemer, 2019, Wittmer, 2019])

2.2.7 Costs

If Microservices are running on different hosts, the hosting infrastructure, security and maintenance is more expensive. Furthermore require interacting services a huge amount of remote calls. This will also increase the network latency and processing costs. (see [Newman, 2019, Fowler, 2020])

2.3 Boundaries

Both SOA, Monolithic and Microservices have advantages and disadvantages. Thus Microservices are no patent remedy for each type of application. It has to be determined if a Microservices architecture has more advantages rather than disadvantages for the type of application which needs to be implemented. If using Microservices distributed transactions or the CAP Theorem needs to be minded. Furthermore an appropriate and good architecture has to meet some quality criteria.

3 Software Quality

In the IEC 90003:2018 a software product is defined as a "set of computer programs, procedures and possibility associated documentation and data" ([ISO, 2018,

P. 3]). The source code quality such products is a major part of the entire quality of an application. Issues in the source code of application can lead to problems ranking from exceptions, to application crashes or even to damages at the system. According to [Galin, 2018] it must be distinguished between faults and failures.

- Faults: a software fault is a software error which causes improper functioning of the software. In some software fault cases, the fault is corrected or can be neutralized by following code lines. E.g.: Exception Handler
- Failures: software failures are disrupting the use of a software and is a result of an software fault. E.g.: Software closes

In order to both ensure that there are no faults and failures in an software product, developers need to ensure a proper code quality, but also an architectural quality. In [Alenezi, 2016] the importance of software architecture is summarized in six aspects:

1. Understanding: Software architecture is a mechanism in order to simplify the ability to understand complex and large systems by presenting them in a higher level of abstraction.
2. Reuse: It support reuse of components and frameworks. There are different promoters for reuse, like architectural patterns, domain specific architecture, components, etc.
3. Construction: It provided some sort of a blueprint for development and implementation by showing the major components and dependencies between them.
4. Evolution: Software architecture shows possibilities where a system may evolve in future.
5. Analysis: It can be seen as a mechanism to analyze the entire system. These analyzes may also consistency checking, conformance to constraints, etc.
6. Management: Software architecture can be seen as a huge aspect in any software development process. Evaluation of an architecture leads to clear understanding of requirements, implementation plans and possible risks. This will reduce the amount of rework needs to be done in order to address problems later in the life-time of the system.

3.1 Quality Attributes

A quality attribute (QA) is a non-functional requirement and a measurable feature of a system, which is used to determine how well a system satisfies the stakeholders. In the ISO/IEC 25010 - also known as SQuaRE (Software product QUality Requirements and Evaluation) -the product quality properties

are categorized into eight characteristics (see [ISO, 2011a, P. 3]): Functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability and portability. Each of these characteristics is itself composed by a set of related sub characteristics.

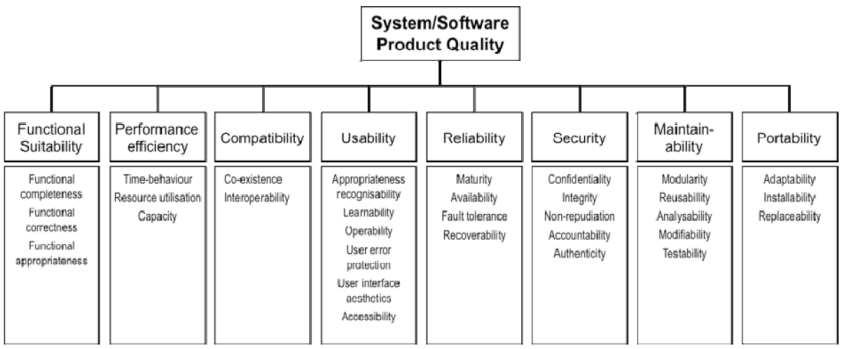

Figure 2: ISO 25010 Product Quality Model (see [ISO, 2011a, P. 4])

Those quality attributes are strongly related to non-functional requirements of a system. Those quality attributes must be defined before designing the system, in order to ensure that all of them are met accordingly.

In the next section all the important quality attributes for a Microservice Architecture are discussed:

3.2 Software Quality Measurement

Measurement in software engineering is seen as a crucial factor to evaluate the quality characteristics like functionality, usability, reliability and so on. Measurement allows to understand the current situation and helps to come up with clear benchmarks, that are useful to set goals for future behavior. Whereas some measurement principles are becoming more and more over stressed, other equally important design methods have been omitted in the architecture measurement process.

According to [ISO, 2018] the main objectives of software quality measurement are:

- To assist the management to monitor and control the development and maintenance of software systems
- Observing the conformance of software to functionality and other requirements
- Serving as data source for improvement by identifying low performance and demonstrating the achievements of corrective actions

Some of the quality attributes state in Figure 2 can be measured using various characteristics of the software architecture like size, complexity, coupling,

cohesion or others. Those measures are well defined in the IEC 25023 (see [ISO, 2011b]). Furthermore each quality attribute can be measured by combining it with others.

In [ISO, 2011a] it is defined, that the measurement is a logical sequence of operation in order to quantify the properties with the respect to a specific scale. The quality characteristics and their sub characteristics can be quantified by applying measurement functions, which is an algorithm used to combine quality measure elements. More than one software quality measure may be used to measure a quality characteristic or their uncharacteristic. An example for this is stated in [Mordal-Manet et al., 2013] where multiple metrics are combined together in order to measure a certain property using either composition or aggregation:

- Composition
  - simple or weighted average of the metrics: can be used only when the different metrics are using same ranges and semantics
  - thresholding
  - interpolation
  - Additionally a combination the the values above can also be used.
- Aggregation (several steps are required)
  1. Apply a weighting function to each metric
  2. Average the weighted values
  3. Compute the inverse function of the average

4 Quality Attributes for Microservice Architecture

According to [Li et al., 2021] there is no relatively systematic panorama for state-of-the-art quality attributes for MSA yet. It is still unknown how QAs are impacted in MSA and which QAs are the most challenging one. Hence many aspects are still unclear, unexplored or even inconsistent. The research team in [Li et al., 2021] did a systematic literate review of QAs which are the most concerned ones, and found that those are scalability, performance, availability, monitorability, security and testability (see [Li et al., 2021, P. 7]) . In their second research question they tried to find tactics, with whom it is possible to reach those QAs.

4.1 Scalability

Scalability can be done in two dimensions: horizontally and vertically. When scaling horizontally (scaling out), more resources to logical units are added - like servers to an already existing cluster. When scaling vertically (scaling up)

more resources to a physical unit are added (like memory to a single computer). In cloud environments the horizontal scaling mechanism is called elasticity (see [Bass et al., 2012, Part II.12]). In [Abbott, 2015] a scale cube with X,Y and Z axis is defined:

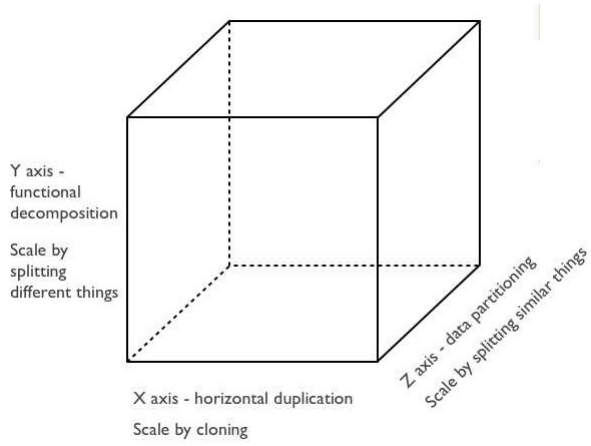

Figure 3: The Scale Cube (see [microservices.io, 2021])

4.1.1 Horizontal duplication (X-Axis Scaling)

With horizontal duplication, multiple instances of an application are running behind a load balancer. If there are N instances, each instance handles 1/N of the load. This approach is commonly used when scaling an application. Each of the copies running has potentially access to all data, this may require a higher amount of memory for the cache to be more effective. This approach does not tackle the problems of an increasing development and application complexity.

4.1.2 Vertical decomposition (Y-Axis Scaling)

With vertical decomposition, an application is splitted into multiple different services. Each of the service is then responsible for one or more closely related functions. One possibility to split would be a "verb-based" decomposition where services with single usecases are implemented (Logical Coupling). Another possibility would be to decompose them "noun-based", where services are responsible for all operations which are related to a particular entry (Semantic Coupling) (see [Abbott, 2015, microservices.io, 2021]). Also a mixture of both is possible. When using vertical decomposition the granularity must be decided carefully in order to get an appropriate balance between scalability and performance, since the network is an important factor.

4.1.3 Lookup-Oriented Split (Z-Axis Scaling)

With Z-Axis scaling each server runs an identical copy of the code, which is similar to X-Axis scaling. The difference is, that each server is only responsible for a part of the data. A component of the server is responsible for routing requests to the appropriate server. A possible routing criteria would be the primary key of an requested entity. This splitting method is used to scale databases, where data is partitioned (or sharded) over a set of servers based on an attribute of the dataset. Z-Axis scaling comes along with some pros and cons:

Pros:

- A server only needs to deal with a subset of data
- Cache utilization is improved and memory usage is reduced
- Transaction scalability is improved, since requests are distributed over multiple servers.
- Fault isolation is easier, since only a part of the data is affected
- Graceful behaviour in case of errors is possible

Cons:

- Platform complexity is increased
- Partition Scheme must be implemented, which can be hard if the data needs to be repartitioned
- Does not solve the problems of increasing development and application complexity.

4.2 Performance

Performance is the measurement of an systems ability to meet certain timing requirements as an response of an event. The allocation of different amounts of resources may influence the throughput or the response time of an service. On the other side the communication among Microservices via the network may directly impact the performance of systems. Thus performance and scalability can be rely on each other: Smaller Microservice may increase the scalability, but may decrease the performance due to a higher amount of interactions between the services. Whereas the performance may be improved if two Microservices are merged in order to reduce the communication overhead. According to [Cojocaru et al., 2019] performance tactics may be divided into four categories:

4.2.1 Resource Management and Allocation

With resource management it is tried to respond in an effective way to requests. According to [Bass et al., 2012] resource demand of systems is not controllable. With this, effective management of the available resources on the response side can be seen as critical factor in terms of performance. In order to use this tactic, a performance analytical model is usually using what-if analysis and resource planning approaches in a systematic manner. As example for performance modeling in [Khazaei et al., 2020] 3-dimensional Continues Time Markov Chain (CMTC) or Fluid Stochastic Petri Nets (FSPN) are mentioned. Possible performance objectives are latency, costs or resource utilization. Based on the performance model, engines are necessary for the allocation and management of containers, virtual machines or physical servers. The main constraint in this tactic might be searching the best combination of relevant variables and specific objectives. In order to find this combination, simulations or experiments could be performed.

4.2.2 Load Balancing

With load balancing it is tried to distribute the incoming load of a single Microservice among many instances. With this way the response to requests may be faster and the available capacitance utilization may be better. Furthermore it is ensured, that none of the services is significantly overworked in comparison to other ones. In an MSA the requests to services may form some sort of a chain. High load on particular services may block the processing of requests and decrease the performance significantly. To ensure a proper distribution of the load among all available microservices, a load balancer comes into place. For microservices there are two types of load balancer which can be seen in 4:

1. Centralized Load Balancing: This is currently the most common way of load balancing. With this approach a centralized load balancer (on server-side) is created. This type of Load Balancer usually works with push based algorithms, like Round Robin, JSO or JIQ (see [Abad et al., 2018]). In this approach the centralized load balancer is transparent to the clients and the clients are not aware of the service list. The load balancer itself is responsible for the server-pool underneath and monitors their health. A server-side load balancing solution which also comes along with service discovery is consul.io or Eureka.This approach has its drawbacks when the load balancer has to deal with a huge amount of clients and can also be seen as a single point of failure.

2. Distributed Load Balancing: More and more MSA are using an de-central approach while load balancing. Therefore the client itself holds an load balancer and gets a list of available servers and distributes the request to several servers according to specific pull-based load balancing algorithms (see [Rusek et al., 2017]). The client holds a list of available servers and selects one of the list either randomly or according to an algorithm. Net-

flix implemented their own client-side load balancer "Ribbon" which is also available open-source (see [Netflix, 2021a]). The active and frequent pulling of data by the clients may lead into not recent data every time. As a result this will cause a waste of bandwidth.

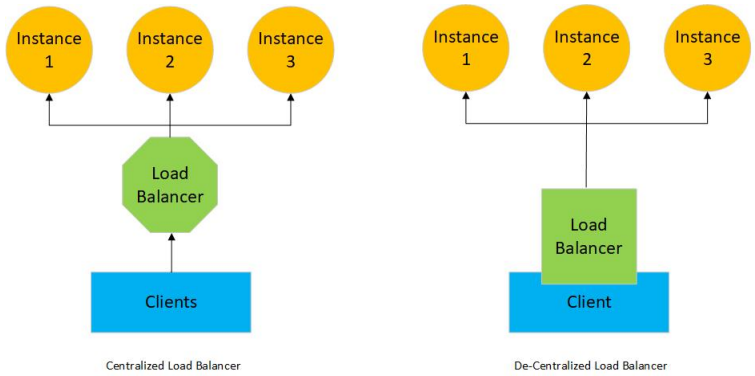

Figure 4: Centralized and De-Centralized Load Balancing Mechanisms

4.2.3 Containerization

Containerization is used more and more in the field of MSA and provides a higher performance than virtual machines (VM), since simple processes are not enough for Microservices (see [Eberhard, 2018, Part 5.1]). For example a crashing Microservice should not affect others or decrease their performance. The reason is, that each VM ususally has an own guest-operating system, which causes a huge overhead in terms of memory and storage. The most representative technology in terms of containerization is Docker. Docker increases the performance of microservices in two terms (see [Kang et al., 2016]):

1. Containers can share the same host OS and require guest processes to be compatible with the host kernel. This can reduce the overhead of communication among different guest operating systems (OS) for different container processes.

2. The lightweight characteristics of Docker containers can help creating and running more microservices, which furthermore contributes to a higher resource utilization. Furthermore containers can be easily moved to more performant machines.

Nevertheless containerization has also some drawbacks: all services need to access some OS-level modules for data (e.g.: in memory state, kernel modules). This makes it in particular difficult to hold them portable and scaleable. To achieve this Kubernetes brings a set of important features:

- Docker-Container can run in clusters and share all resources of those cluster

- In case of crashes it is possible to restart Docker Container automatically.
- Kubernetes is operating on the same level as Docker-Containers. Thus Microservices can run without any code-dependencies to Kubernetes.

4.2.4 Profiling

With profiling it is tried to detect performance issues and make a preparation for performance optimization. Some architectural design decisions may impact the performance of Microservice-based systems. Those might be the granularity or the deployment environment which is selected. With profiling it is tried to address the sources of performance unpredictability accross an Microservice's critical path. Furthermore it is tried to figure out potential impacts, bottlenecks as well as optimizations. Therefore profiling analyzes the characteristics of an Microservice-based system including required memory, CPU or bandwidth and provides both a scheduling engine and an auto-scaling resource manager with essential information to optimize the performance. (see [Filip et al., 2018]). The two most common profiling tactics are (see [Nicol et al., 2018]):

- CPU Profiling: analyzing the execution time of a service and finding the hotspots that needs to be optimized.
- Memory Profiling: analyzing the memory state or the memory allocation events. It is not only used when finding memory leakages, but also for optimizing memory usage.

Two well known profiling frameworks are Google-Wide Profiling (see [Ren et al., 2010]) or Netflix Vector (see [Netflix, 2021b]).

The usage of some lightweight profilers - like Linux Perf - can lead to an "always on" approach and support continuous performance assurance. (see [Nicol et al., 2018]). This should be avoided in order to not increase unnecessary costs in terms of data processing and storage. Profiling is important for performance predictability of an Microservice based system and is often needed for Monitoring and Auditing.

4.3 Availability

Availability measures the ability of a system to repair faults so that the period of an unavailable service does not exceed a required value. (see [Bass et al., 2012]). According to[Febrero et al., 2016] availability is a broad perspective and encompasses what is normally called reliability. The availability of a system can be calculated as the probability that it will provide the specified services within required bounds over a time interval. In order to calculate the availability-rate shown in 1 where MTBF is the meantime between failure and MTTR is the meantime to repair is used.

Algorithm 1 Availability Rate

$$AvailabilityRate = \frac{MTBF}{(MTBF + MTTR)}$$

This formula also has some drawbacks: Scheduled downtime (when the system is intentionally taken out of service) may not be considered when calculating. In those cases the system is marked as "not-needed". If a user whats to access the system during this scheduled downtime and the user is waiting for it, it is not counted against any availability requirements. (see [Bass et al., 2012]) Availability tactics can be grouped into four types:

4.3.1 Fault Monitor

The Fault monitor approach is commonly used for discovering faults of Microservices through continuous health monitoring using a specific component. In order to ensure a high availability it is necessary to detect or anticipate the presence of a fault before the system can take actions to recover from faults. (see [Bass et al., 2012]). Therefore Microservices require continuous monitoring, so that their health can be analyzed automatically and responsively react to failures with as less human interaction as possible. Below three typical kinds of fault monitors are described:

- Centralized Monitor: A centralized monitoring system collects the results of service invocation based on the health analysis provided by the service discovery mechanism. This updates the service status in the service registry, which can minimize the downtime of any Microservice during the recovery from failures. This approach is mostly implemented as service side application. As a drawback an additional service needs to be deployed, which will consume some resources and may cause a single point of failure. (see [Cojocaru et al., 2019]).

- Symmetric Monitor: With symmetric monitoring a service approaches its service neighbors, including successors and predecessors. For example, an Internet of Things (IoT) watchdog implements this technique for neighborhood monitoring, which is purely symmetric in monitoring relationships. Broken Microservice instances will be replaced by repaired or replaced by a replica in another IoT device. As a drawback this approach might lead to inconsistencies. (see [Celesti et al., 2017])

- Arbitral Monitor: An arbitral monitor detects failures or conflicts based on a decentralized and independent group of nodes. A detected failure by a single node must be confirmed by the majority of nodes in the arbitrator group. With this method cascading failures can be addressed and the number of nodes leaving the system can be reduced. (see [Kakivaya et al., 2018])

4.3.2 Service Registry

With this approach the location of all running services is stored at a central point. This central point can be used by other Microservices to retrieve binding information about other services in the MSA. Microservices might have various instances deployed and this amount may change from time to time. Thus a service needs a way to locate the position of another specific service to detect faults with high efficiency. After startup the service registry registers all the available services and removes them if - for example - a periodic heartbeat is not received from the application. Typically there are three approaches for service registry (see [Haselboeck et al., 2017]):

- Self Registry: each service can register and deregister itself using a local registry. Therefore the service registry has to store the locations of the Microservice instances. The client service itself has to implement the service registry client libraries.
- Third-Party registry: With this approach a third-party registry service, which does the automatic registration or deregistration of Microservices. This third-party registry has a registrar component and an additional service registry which actually stores the locations. With this approach an external library is no more needed.
- Manual registry: The registration and deregistration of services is done manually at the service registry. On the one hand this provides a better interoperability since it supports users to include Microservices from other systems based on their needs. On the other hand this approach may cause long time of responding to Microservices with failures since the registry cannot check the health of the services. Therefore this solution is not suitable for big systems whose Microservices have been available and stable for a long period.

The service registry itself also has some drawbacks, since it is a single point of failure. Additionally the implementation of clients should adapt to all programming languages used.

4.3.3 Circuit Breaker

When using with Microservices, a circuit breaker prevents request to a service in case of a failure. Whenever a failure is faced the circuit breaker "opens" depending on a specific threshold. After a defined time the circuit breaker closes again and allows request to a Microservice. The breaker will not open again until another error occurred. This approach is used, if a failure of one service is cascading to other services which also may lead to the failure of an entire system. According to [Torkura et al., 2017]a circuit breaker may have three different states:

- Closed: if the circuit breaker is closed, request to a service are allowed to be passed to the target Microservice.

- Open: if the count of faults or timeouts is exceeded, the circuit breaker will change its state to open. This will prevent requests to be forwarded to the Microservice.
- Half-Open: After a periodic observation time of the health state, the "half-open" state may be triggered. This will allow a certain amount of requests to be forwarded to the service.

In [Montesi and Weber, 2018] three different implementation approaches for the circuit breaker pattern are mentioned

- Client-Side Circuit Breaker: with this approach a separate circuit breaker for intercepting calls to an external service is implemented. The main advantage is, that an open circuit breaker will prevent its target service from receiving further requests. In contrast to that, clients can be malicious and the information about the availability is no more actual. To counteract this issue, all clients might regularly ping each target service to get information about its health.
- Server-Side Circuit Breaker: This approach needs an internal circuit breaker, which decides whether the invocation should be processed or not. This requires resources to execute the service-side circuit breaker and receives messages even if the circuit breaker is open.
- Proxy Circuit Breaker: In this approach uses a proxy between the client and the service. The proxy can be a single one for multiple services or also a single circuit breaker for each single service. This way has two main benefits. First, no deep changes need to be made either at the client, nor at the service. Second, clients and services are equally protected from each other. Therefore clients are resilient against faulty services and services are shielded against too many request from a single client. As a drawback it has to be mentioned, that the proxy needs to be updated every time a service API changes.

The approach of circuit breakers requires additional requests and responses to perform some sort of handshaking before each communication. Furthermore it is challenging to get response in case of open circuit breaker state.

4.3.4 Inconsistency Handler

Consistency in huge MSA is hard to achieve, since users want to have the exact same answer if forwarded to multiple nodes. Therefore multiple instances of a Microservice in a MSA also need to interact in parallel with a data repository, which again creates some challenges in terms of data consistency. According to the CAP Theorem consistency, availability and partition tolerance are traded off between each other. Only two of them can be kept in failure mode. For a Microservice based architecture that needs to have a high availability and needs to be partition tolerant - means running over the network - the consistency has to be limited (see [Furda et al., 2018])

Inconsistency handler usually have to deal with unstable or unreliable networks with limited bandwidth. Therefore it is sometime unfeasible to simultaneously access multiple replicas in MSA. All read and write operations are only done on one replica each time which furthermore leads to temporary data inconsistencies. This problem needs to be addressed with synchronization of data throughout all replicas to achieve a momentary consistency of data. The events can be stored in a database using transaction logs, which then can be replayed in order to update all other instances of the Microservice. For inter - service communication asynchronous messaging techniques, like Kafka or RabbitMQ, are used. Those provide different publish-subscribe techniques to deliver events to the respective subscriber. (see [Furda et al., 2018])

The final consistency may cause some problems when a backup of an entire system needs to be performed. It might be possible, that the backup stays inconsistent because of those two reasons mentioned in ([Pardon and Pautasso, 2017])

- Broken Links: A reference can no longer be followed (e.g.: a Microservice A cannot be found from Microservice B because it is no longer available)
- Orphan State: This occurs when there is no reference to be followed. (e.g.: Microservice A is no longer references from the state of the Microservice B recovered from an obsolete backup)
- Missing State: Occurs whenever multiple instances backups are not consistent. (e.g.: Microservice A remains consistent with Microservice B until it crashes. After recovery from an obsolete Backup, A does not have the state corresponding to the latest events logged by B)

Therefore an appropriate disaster recovery plan needs to be prepared during the architectural phase of the Microservices. It needs to be defined how backups can be performed and also with the consequences of restoring a backup from a possible inconsistent backup.

4.4 Monitorability

Monitorability is used to measure the ability to support the operations staff to supervise the system while it is executing. It is a complex part of Microservice based systems but becomes an more and more essential due to the high level of dynamic structure and behaviour of such a system. As an example the monitoring might be on infrastructure level (VM or Container), application level (e.g.: response time) or on environment level (e.g.: network). Improving monitorability may also have an impact on other QAs like scalability, performance and availability. (see [Bass et al., 2012])

4.4.1 Generating Monitored Data

Whenever a Microservice needs to be monitored, some data on different levels (host, platform, service,...) must be collected. It needs to be defined which data should be collected. The data generated at different levels of the architecture

relates to different concerns and requirements. Monitoring of host, platform or services metrics can assist to know the runtime information of microservices in different aspects like available hosts, service reponse time, failure rate and others. Those metrics are important in terms of Microservice health and help to decide in relation to performance. In [Mayer and Weinreich, 2018] three different solutions are proposed:

- Instrumentation: Instrumentation service is a prerequisite in order to collect the static and dynamic (runtime) information for each service at different levels. Instrumentation can be again divided into three main tactics:
  - Host Instrumentation: Therefore an agent needs to be installed on each host of the MSA. This agent automatically detects new Microservices and starts to collect the data.
  - Platform Instrumentation: Each runtime environment (eg..: Database Server, Webserver,...) is instrumented. An agent needs to be installed on each platfrom at a specific host. This implementation is platform independent from the operating system and can be more technology specific than an agent on the host level.
  - Service Instrumentation: Requires an agent for each service. This provided analysis of runtime metrics and interactions but no information about the service runtime environment.
- Logging: With logging all incoming and outgoing requests are logged to a file on the hosts filesystem. Each log entry may represent a specific request, composed by a timestamp, response time, response code and others. These files are periodically fetched and analyzed by specific logging frameworks like Amazon CloudWatch or Logstash.
- Distributed Tracing: This tactic allows to determine the initiator of a request sequence and helps with searching for root causes of issues. Basically it is logged which service (or instance) called another service (instance).

All of those mentioned tactics need support of other technology stacks which also can be some sort of a burden for developers.

4.4.2 Storing monitored data

After generating monitoring data (see 4.4.1) the collected data needs to be stored . Storing of monitoring data can be either central or de-central (see [Mayer and Weinreich, 2018]):

- Centralized Storage: All data from distributed Microservices is stored at one central place. It allows the analysis of data in a centralized style with less administration and overhead needed. However a central storage may become a bottleneck and also a single point of failure.

- Decentralized Storage: On each host / platform or service a local component is responsible for storing monitored data. With this approach it is possible to analyze service interaction which is a crucial requirement in each MSA. This solution provides a higher scalability since the installation of new services on new hosts will also provide new local storage. As a drawback a loss of monitoring data in case of unavailability of a local service can be see. Furthermore the administrative work is higher.

The storage of monitored data is closely related to generating monitoring data, since the generated data can be stored with this tactic.

4.4.3 Processing monitored data

After generating and storing monitored data, the monitored data itself should be processed accordingly. With this various kind of monitoring data collected, targeted processing and analyzing tactics are required to deal with the different purposes. Mainly there are two different solutions for different purposes (see [Mayer and Weinreich, 2018]):

- Aggregation: With aggregation the log data is stored in an aggregated form. With this tactic the amount of storage can be decreased and the data is available for long term analysis.
- Non-Aggregation: Data is stored in native form which enables detailed analysis. The data is not lossfully and thus the amount of storage is higher.

4.4.4 Presenting monitored data

After processing and analyzing the monitoring data needs to be presented and visualized for various stakeholders (see [Haselboeck and Weinreich, 2017]). Different stakeholders require different views of the system and key metrics to be visualized in order to be aware of the current state of the entire system and make on-time decisions (see [Mayer and Weinreich, 2018]. According to [Li et al., 2021] there are five different approaches for presenting monitoring data:

1. Microservice specific metrics for analyzing response time, failure rates, throughput.
2. IT Landscape specific monitoring data allows analysis and long term reports data of available hosts and data centers, host utilization or service allocation to hosts.
3. Infrastructure specific monitoring data visualizes data like CPU and memory consumption.
4. The map of running services provides an overview of all running services and their interaction.

5. All responses from a service are shown. This helps finding the root cause in case of service failures.

A solution for monitoring of Microservice is the Elastic Stack (Elasticsearch / Logstash / Kibana / Beats). Elasticsearch is an open-source REST based search engine for logs. Logstash is the data-collection and log-parsing engine. Kibana is the visualization engine for all the collected log and also be used for other scenarios as well (e.g.: Observation,...) . Beats is a collection of light-weight data shippers.

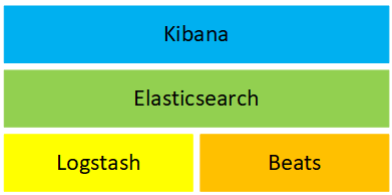

Figure 5: Elastic Stack

4.5 Security

Security is the the measure of the systems ability to protect data and information against unauthorized access while still providing access to people and systems which are authorized. (see [Bass et al., 2012]) Due to the fact, that an MSA distributes logic among multiple services, the network interaction is getting more complex. Thus attackers can exploit this complexity to launch certain attacks.

4.5.1 Security monitor

Microservices introduce many security challenges to be addressed. Possible concerns are the complex communication infrastructure or the lack of trust on specific Microservices (see [Sun et al., 2015]). Therefore a tactic is needed in order to flexibly monitor risky aspects of Microservice and their infrastructure to defend against internal and external threads. Security Monitors can either be a local or external monitor (see [Otterstad and Yarygina, 2017]):

- Local Monitor: is a way to include local security monitors into Microservices to monitor the network events.
- External Monitor: Enables more complex evaluation and an holistic view of the overall system. Vulnerability information can be directly consumed by other security applications and used for security tasks such as automated configuration of firewalls and also need to be integrated into Intrusion Detection Systems (IDS)

For assessment and monitoring of vulnerabilities and network-related security also security policies are necessary (see [Alenezi, 2016])

- Policies for vulnerabilities monitor:
  - Global policy for all the Microservices of an application
  - Microservice-specific policy targeting specific Microservice implementation technologies
  - Virtual Machine and Container Policy at infrastructure level in a cloud native environment
- Policies for network related security monitor:
  - Connection Policy: Determines if a Microservice is allowed to connect to another Microservice
  - Request Specific Connection Policy: Defining what kind of request a Microservice can make to another Microservice
  - Request Integrity Policy: Enforces request to be compliant to an defined schema and that the requested data is not compromised.

4.5.2 Authentication and Authorization

Authentication is a process in which a user or a system confirms its identity. Authorization is a mechanism by which a principal is mapped to an action allowing an identity to do. In some extends Microservices therefore have to imitate what monolithic structures are doing. (see [Jander et al., 2019]) That is each service is using an own database or shared database that stores credential data and implements own functions to verify the user independently. This approach has several drawbacks:

- Joining a new service in the system forces the authentication and authorization to be re-implemented.
- To be trustworthy, it is not enough to secure the Microservices individually. Communication channels also must be authenticated.

The tactic of authentication and authorization consists of multiple solutions, three are mentioned below:

- Key Exchange-based communication authentication: The basic mechanism for secure and authenticated communication is executing a key exchange between the processes. Key may be passwords, key or asymmetric key pairs. Therefore the authentication of the exchange is verified by the processes. The exchanged key is usually associated with all verifiable roles. An ephemeral key is usually generated by a symmetric authentication encryption scheme for the validation of encrypted messages, which also can be tagged with roles that were verified during the key exchange. This provides both confidentiality and authenticity. Another technology would be symmetric AES and the HMAC mechanism (based on SHA256) to ensure confidentiality and integrity. (see [Jander et al., 2019]). Key

Exchange may require a key management and moves the focus of trust from a small amount of (cryptographic) protocols to various unspecific mechanisms. This again leads to an trade-off between performance and security.

- Client Certificate: With Transport Layer Security (TLS) the communication between Microservices can be secured. Therefore HTTPS, which was originally based on SSL and now is based on TLS, can be used. TLS is able to authenticate both sides of a communication channel through server and client certificates (like X509). (see [Walsh and Manferdelli, 2017]). TLS Based client certificates are not used frequently and support should be invoked by the application source code, since this certificate does not have a strict mandate on the use and validation of certificates.

- Federal Identity: With this tactic the host-authenticated TLS tactic is extended with in-band authentication options which allows Microservices to use an identity management system that stores the identity of users for authentication. (see [Fetzer et al., 2017]). The trusted third party system usually permits in form of tokens (JWT - JSON Web Tokens). These systems can be implemented as separate service or just as a single sign-on framework like OpenID Connect, SAML or OAuth. Sibboleth or PingID are typical solutions for federal identity and security. Deploying and integrating of such security systems is very complex and requires special knowledge.

4.5.3 Intrusion Detection

It is necessary to understand the security status of an application and - if needed - react on security breaches. This is also important if an system is developed further with new versions and features. Vulnerabilities can be detected either by an local or external security monitor as described in 4.5.1. Once a vulnerability is detected, actions should be taken immediately in order to ensure the security of the system. Those actions might be the following (see [Yarygina and Otterstad, 2018])

- Rollback / Restart: Destroy the existing service instance and restart a new one with the same or even older configuration which was working without problems

- Isolation / Shutdown: Decouple the problematic services physically through shutting down

- Diversification: Recompilation or binary rewriting to incorporate randomness into the executing binaries. Also moving the service to another host may help to mitigate the attacks.

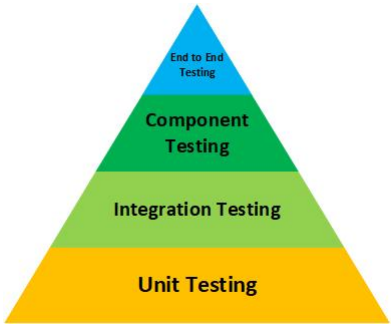

Figure 6: Testpyramid (see [Crispin and Gregory, 2009])

4.6 Testability and Documentation

According to [ISO, 2013] software testing is defined as:

> An activity in which a system or component is executed under special conditions, the results are observed or recorded and evaluation is made of some aspect of the system or the component

As the relationship between Microservices might be complex and some Microservices are evolving over time, it is necessary to pay attention to the testability of Microservices. Testability has a direct impact on performance, availability and security and is a measure of a system's ability to demonstrate its faults through (typically execution-based) testing. (see [Bass et al., 2012]). As stated in Figure 6 the type of testing can be visualized in form of a pyramid with the following items (see [Sam, 2015])

- Unit Test: Typically Functions and Methods are tested. No services are started and the usage of external files and network connections will be limited (Classes within Service)
- Integration Test: Tests Inbound and Outbound Adapters and their dependencies. (Service Adapters)
- Component Test: Testing is performed on each individual component separately without integrating with other Components (Entire Service)
- End-to-End Test: End-to-end tests involve the entire system. Sometimes also the user interface comes into account. (Multiple Services or Application)

As stated in the definition, testing may be performed on the entire software system (System Test or Acceptance testing) or on a Module (Unit Test). Those two types of tests can be automated. When developing in an agile way, the agile testing quadrants are a good schema to rely on.

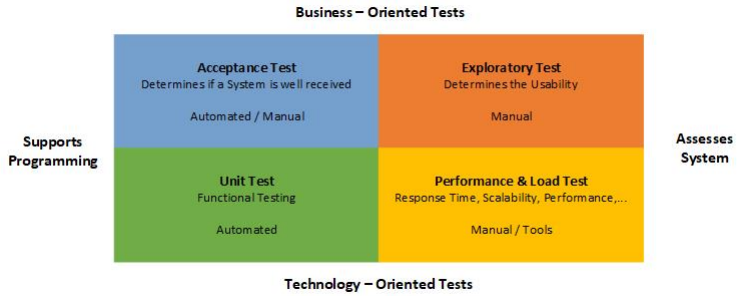

Figure 7: Agile Testing Quadrants (see [Crispin and Gregory, 2009])

4.6.1 Automating Test Procedure

With automated testing it is tried to limit the complexity of the testing process using some automatic techniques. This tactic came into place since it is impossible to test a large amount of Microservices by hand because of the time needed, the large number of tests, the complexity and the amount time which is needed to check the results. For efficiency improvement it is imperative to use automated testing. In order to implement automatic testing a testing model needs to be defined (see [Sam, 2015]). Examples for tools which will help to automize tests are Jenkins, Gitlab, Kubernetes and others.

4.6.2 API Documentation and Management

Since Microservices are relying on RESTful Webservices for the intercommunication, an API description is necessary and useful for service testing and documentation. Continuous updates or upgrades of APIs also require a fine-grained documentation. Therefore Swagger (https://swagger.io) is a widely used tool which provideds such components which are reducing the work for API documentation. When a RESTful API is configured with Swagger, RAML (https://raml.org/) can also be used to automatically provide a JSON file as a resource that will fully define which APIs are available in that RESTful service. With this tools testing of endpoints can be conducted in an more efficient way.

Another emerging way is the usage of gRPC (https://grpc.io/), which is a modern framework for remote procedure calls where service definition files are exchanged between two parties. gRPC provides code generators, which create the source code for the respective interface automatically.

4.7 Conclusion

A Microservice Architecture should be embraced with caution and decisions should be carefully made when deciding to migrate from an Monolithic Approach to Microservices. Whenever a new MSA should be developed it is easier, since one can start from scratch. Best practices for Microservice Architecture (starting from scratch) will be mentioned in section Microservice Architecture

Best Practices. But if an Monolithic Application should be changed to an MSA the return on investment should be taken into account. This calculation may refer to some not tolerable facts in the legacy systems that should be addressed by the new architecture. The investment means additional efforts and cost to implement the system, but also for specific QAs of the identified set.

Furthermore the complex relationships among QAs require to be considered during the migration to MSA, e.g, dependency or trade-off. On the one hand, the improvement of one QA may have positive influence (dependency) on other QAs (e.g.: for monitorability would also offer data support for performance optimization and capacity planning). On the other hand, addressing one specific QA may negatively impact certain other QAs. To summarize, there may be trade-off relationships between two or more different QAs.

5 Microservice Architecture Best Practices

In order to create an sustainable architecture for Microservices some principles must be met (see [Eberhard, 2018]):

- A Microservice Architecture must split its item into modules.
- Modules should interact via non-proprietary interfaces
- Certain modules should not depend on implementation details of other modules
- Integration options must be limited and standardized
- Communication should be limited to RESTful Webservices or Messaging Protocols. Authentication must be standardized on all services.
- Each Module should have an own continuous delivery pipeline
- Operation should be standardized: Same configuration, deployment, log analysisMicroservice Architecture of CaRE,..
- Modules should be resilient. Modules should not fail if other modules are not reachable.
- Two independent layers in terms of architectural approach:
  - Macro Architecture: Decisions regarding all modules (e.g.: Authentication)
  - Micro Architecture: Decisions which can be made for each single module. (e.g.: Database System)

But if all of those principles are met, it is not sure that the architecture is well designed and without mistakes. The complexity of an MSA leads to attack surfaces in terms of bad decisions when defining and implementing the architecture. Therefore it can be distinguished between Antipattern and Code-Smells.

Sometimes Antipattern and Smells are seen as one same issues, also the transition in literature is unclear and sometimes blurred. In [WikiWikiWeb, 2021] Architectural Smells and Antipattern are defined as followed:

- Architectural Smells: Smells should be investigated. Smells may and may not be bad.
- Antipattern: Will lead to worse design of a system. Antipattern are always bad and should be solved immediately.

In the following sections some Architectural Smells and Antipattern in relation to Microservices are discussed. It was tried to distinguish between Bad Smells and Antipattern, but - as mentioned - this difference is blurred.

5.1 Architectural Smells

In this part some Architectural Smells within MSA are discussed. It is also discussed how such smells can be prevented and/or solved.

5.1.1 Hard-Coded Endpoints

Hard-coded IP Addresses and Ports of the service which needs to be used. This leads to problems when the service location needs to be changed. (see [Pigazzini et al., 2020]) This could be solved by using either a service discovery or a well designed DNS System with the ports set in an configuration file.

5.1.2 Shared Persistence

Different Microservices are accessing the same database. In worst kind, different services are using the same entities of a service. This approach couples the Microservices connected to the same data and reduces the service independence. (see [Pigazzini et al., 2020]) Solutions for the smell are (see [Taibi and Lenarduzzi, 2018]):

- usage of independent databases for each service,
- usage of a shared database with a set of private tables for each service which can only be accessed by the service, or
- usage of a private database schema for each service

5.1.3 Independent Deployability

In MSA each service should be running independent from other service, which means that it should be possible to deploy and undeploy a Microservice independently from others. (see [Newman, 2019]) This also implies, that a Microservice can be started without waiting for other Microservices to be running.

Container Systems (like Docker) are providing an ideal way to deploy independently, if each service is running in an own container. If two services (A &

B) are placed in the same container, both of them would operationally depend on each other, since launching an new instance of service A would also start an instance of service B.

5.1.4 Horizontal Scalability

A consequence of independent deployability (5.1.3) is the possibility of adding and removing replicas of a Microservice. In order to make sure, that the architecture his horizontal scaleable, all replicas of Microservice A should be reachable by the Microservices invoking A.

This smell occurs when no API Gateway is used or locations of other services are hard-coded into the source code (Endpoint-Based service Interactions). To solve the smell of Endpoint-Based service interactions, a service registry should be implemented. (see [Abbott, 2015])

5.1.5 Isolation of failures

A Microservice can fail for many reasons: Hardware issues, network issues, application level issues, bugs,... and becomes unavailable for other Microservices. Furthermore the communication may fail due to the high complexity of the entire systems. Therefore Microservices should be designed in order to tolerate such failures and still remain up and running. Failures should not be cascaded to underlying services, which means that a failure in Microservice A results in triggering an failure in Microservice B. The most common solution is to use a Circuit Breaker (see 4.3.3). (see [Jamshidi et al., 2018])

5.1.6 Decentralization

In all aspects of Microservice-based Applications decentralization should occur, which implies that also the business logic should be fully decentralized and distributed among the Microservices. (see [Zimmermann, 2016]) This occurs whenever a Enterprise Service Bus (ESB) is misused. Whenever a ESB is placed as a central hub - with the other Microservices as spokes - it becomes an bottleneck. This approach may lead to undesired centralization of business logic. (see [Rusek et al., 2017])

5.2 Anti Pattern

As already mentioned Anti-Pattern are leading to a worse design of an Architecture. According to [Tighilt et al., 2020] Antipattern can be divided into four groups:

- Design Antipattern
- Implementation Antipattern
- Deployment Antipattern
- Monitoring Antipattern

5.2.1 Design Antipattern

In this section examples for Antipattern related to a bad design of Microservices are discussed.

- Wrong Cut: Occurs if Microservices are split on the basis of technical layers (presentation, business and data-layers) instead of business capabilities and leads to a higher complexity. In order to mitigate this, a clear analysis of business processes needs to be performed. (see [Taibi and Lenarduzzi, 2018])
- Cyclic Dependencies: In cyclic dependencies a Service B is called by A, B calls C and C again calls A. Microservices involved in cyclic dependencies can be hard to maintain or reused in isolation (see [Pigazzini et al., 2020]). As a solution the cycle should be refined and an API Gateway should be applied.

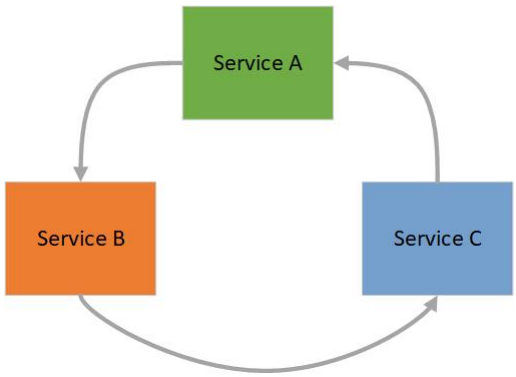

Figure 8: Cyclic Dependencies

- Nano Service: Single Microservice should be designed in order to fulfill single business capabilities, not more but also not less. Nano services may increase the opportunities for reuse and help developers to focus on important Microservices. Nevertheless they require more developers and the context is switching often. During runtime Nano Services are creating more overhead and communication. (See [Pigazzini et al., 2020])
- Mega Service: As opposite too Nano Services, also Mega services should be avoided. Microservices should be designed to fulfill only one business capability. Mega Services are difficult to test and also create maintenance efforts and increased complexity. (see [Pigazzini et al., 2020])

5.2.2 Implementation Antipattern

In this section examples for Antipattern related to implementation of Microservices are discussed.

- Shared Libraries: Microservices should not share runtime libraries and source code directly. This somehow breaks the boundaries between Microservices, which then cannot be seen as independent and independent deployable. (see [Pigazzini et al., 2020])

- Too many standards: Although Microservices are allowing the usage of different technologies, too many different protocols, frameworks, development languages etc. are used. This may lead to problems in companies, especially in the event of developer turnovers. (see [Taibi and Lenarduzzi, 2018])

- Too new technology: The technology used needs to be defined will. Too new technology is not always the best choice, since it might not be fully developed. (see [Taibi and Lenarduzzi, 2018])

Depending on the definition of Antipattern, also Hard Coded Endpoints (5.1.1) can be seen as an implementation Antipattern.

5.2.3 Deployment Antipattern

In this section examples for Antipattern related to deployment of Microservices are discussed.

- Manual Configuration: Basically Microservices are relying on some sort of automation. Everything which is possible, should be automized. Therefore a configuration file for each Microservice is not the best solution. Instead a configuration server should be used which automizes the configuration process. (see [Taibi and Lenarduzzi, 2018])

- No Continuous Integration (CI) / Continuous Delivery (CD): The independent deployability of Microservices also offers the possibility to apply iterative continuous development and deployment (DevOps) processes. The integration of DevOps results in a reduced delivery time, increased delivery efficiency, a decreased time between releases and a higher software quality. (see [Taibi and Lenarduzzi, 2018])

- No API Gateway: Microservices are communicating directly with each other. If no API Gateway is in place, the service consumers are also communicating directly with the Microservices. This increases the complexity of the system but also makes a system harder to maintain. (see [Tighilt et al., 2020])

- Timeouts: The availability of a Microservice refers to the possibility for each service consumer to connect and send a request. Responsiveness is the time which is taken by the service to respond to a certain request. In distributed systems consumer applications / tasks use timeouts to handle the unavailability or unresponsiveness of a service. It is hard to find a right timeout value: a too short value will quickly lead to exceptions, a too long value will force clients applications to wait too long before

stopping the request. As a solution Circuit Breaker (4.3.3) can be used. (See [Pigazzini et al., 2020])

- No API Versioning: In some cases multiple Versions of APIs must be exposed by a service. This is usually the case if major changes had been done on an particular Microservice. Therefore a version number can be part of the request URL or the version can be inserted into the HTTP header of the request. (see [Taibi and Lenarduzzi, 2018])

5.2.4 Monitoring Antipattern

In this section examples for Antipattern related to monitoring of Microservices are discussed.

- No Health Check: Since a Microservice can be deployed everywhere and also can be unavailable due to certain reasons, the status of a Microservice might be unknown. Therefore an health check API endpoint should be implemented, which periodically verifies the health status and the ability to answer requests. (see [Pigazzini et al., 2020])

- Local Logging: During runtime each Microservice produces a lot of information with logging. These logs are usually stored directly on the file system and cannot be accessed directly. Therefore a central logging service should be implemented and used (4.4) . Distributed logging mechanisms are easy to implement and make debugging easier. (see [Pigazzini et al., 2020])

- Insufficient Monitoring: If Microservices are part of Service Level Agreements (SLA), the behavior and performance is crucial. Insufficient Monitoring is linked to "Local Logging" and may hinder maintenance activities in a Mircoservice-based system, because failures become difficult to catch and performance tracking is not available. This can even affect the ability to comply with the SLA. To solve this a global monitoring tool should be implemented. (see [Pigazzini et al., 2020])

6 Microservice Architecture of CaRE

In this section the Microservice Architecture of CaRE (Custom Assisstance for Remote Employees) will be described. With all quality aspects and best practices in mind an Microservice Architecture for installation and commissioning of Transformers was created. An architectural overview is shown in Figure 9.

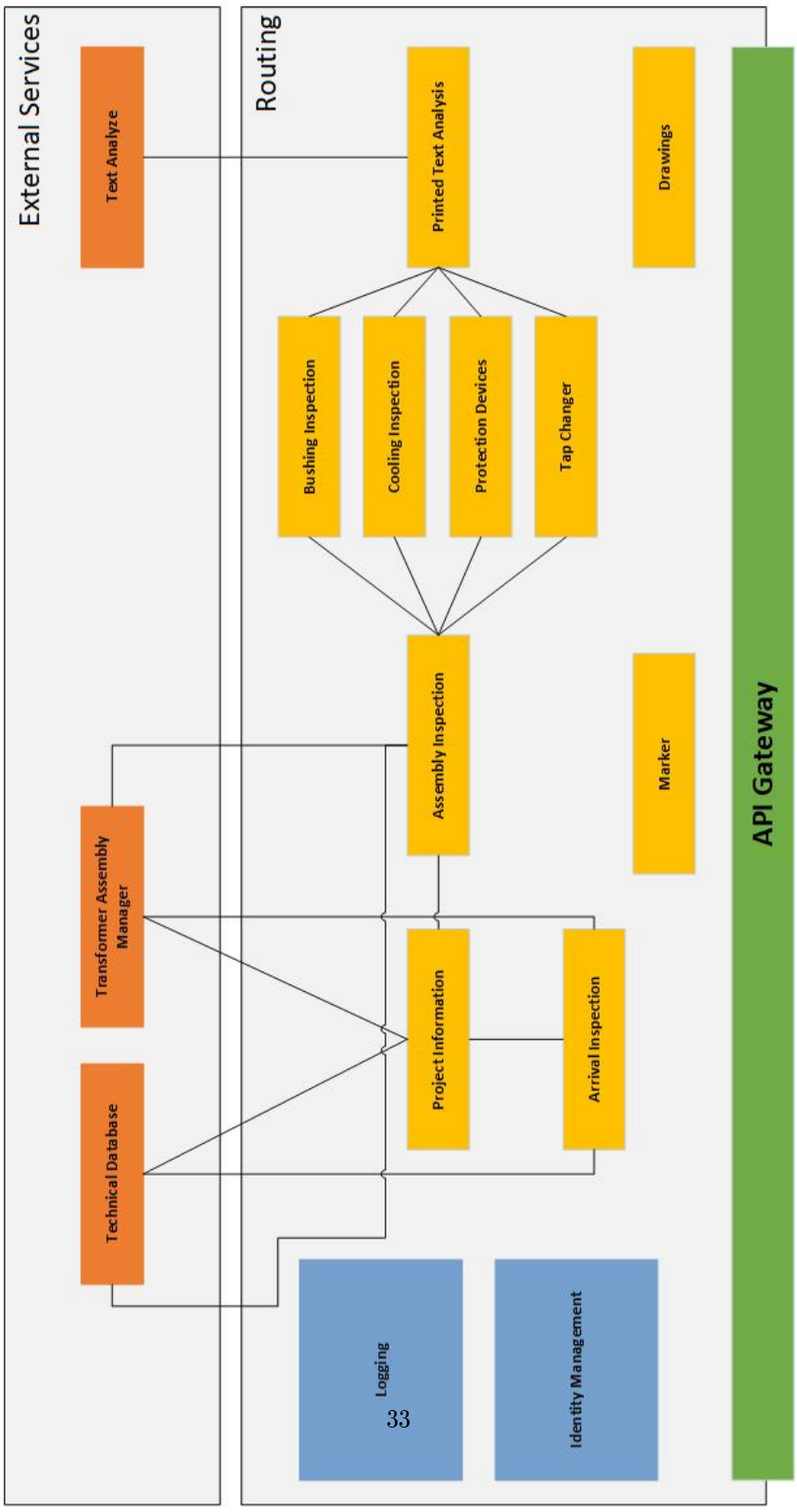

Figure 9: Architectural Overview of CaRE

6.1 Microservices

For the proof of concept, each business capability (or process) was developed as an own service. Each of those Microservices was implemented as a standalone service with its own database. For now, all services were implemented in .net Core and can thus run in own Docker Container. Thus the following services were implemented:

- Project Information: Holds all the information about a single project. Basic Information about Projects is collected from the external Services "Technical Database" and "Transformer Assembly Manager".
- Arrival Inspection: With Arrival Inspection the process of Arrival Inspection is reproduced. All information regarding Arrival Inspection is collected and provided. Therefore the Microservice connects to the external services "Technical Database" and "Transformer Assembly Manager"
- Assembly Inspection: With Assembly Inspection the process during Transformer Assembly is reproduced. The Assembly Inspection therefore get design and structure information from two external services "Technical Database" and "Transformer Assembly Manager" and sends the respective data to the respective sub-services. If a part is not available at the respective Transformer, no information will be sent to the respective Microservice.
  - Bushing Inspection: This Service is responsible for the process of Transformer Bushing installation and commissioning.
  - Cooling Inspection: This Service is responsible for the process of installing and commissioning the Cooling System of a Transformer.
  - Protection Devices: This Service is responsible for the process of installing and commissioning the Protection Devices. It was decided to use an single Microservice here, because those devices are changed frequently and each change must be logged accordingly. It must be ensured, that the history of changes is not compromised.
  - Tap Changer Inspection: This Service is responsible for the process of installing and commissioning the Tap Changers of Transformers.
- Printed Text Analyze: In some cases long serial numbers need to be inserted in order to track them. From usability perspective this is not ideal.. So an service was developed which does OCR on the images which were send. This service is relying on an external OCR Service in Azure.
- Drawings: Since a lot of drawings are necessary for Transformer commissioning an own service for maintaining and storing of those files was developed. For now this service is not connected to any other external services, but it would be possible to connect it to an company-internal drawing repository.

- Marker: In some cases "digital redlining" needs to be performed, which means that annotations are done on pictures or changes need to be marked on drawings. These information also need to be stored persistent.

For creating the database structure of each service the tool "Entity Developer" was used. With the tool it is possible to graphically create the database structure and automatically generate the database tables and ORM (Object Relational Mapper) classes. In Figure 10 a screenshot of the - rather easy - database structure of the Marker Service is shown.

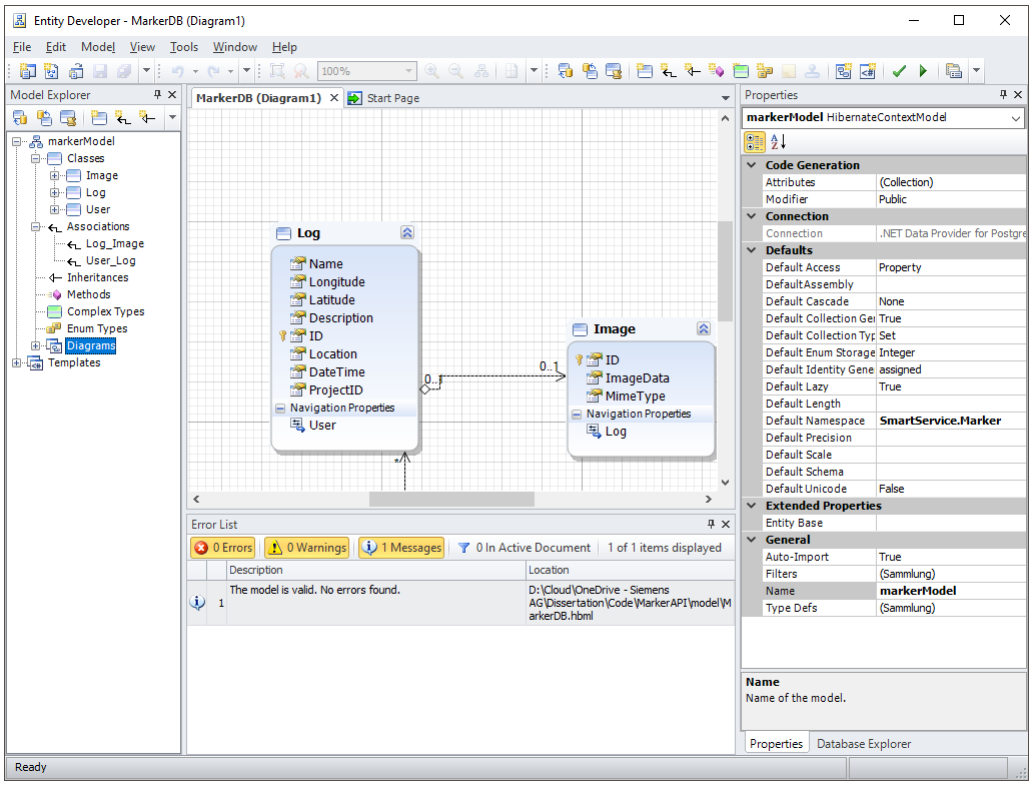

Figure 10: Marker Database Structure in Entity Developer

6.2 Auxiliary Systems and Services

6.2.1 Logging

For logging purposes a central logging system with the Elastic Stack was created. For the proof of concept Serilog (https://serilog.net/) was used in each of the services for Logging. In contrast to Logging Frameworks like log4j, NLOG, log4net and others, Serilog provides the possibility to store log files in a structured way. This means, that also objects - in form of a JSON String - can be

logged. With sinks extensions it is possible to either store data on the file system or send it to central repositories. In the following code snipped the configuration of the Logging Service is shown.

```
_Log = new LoggerConfiguration()
.MinimumLevel.ControlledBy(levelSwitch)
.Enrich.FromLogContext()
.Enrich.WithCaller()
.WriteTo.Console(outputTemplate: outputTemplate,
        theme: SystemConsoleTheme.Literate)
.WriteTo.Elasticsearch(
        new ElasticsearchSinkOptions(
                new Uri("http://localhost:9200"))
                {
                IndexFormat = cMyProcessName + "-serilog-{0:yyyy.MM}",
                })
.CreateLogger();

_Log.Verbose("Logging started");
```

Each log message was enriched with the following fields:

- Hostname: To make it easier to find the respective machine where the Microservice is running on
- Topic: The Process Name
- CallerClass: the Class where the Log Message was generated
- Caller: the Caller method where the message was generated

With these additional fields, filtering of certain sources or roots is easier. In Figure 11 the resulting log messages in Kibana can be seen.

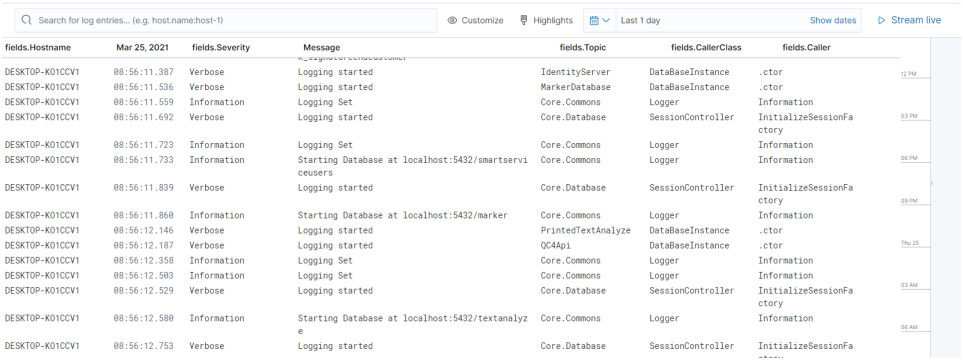

Figure 11: Serilog messages in Kibana

6.2.2 Identity Management

In terms of Microservice Security as basis the "IdentityServer4" (see [IdendityServer4, 2021]) was used, which is an open-source OpenID Connect and OAuth 2.0 framework. On database side again an PostgreSQL database server was used. The passwords are stored with the library bcrypt (see [Bcrypt, 2021]). Whenever a request to the Identity Service is sent, the identity service returns an token in JsonWebToken (JWT) Format. In this token the following information - within other - is sent to the client:

- Expiration Time
- Authentication Time
- Issue Time
- Scopes
- Refresh Token

In each of the requests to the Microservices, the Token has to be added to header in terms of Authorization: Bearer <Token>. If the token is no more valid or not available, each endpoint returns the HTTP Status code 401 for unauthorized. The following code snippet shows how it is ensured, that the JWT token is used in a respective Microservice:

```
services.AddAuthentication(options =>
{
  options.DefaultScheme = JwtBearerDefaults.AuthenticationScheme;
  options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
  options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
  options.TokenValidationParameters = new TokenValidationParameters
{
  ValidateAudience = false,
  ValidateLifetime = true,
  LifetimeValidator = (notBefore, expires, securityToken, validationParameter) =
  expires >= DateTime.UtcNow
};
  options.Authority = secusettings.IdentityUrl;
});

...

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  ...
  app.UseAuthentication();
  app.UseAuthorization();
```

6.2.3 External Services

Some of the Microservices implemented are getting their information from external systems. Those systems are:

- Technical Database: In this database design parameters for transformers are stored. E.g.: Type of Coolers, Bushings,...
- Transformer Assembly Manager: In this database information about the Transformer assembly is stored. E.g.: Detailed Outline Drawings, Amount of Coolers, Bushings,....
- Text Analyze (Azure): The OCR text analyze is used from Azure in order to convert long serial - numbers into machine readable characters.

6.3 API Gateway

As API Gateway Consul.io was used and installed on an virtual machine with Ubuntu as OS. In this case consul acts as Service Registry and API Gateway. All above mentioned Microservices were added.

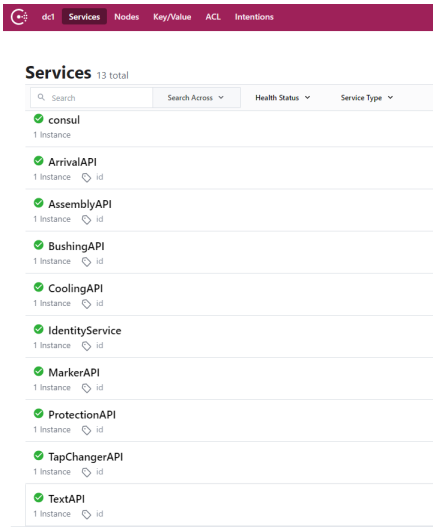

Figure 12: Microservices in Consul

Additionally routes were created: if a request path contains the service name, it will be forwarded to the respective service. e.g.: A request at the API Gateway to http://<ip>/arrivalapi/something will be directly forwarded to the Arrival API. For each Mircoservice two checks were implemented:

- SSH TCP Check on Port: This check simply verifies, if a respective port is open and reachable. This also implies that the machine or the container is at least reachable.

- HTTP check on "http://<ip>/health": Each Microservice has an endpoint which responds its current status with some metrics in regard to the service and the environment. These metrics can then be used for further monitoring. The fields are:
  - Memory consumption
  - Hostsystem
  - Machinename
  - Uptime in minutes
  - Each Disk with capacity, free space and free space in percent.

Figure 13: Microservice Health Check in Consul

6.4 Documentation

The documentation of each Microservice is done with Swagger. This documentation is only available internally and not visible via the API Gateway. Nevertheless this provides an easy and straight forward way to test the respective endpoints during development. In order to provide this sort of documentation, the following Nuget Packages were used:

- Swashbuckle.AspNetCore
- Swashbuckle.AspNetCore.Swagger.UI
- Swashbuckle.Core

With the following code snippets, each endpoint in a Controller is automatically parsed into an Swagger endpoint with the respective parameters and documentation:

```
services.AddSwaggerGen(c =>{
  c.SwaggerDoc("v1", new Microsoft.OpenApi.Models.OpenApiInfo
  {
    Title = "Marker API",
    Version = "v1",
```

```
    Description = "API for Annotating Images"
  }
  );
}

...

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
  app.UseStaticFiles();
  app.UseSwagger();
  app.UseSwaggerUI(c =>
  {
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Marker API");
    c.RoutePrefix = String.Empty;
  }

  ...
);
```

The output inside a browser window is shown in Figure 14.

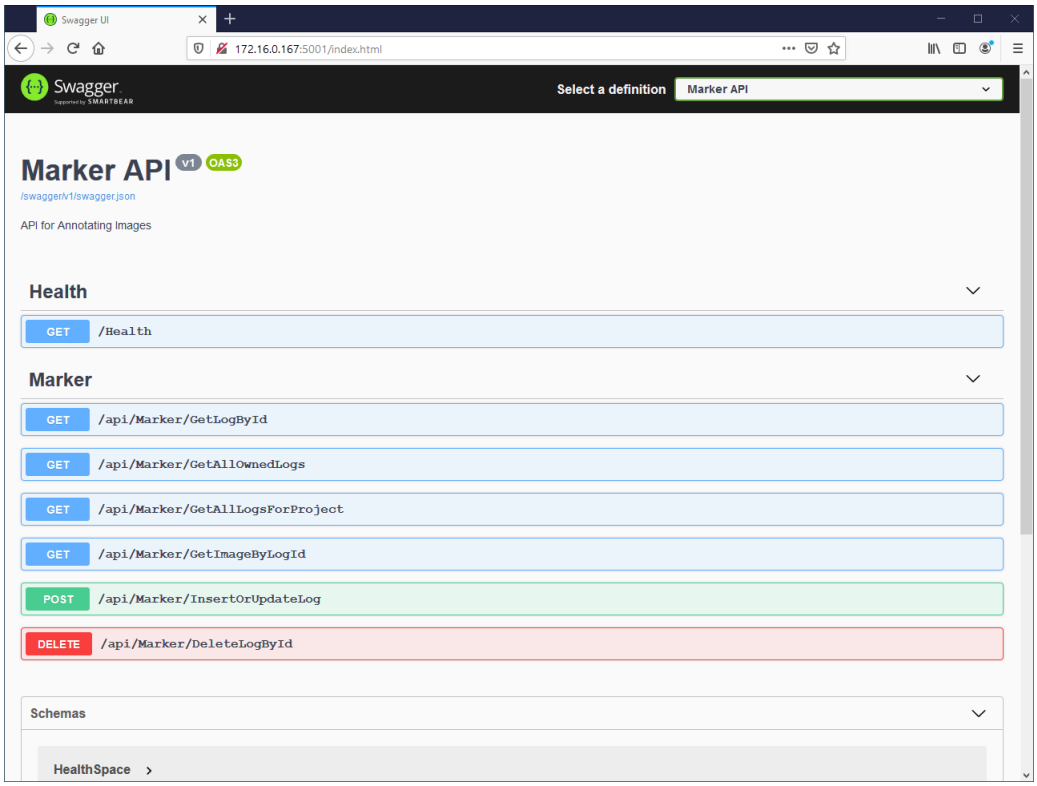

Figure 14: API documentation with Swagger

7 Conclusio

As an more and more accepted and adopted architectural style, MSA overcomes the limitations of the traditional monolithic architecture. During implementation of this proof of concept a lot of refactoring was necessary in order to meet all the desired quality attributes. First of all it is hard to find the correct granularity for the Microservices. In the first approach it was more or less a monolith, which was caused due to a implementation start without a clear big picture, no strategy and no structure. Without those, the implementation will surely lead into anti-pattern and architectural smells.

After the final definition of the entire working process of the supervisors at site, those process steps were meant to be a single Microservice. This process was identified with the user centered design approach and through interviews with the users which will later work with the system. In order to ensure a proper code quality, test driven development with NUnit was used for both the model and RESTful interface.

With all those Microservices in place, it could be seen that the Logging into

files was not the best solution to debug. Therefore the interface to the Elastic Stack was implemented. After that it could be seen, that the Identity Managment should be taken into account right from the beginning of the programming. The integration of the Identity Mangement into already existing Microservices took a lot of work and refactoring. If it is integrated from scratch, the work which needs to be spend, will decrease significantly.

Since the CaRE System is a proof of concept Scalability, Availability and Performance were not taken into account in terms of the Microservices. At User Interface level the performance of the entire application was of course measured during the usability tests. In case of unavailability of the Microservices, changes on UI level were buffered and later synchronized with the Microservice.

Sometimes it makes sense to make use of already existing services rather than implementing them again. As an example in the CaRE architecture the Text-Analysis needs to be mentioned. If - for example - a system requires language processing, it also makes sense to use already existing services.

Additionally the integration of the external services - Technical Database and Transformer Assembly Management - needs to be mentioned. Due to data security and restrictions within Siemens, it was hard to get access to those services, even if they are crucial for the entire application. The company itself feared data breaches and / or data loss, since some design specific data is transmitted outside the Siemens Intranet. In order to come over that, only read access to well defined endpoints was granted, which was also done with JWT Tokens and defined scopes. Intentionally it was planned to also integrate the drawings database directly to the system. Unfortunately this was prohibited, so a simple file upload was implemented for this prove of concept.

To summarize it needs to be mentioned that a clear understanding of the big picture is necessary before starting with implementation, otherwise a lot of refactoring needs to be done. Furthermore the best solution must be found for each individual project. Solutions of big players like Netflix, Facebook and so one sometimes do not really fit for small applications and vice versa. Even if the usage Microservice architecture is emerging, it is not always the best choice and also needs to be investigated carefully.

References

[Abad et al., 2018] Abad, C. L., Boza, E. F., and van Eyk, E. (2018). Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE 18, page 101, New York, NY, USA. Association for Computing Machinery.

[Abbott, 2015] Abbott, M. L. (2015). The art of scalability : : scalable web architecture, processes, and organizations for the modern enterprise.

[Alenezi, 2016] Alenezi, M. (2016). Software architecture quality measurement stability and understandability. *International Journal of Advanced Computer Science and Applications*, 7:550–559.

[Bass et al., 2012] Bass, L., Clements, P., and Kazman, R. (2012). *Software architecture in practice*. SEI series in software engineering. Addison-Wesley Professional.

[Bcrypt, 2021] Bcrypt (2021). https://github.com/bcryptnet/bcrypt.net.

[Celesti et al., 2017] Celesti, A., Carnevale, L., Galletta, A., Fazio, M., and Villari, M. (2017). A watchdog service making container-based micro-services reliable in iot clouds. In *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 372–378.

[Cojocaru et al., 2019] Cojocaru, M., Oprescu, A.-M., and Uta, A. (2019). Attributes assessing the quality of microservices automatically decomposed from monolithic applications. pages 84–93.

[Crispin and Gregory, 2009] Crispin, L. and Gregory, J. (2009). *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 1 edition.

[Eberhard, 2018] Eberhard, W. (2018). *Das Microservices-Praxisbuch: Grundlagen, Konzepte und Rezepte*. dpunkt.verlag.

[Febrero et al., 2016] Febrero, F., Calero, C., and Moraga, M. (2016). Software reliability modeling based on iso/iec square. *Information and Software Technology*, 70:18–29.

[Fetzer et al., 2017] Fetzer, C., Mazzeo, G., Oliver, J., Romano, L., and Verburg, M. (2017). Integrating reactive cloud applications in sereca. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, New York, NY, USA. Association for Computing Machinery.

[Filip et al., 2018] Filip, I., Pop, F., Serbanescu, C., and Choi, C. (2018). Microservices scheduling model over heterogeneous cloud-edge environments as support for iot applications. *IEEE Internet of Things Journal*, 5(4):2672–2681.

[Fowler, 2020] Fowler, J. L. M. (2020). Microservices - a definition of this new architectural term.

[Furda et al., 2018] Furda, A., Fidge, C., Zimmermann, O., Kelly, W., and Barros, A. (2018). Migrating enterprise legacy source code to microservices: On multi-tenancy, statefulness and data consistency. *IEEE Software*, 35.

[Galin, 2018] Galin, D. (2018). Software quality concepts and practice.

[Haselboeck and Weinreich, 2017] Haselboeck, S. and Weinreich, R. (2017). Decision guidance models for microservice monitoring. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 54–61.

[Haselboeck et al., 2017] Haselboeck, S., Weinreich, R., and Buchgeher, G. (2017). Decision guidance models for microservices: Service discovery and fault tolerance. pages 1–10.

[IdentityServer4, 2021] IdentityServer4 (2021). https://identityserver4.readthedocs.io/en/latest/.

[ISO, 2011a] ISO (2011a). Iso/iec 25010 systems and software engineering: Systems and software quality requirements and evaluation (square) - system and software qualitymodels. Technical report, ISO / IEC / IEEE.

[ISO, 2011b] ISO (2011b). Iso/iec 25023 systems and software engineering - systems and software quality requirements and evaluation (square) - measurement of system and software product quality. Technical report, ISO / IEC.

[ISO, 2013] ISO (2013). Iso/iec/ieee 29119-1:2013 software and systems engineering - software testing - part 1: Concepts and definitions. *ISO / IEC / IEEE*, page 56.

[ISO, 2018] ISO (2018). Iso/iec/ieee 90003:2018 - software engineering: Guidelines for the application of iso 9001:2015 to computer software. Technical report, ISO / IEC, Genf.

[Jamshidi et al., 2018] Jamshidi, P., Pahl, C., MendonÃ§a, N., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35:24–35.

[Jander et al., 2019] Jander, K., Braubach, L., and Pokahr, A. (2019). Practical defense-in-depth solution for microservice systems. *Journal of Ubiquitous Systems and Pervasive Networks*, 11:17–25.

[Kakivaya et al., 2018] Kakivaya, G., Modi, V., Mohsin, M., Kong, R., Ahuja, A., Platon, O., Wun, A., Snider, M., Daniel, C., Mastrian, D., Li, Y., Xun, L., Rao, A., Kidambi, V., Wang, R., Ram, A., Shivaprakash, S., Nair, R., Warwick, A., and Fussell, M. (2018). Service fabric: a distributed platform for building microservices in the cloud. pages 1–15.

[Kang et al., 2016] Kang, H., Le, M., and Tao, S. (2016). Container and microservice driven design for cloud infrastructure devops. pages 202–211.

[Katkoori, 2019] Katkoori, P. (2019). Application architecture: Monolithic vs soa vs microservices.

[Khazaei et al., 2020] Khazaei, H., Mahmoudi, N., Barna, C., and Litoiu, M. (2020). Performance modeling of microservice platforms. *IEEE Transactions on Cloud Computing*, pages 1–1.

[Li et al., 2021] Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., and Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A systematic literature review. *Information and Software Technology*, 131:106449.

[Mayer and Weinreich, 2018] Mayer, B. and Weinreich, R. (2018). An approach to extract the architecture of microservice-based software systems. In *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 21–30.

[microservices.io, 2021] microservices.io (2021). The scale cube. Web.

[Montesi and Weber, 2018] Montesi, F. and Weber, J. (2018). From the decorator pattern to circuit breakers in microservices. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC18, pages 1733–1735, New York, NY, USA. Association for Computing Machinery.

[Mordal-Manet et al., 2013] Mordal-Manet, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., and Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 25:1117–1135.

[Nemer, 2019] Nemer, J. (2019). Advantages and disadvantages of microservices architecture. Webpage.

[Netflix, 2021a] Netflix (2021a). Ribbon. Webpage Github.

[Netflix, 2021b] Netflix (2021b). Vector. *https://github.com/Netflix/vector*.

[Newman, 2019] Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly.

[Nicol et al., 2018] Nicol, J., Li, C., Chen, P., Feng, T., and Ramachandra, H. (2018). Odp: An infrastructure for on-demand service profiling. pages 139–144.

[Otterstad and Yarygina, 2017] Otterstad, C. and Yarygina, T. (2017). Low-level exploitation mitigation by diverse microservices. pages 49–56.

[Pardon and Pautasso, 2017] Pardon, G. and Pautasso, C. (2017). Consistent disaster recovery for microservices: the cab theorem. *IEEE Cloud Computing*, PP:1–1.

[Permikangas, 2020] Permikangas, T. (2020). Digital ecosystem development: The future of integration environment.

[Pigazzini et al., 2020] Pigazzini, I., Fontana, F. A., Lenarduzzi, V., and Taibi, D. (2020). Towards microservice smells detection. In *Proceedings of the 3rd International Conference on Technical Debt*, TechDebt 20, pages 92–97, New York, NY, USA. Association for Computing Machinery.

[Ren et al., 2010] Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., and Hundt, R. (2010). Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE Micro*, pages 65–79.

[Rusek et al., 2017] Rusek, M., Dwornicki, G., and Orlowski, A. (2017). A decentralized system for load balancing of containerized microservices in the cloud. volume 539, pages 142–152.

[Sam, 2015] Sam, N. (2015). *Microservices (mitp Professional): Konzeption und Design*. mitp-Verlag, 1 edition.

[Sun et al., 2015] Sun, Y., Nanda, S., and Jaeger, T. (2015). Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 50–57.

[Taibi and Lenarduzzi, 2018] Taibi, D. and Lenarduzzi, V. (2018). On the definition of microservice bad smells. *IEEE Software*, vol 35.

[Talend.com, 2020] Talend.com (2020). Microservices vs. soa: Was ist der unterschied?

[Tighilt et al., 2020] Tighilt, R., Abdellatif, M., Moha, N., Mili, H., El-Boussaidi, G., Privat, J., and GuÃ©hÃ©neuc, Y.-G. (2020). On the study of microservices antipatterns: a catalog proposal. pages 1–13.

[Torkura et al., 2017] Torkura, K., Sukmana, M. I. H., and Meinel, C. (2017). Integrating continuous security assessments in microservices and cloud native applications.

[Walsh and Manferdelli, 2017] Walsh, K. and Manferdelli, J. (2017). Mechanisms for mutual attested microservice communication. pages 59–64.

[WikiWikiWeb, 2021] WikiWikiWeb (2021). Wiki wiki web.

[Wittmer, 2019] Wittmer, P. (2019). Microservice disadvantages and advantages. Web.

[Yarygina and Otterstad, 2018] Yarygina, T. and Otterstad, C. (2018). *A Game of Microservices: Automated Intrusion Response*, pages 169–177.

[Zimmermann, 2016] Zimmermann, O. (2016). Microservices tenets. volume 32, pages 301–310. Springer Science and Business Media LLC.