Macro Architecture for Microservices: Improving Internal Quality of Microservice Systems

Lars Braubach, Kai Jander, and Alexander Pokahr

Abstract Microservices have established themselves as a common software engineering pattern for large-scale systems. However, while the focus of the approach on maximum flexibility for development of individual services has increased software development efficiency, the inter-service architecture of microservice-based systems has received little attention, resulting in systems with a multitude of confounding and poorly planned interactions between services, impeding initial development, maintenance and continued development of applications. This paper identifies three areas that currently require improvement in order to address the global architectural challenges of microservice systems. Furthermore, a solution is proposed for each of the areas with an implementation demonstrating how the proposed solution can improve development efficiency for and internal quality of microservice systems.

1 Introduction

Microservices are currently one of the most favored approaches for the development of large-scale service-oriented architecture (SOA)-based applications. By vertically slicing an application into microservices, each responsible for its own data, logic and sometimes also user interface layer, allows for rather independent development and enhancements of the services. This is further organizationally supported by having small teams that develop the services in a DevOps manner and in this way can release frequently and react to changing requirements very quickly. Even

L. Braubach (✉)
Bremen City University of Applied Sciences, Bremen, Germany
e-mail: lars.braubach@hs-bremen.de

K. Jander
Brandenburg University of Applied Sciences, Brandenburg, Germany
e-mail: jander@th-brandenburg.de

A. Pokahr
Actoron GmbH, Hamburg, Germany
e-mail: ap@actoron.com

D. Camacho et al. (eds.), *Intelligent Distributed Computing XIV*, Studies in Computational Intelligence 1026, https://doi.org/10.1007/978-3-030-96627-0_10

though these advantages led to widespread adoption, the global architecture level of microservice systems usually overlooked. This leads to complex configurations, unclear responsibilities and dependencies among services. Architectural deficiencies are considered to result in reduced internal quality for software. Internal quality in software engineering refers to the ability to continue development on a project while external quality deals meeting customer requirements [5]. As a result, internal quality is key from a development perspective as it directly influences development efficiency. In this paper, we will identify three important architectural challenges of current microservice approaches and address them with the goal of improving internal software quality.

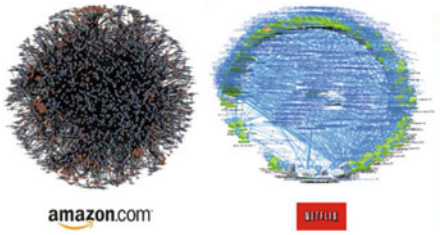
In the next Sect. 2 the challenges are put forward each also discussing relevant related work explaining current solutions. Thereafter, in Sect. 3, an implementation for the approach is presented and evaluated using practical examples. Finally, in Sect. 4 concluding remarks and a short outlook on future work is given.

2 SOA Challenges and State of the Art

Microservices are an architectural pattern aiming at supporting scalability on multiple levels. E.g., on a technical level functional decomposition allows for separate scaling and load-balancing of each functionality. On an organizational level independent DevOps teams that can use their preferred programming languages and tools and new technology can quickly be adopted for new modules without having to migrate existing ones. The independence of microservices compared to, e.g., modules of a monolithic application, thus provides many benefits but also comes with a number of drawbacks (cf. e.g. [15]) that pose challenges when developing microservice applications. In the following three important developer-oriented challenges will be discussed.

2.1 Inter-service Architecture Challenges

In microservice architectures generally no restrictions are placed on visibility of services, e.g. every microservice is technically capable of invoking any other microservice. While this approach grants a large degree of flexibility and freedom for implementing each microservice, it often leads to a confusing global architecture that is difficult to analyze and debug. This issue is often demonstrated using Deathstar visualizations (see Fig. 1a), which shows interactions between individual microservices with services in a spherical shape. The resulting overview shows almost no structure at the global application level. While the Deathstar architecture style sometimes argued as the price for flexibility, special management approaches are required to deal with the high complexity of the system [14].



Maintainability (ISO 25010)	Intra-Service Level (Local)	Inter-Service Level (Global)
Modularity	+ (package / module mechanisms)	= (coherence and coupling of services, decomposition problems)
Reusability	+ (libraries, helper classes)	= (granularity and domain orientation of services, decomposition problems)
Analysability	+ (low complexity)	- (no explicit system-level architecture)
Modifiability	+ (low complexity)	- (unclear service dependencies and versions)
Testability	+ (limited functionality and scope)	- (integration and end-to-end tests)

Fig. 1 **a** "Deathstar" visualization of the Netflix microservice architecture (from [2]) and **b** Maintainability challenges

Figure 1b further illustrates challenges to important aspects of maintainability according to the ISO 25010 software quality standard [6] when comparing intra- and inter-service levels. Regarding intra-service levels, good solutions for all aspects are available, while at the inter-service level remain challenging. To achieve modularity and reusability for a single service, all well-known software engineering mechanisms like libraries are applicable. Analysability and modifiability can be attained easily due to the low complexity of a single microservice (it is simple by definition). In addition, for testability the scope is narrow (unit tests and service interface tests).

The inter-service level is far more demanding (cf. e.g. [15]). Modularity depends on the resulting coherence and coupling of services. Even if developers identify perfect services, they are missing a hierarchical decomposition concept, so service use and reasons for it remain unclear. Reusability primarily depends on granularity and domain orientation. If done right, services are flexible to orchestrate. However, with great number of services, it becomes difficult to identify services to be combined due to non-existing grouping and/or composition methods. Analysability remains ok for small systems but partially falls apart as the system grows. Such evolution also harms the modifiability of the system. Typically, microservice systems are said to be easily extendable by adding services implementing desired customer requirements. However, when functionality of services become intertwined and services are not removed, systems tend to be prone to the lava flow antipattern [1] leaving unused code and thus blurring the architecture. Finally, testability of microservices is hard at the integration and end-to-end layer [11]. Besides non-SOA-typical problems like increased setup complexity and time consumption, responsibility problem arise - who is responsible for tests and when are they executed? If, e.g. the tests are done as part of a build pipeline, how can buggy services avoid disturbing building others service? If mocks reduce the problems somewhat, a new problem arises that original and mocked service must be in sync. This led to partially drastic solutions in practice such as omitting tests altogether in favor of testing with synthetic transactions [11].

Few approaches for developing inter-service level architectures exist so far. In [9] the "monolith first" idea is mentioned, which suggests a monolith prototype before deriving the architecture. This requires additional development effort and it remains unclear if results are transferable (e.g. monolithic layered architectures slice systems vertically instead of horizontally), though suggestions exist on migration,

	REST	RPC	Publish Subscribe	Service Mesh
Implementations	Jersey, Express	gRPC, Thrift	Kafka, ActiveMQ, Mule	Istio, linkerd, Consul
Interface Expressiveness	CRUD	full, defineable via IDL	events, no interface	CRUD
Typing	weak	strong	weak	weak
Infrastructure	decentralized	decentralized	centralized	centralized
CallDistribution/Load Bal.	no	no	yes	yes
Service Discovery	necessary	necessary	not necessary	not necessary
Request-Reply	yes	yes	indirectly yes	yes
Subscription	yes (via long polling)	typically no	yes	typically no
Complex Interactions	no	no	yes	no

Fig. 2 Comparison of interaction approaches

e.g. [12]. Domain Driven Design (DDD) [3] is available for migration and greenfield projects, allowing for decomposing a system into so-called bounded contexts, which often allow straight-forward mapping to services. While DDD is useful and often an important success factor for microservice systems [11], its approach does not consider which services should exist or interact. In a whitepaper from MuleSoft,[1] six important micro service design patterns have been identified. One is called "layered APIs over fine-grained SOA" and proposes to layer services in order to cope with increasing complexity. Although service composition may lead to slight performance penalties, a layered API can lead to simpler designs due to clear scopes and responsibilities of services.

2.2 Interaction Challenges

Interactions between services are challenging because there are different communication means available each with different characteristics. The common approaches are shown and compared with respect to several dimensions in Fig. 2. Considering interface expressiveness, RPC allows for using the most natural style well-known and established in object-oriented systems. Typically, a programming language independent interface definition language (IDL) is employed that allows for specifying methods with parameter types and return values in the spirit of CORBA [13] thus ensuring strong typing of interfaces. IDL descriptions are typically used to automatically generate code stubs for the specific implementation languages. In contrast to RPC, REST has a so called uniform interface [4], which uses CRUD (create, read, update, delete) semantics with reinterpreted HTTP methods on resources. Although many use cases can directly be implemented based on that scheme it is a restriction of expressiveness and for some use cases workarounds such as virtual resources or extra resource properties have to be employed. In addition, as REST uses HTTP, parameter and response types remain rather implementation than specification aspects, even though approaches like OpenAPI improve the situation. As today, service mesh middleware relies on REST the same restrictions apply for them as well. Using publish-

[1] https://www.mulesoft.com/lp/whitepaper/api/top-microservices-patterns.

subscribe approaches like message brokers or ESBs, there are no interfaces at all. Instead, event types need to be specified and service consumption and production of event types must be documented.

REST and RPC are decentralized solutions that require no middleware being present. Instead, communication is performed directly between a service provider and a consumer. This is in contrast to middleware-oriented solutions like message brokers and service meshes. Though some implementations of brokers allow for redundant functionalities [10], system communication is nevertheless transmitted using a dedicated middleware infrastructure instead of point-to-point communication. While it sounds like the decentralized solutions are advantageous due to avoiding a single point of failure and allowing for a simpler system setup, this only partially true. Middleware solutions simplify service discovery and may offer features like automatic load balancing or recovery of services by implementing robustness patterns like circuit breaker.

Considering the possible interaction patterns, publish subscribe is the most generic solution allowing all sorts of protocols being realized as sequence of events (including request-reply and subscriptions). As only events are distributed, services become less coupled and do not need to know which other services depend on them. Although direct coupling is lower, the services still depend on specific types of events being produced, which makes coupling more indirect and less visible on a system level. Thus, changing services with respect to event generation will not lead to errors on the compile level but defer problems to the runtime layer and possibly harm to process execution. Using REST or RPC emphasizes the request-reply interactions and typically needs extensions or workarounds to handle more complex patterns like subscriptions. Especially problematic are protocols including multiple steps like e.g. auctions. Those require interfaces with methods for the different protocol steps and additionally imply that those methods have to be called in the right order for the protocol being properly executed.

2.3 Service Discovery

Getting microservices to know each other is a surprisingly hard challenge. Figure 3 compares common discovery approaches with respect to important developer characteristics, more specifically, the following approaches are taken into consideration: Hard coding addresses means that no discovery mechanism is used at all and services must be known and are selected at build time. Registry approaches come in different two variants: Simple data stores typically are key value based and are primarily general purpose but are sometimes used for service registration. Service registries represent software specifically developed to store and query service data.

Also event based approaches come in two different flavors: Message brokers realize a publish subscribe infrastructure while ESBs add further integration functionalities on top of communication layer. These approaches strongly emphasize decoupling services, allowing communication without explicitly knowing the partic-

	No Discovery	Simple Data Store / Registry	Message Broker / ESB	Mesh
Implementations	-	Etcd, ZooKeeper / Eureka, Consul	Kafka, ActiveMQ / Mule, BizTalk	Istio, linkerd, Consul
Programming model	Hardcoding service addresses in code	Send queries and directly interact with services	Send events to indirectly interact with services	Invoke side car services via hardcoded names
Simple and concise prog. experience	-	-	-	=
High expressiveness for service selection	-	- / +	+	-
High invocation performance	+	+	=	=
Further features available (load bal.,..)	-	-	- / +	+

Fig. 3 Comparison of discovery challenges

ipants. Instead, services require knowledge which events are produced and consumed by other services. Finally, a service mesh uses so called side-cars for each service which represent proxies attached to a middleware routing invocations to suitable services.

Each of these approaches appear to exhibit different strength and weaknesses: For example, only dedicated registries and ESB offer the means for selecting services according to complex criteria. Meanwhile, service mesh approaches instead tend to resolve a service name to a service address, resulting in a rather implicit approach for service selection. Service invocation performance is good for all approaches directly invoking selected services, except for ESB and mesh where each call is routed indirectly via middleware to the target, leading to some performance penalty compared to direct calls. However, this minor disadvantage is usually accepted because middleware approaches like ESB and mesh allow for integration of further functionalities such as load-balancing, monitoring, tracing or fault detection and recovery via circuit breakers without burdening developers with separate infrastructure for each of these tasks.

Each of the approaches result in a particular SOA programming model used by programmers to develop systems based on these approaches: Hardcoding service addresses in code requires little to no infrastructure or setup, however, the code quickly becomes unmaintainable. For example, a single service address change could instantly break the whole system. On the other hand, registry-based approaches require developers to first issue queries to the registry whenever a service is needed (a.k.a. “SOA Triangle”). While this may sound simple, the resulting code quickly becomes complex due to edge cases such as a query returning no results. In general, temporarily absent services are an unavoidable fact for distributed system due internal errors or network partitions, but also in situations like a system startup, where services depend on each other but have varying initialization times. As a result, developers are forced to implement repeated but temporarily spaced service searches within an appropriately chosen grace period in order to guarantee robust behavior in which the

service can either be acquired in a timely manner or it aborts occurs after a defined number of retries.

For message-broker-based systems, the SOA programming model becomes event-oriented instead of service-oriented. Instead of APIs, service invocation occurs as reaction to consumed events. Proponents of this approach often claim that using an event broker leads to loosely coupled systems and fosters independent service development. However, while services do not require direct knowledge of each other, this direct relationship is instead substituted by requiring knowledge of event types that are emitted and consumed. As a result, the API of a service is represented by a set of consumed and produced events, often hidden in code (sometimes as literals). Furthermore, if events are used by multiple services, changes to events could easily break a system without clear indication regarding the cause. As a result, clear responsibilities for event definitions have to be established among teams.

In contrast to the other approaches, a mesh simplifies the programmer experience by trying to strike a balance: Services are invoked by name but dynamically resolved by the infrastructure. The model assumes that services under the same name are always replicas and developers do not need to care which one is invoked. When this is not the case, workarounds have to be found, e.g. by using different service names for the same type of service.

2.4 Implications

First, many microservice approaches have a weak or incoherent inter-service architecture. This is e.g. illustrated by the strong focus on single service development, while there are very few proposals for modeling or designing the top-level SOA architecture of a microservice system, leading to a number of undesirable consequences such as unclear dependencies and unintended and unnecessary retention of obsolete services and code. As a result, an approach providing *concepts for modeling the global architecture* while retaining the flexibility and low-threshold implementation style of current microservice systems is desirable.

Second, while multiple approaches for service interactions are available, however, each is limited in its own way. For example, approaches based on Remote Procedure Call (RPC) and Representational State Transfer (REST) rarely provide more than simple request-response type interactions and, if provided, the more complex interaction types are limited. While approaches such as Enterprise Service Bus (ESB) provide more flexibility in terms of interaction complexity and loose coupling, the interactions tend to be implicit based on event types leading to unclear interaction patterns. Additionally, using an ESB the API of services cannot easily be seen anymore and dependencies between services becomes buried in the implementations. Thus, an approach is lacking that supports *type-safe and comprehensible complex interaction patterns*.

Finally, service discovery approaches either focus on transparency or control over the service selection. E.g. in ESB and service meshes, service invocations are

largely transparent to the client developer and support sophisticated features such as elasticity and circuit breakers. On the other hand, when the client developer wants more control over the service selection, she has to resort to, e.g., hard-coded addresses or manual lookups using DNS or registries resulting in tedious and error-prone code that explicitly has to deal with e.g. partial failures and mostly lacks the advanced features of ESB and service meshes. Therefore, an approach is needed to *combine the expressive power of manual service selection with automatically handling issues of distributed systems, such as race conditions, partial failures, etc*.

3 Concepts and Implementation

Figure 4 provides an example how the novel approach improves microservice architectures by addressing the outlined shortcomings of current approaches in a combined manner. In general, our contribution is based on three major changes:

- Services can be organized hierarchically with encapsulation support using publication scopes
- Services are automatically retrieved using persistent queries with minimal code
- Communication between services is based on strongly typed asynchronous RPC calls also supporting more complex interaction forms

The example demonstrates an application providing logistics services to external customers via a *Logistics* service type. Meanwhile, the logistics application relies on internal services to provide its functionality. These internal services represent three different types of services: The *Dispatch* type for organizing parcel pickup, the *Delivery* type for transporting the parcel to its destination and *Monitoring* for logging events and monitoring system performance. The service types are backed by service instances, which provide the actual service functionality for each type of service. Multiple instances of each service type can be available: For example, there are two instances of the delivery service, *log.del1* and *log.del2*, both being encapsulated by the instance of the logistics application *log*. Multiple service instances can be either

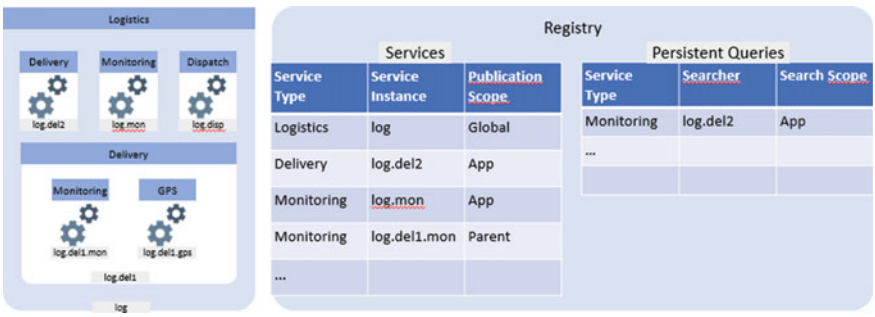


Fig. 4 Approach

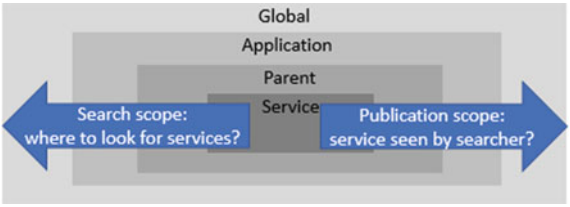


Fig. 5 Search and publication scopes

used to provide redundancy for resilience or horizontal scaling, or, alternatively, to provide service reuse in different contexts: The service instance *log.del1.mon* provides a separate monitoring service as internal service for the delivery service instance *log.del1*. Each service instance can limit visibility of its functionality through the use of publication scopes: For example, *log.del2* sets its publication scope to *App,* only search queries issued by service instances that are part of the *log* application will match. Furthermore, *log.del2* needs a monitoring service and has thus issued a persistent query to the registry for that type. It can be seen in the Figure that the search scope of that query is App, meaning that all services visible in the log application are potential candidates. In this case only *service log.mon* will be found, because the publication scope of *log.del1.mon* is *Parent*, which makes the service invisible outside of the delivery context.

In Fig. 5 the concept of scopes is visualized. While *Service* scope represents the local view of a service, *Parent* scope also allows sibling service visibility. *Application* scope represents all services belonging to a complete application while global scope allows services of different applications to find a service. The figure also illustrates the difference between search and publication scopes. The first is used to limit the search for services, whereas the latter acts as a visibility restriction and determines if a service is visible to a searching service.

Next, the programming model of persistent queries will be explained in more detail. Issuing a persistent query to the registry has the effect that the registry saves the query and automatically tests newly added services whether they match the query. When a match is found, the searcher will be notified, which can be realized very conveniently in a programming model using callback methods that initiate processing for matching new services. Taking this further, our model also allows implicitly defining queries by labeling (callback) methods with an @OnService annotation containing query content such as the search scope. The service type is also implicitly derived from the method parameter. In Fig. 6, this is illustrated with a code snippet used to set up a persistent query for monitoring services in our example scenario. Aside from low overhead, the programming model also exhibits further advantages such as more fine-grained control of the search, e.g. by specifying tags attached to a service or by using quality of service attributes that have to be met. In addition, recovery from service errors is simplified: If multiple services meet a query, the method will be invoked for each discovered instance which can be stored and used as backup services. Even if no service is currently available, the method is called immediately once one becomes available (without further programming effort).

```
@OnService(scope="App")
public void foundMonitoring(Monitoring mon) {
  // handle the newly found service, e.g. store in variable
}
```

Fig. 6 Persistent query example

Due to the layered architecture, service interfaces are a vital factor as they can describe not just micro element functionality (i.e. services) but subsystems and whole applications (otherwise the facade pattern via a web gateway is often used). As a result, a strongly-typed approach is chosen akin to gRPC. All methods declared as asynchronous using future return values. This would normally restrict calls to request-reply interactions, however, an extended set of futures types supporting more complex interaction patterns within the context of one call is made available. For example, a subscription future can be used, which is able to retrieve a potentially unbounded stream of values. In addition, iterator futures allow transmission of multiple values but invert responsibilities by allowing receivers to fetch on request.

Communication between services is facilitated without additional explicit programming effort and little configuration using an overlay network with multiple communication approaches (TCP, WebSockets, ...) and optionally using third-parties as intermediaries to facilitate communication between two isolated services. Routes between services are discovered dynamically and automatically. Data is serialized using a compact binary form (alternatively JSON) and is end-to-end encrypted by default, authenticated using a variety of schemes or opportunistically, allowing for secure communication between services even when interacting through third parties [7].

A running example of the approach has been implemented as part of the Jadex Active Components framework.[2] A number of examples demonstrating the approach are included in the framework, including an example and tutorial showing the approach for a time server system.[3] A global service registry is available and configured by default, however, users can deploy and configure their own registry system, if desired. Details about the registry architecture and realization can be found in [8].

4 Conclusion and Outlook

In this paper three important challenges with respect to microservice systems have been identified: First, the software architecture is typically based on a flat set of a potentially huge number of services, leading to Deathstar antipattern structures. Second, interaction is facilitated using different schemes, each exhibiting their own drawbacks and third, service discovery is either explicit with error prone trial and

[2] https://www.activecomponents.org/.

[3] https://github.com/actoron/jadex/blob/master/docs/tutorials/quickstart/01%20Introduction.md.

error programming models or implicit with a simple programming model but no control over service selection.

To solve these issues an integrated approach has been presented based on **layered services** and a simple, yet fully controllable programming model using **persistent queries**. Decomposition in service layers allow for hierarchical system structures, in which top-level services comprise the system level API, while lower-level services represent subsystems and provide partial functionalities for higher-level ones. The service discovery supports this model by offering publication and search scopes discovery of contextually meaningful services. In addition, persistent queries allow processing and discovering services the moment they become available. This simplifies the programming model for developers by retaining the expressive power of manual queries and combining it with easy handling of typical distributed system issues. Finally, interaction of services is based on **asynchronous RPC-calls** extended with capabilities for directly handling complex interaction types such as subscriptions and iterators. The proposed concepts have been implemented within the Jadex framework.

Currently, the implementation is limited by requiring that services to be implemented in Java, therefore limiting the use of heterogeneous technology stacks for different services. As future work we intend to change this by introducing a polyglot API similar to the approach used by vert.x [4] and REST-based access for the registry.

References

1. Brown, W.J., Malveau, R.C., "skip" McCormick, H.W., Mowbray, T.J.: AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis: Refactoring Software, Architecture and Projects in Crisis. Wiley (1998)
2. Cockcroft, A.: Microservices workshop: Why, what, and how to get there (2016)
3. Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
4. Fielding, R.T.: Architectural styles and the design of network-based software architectures. Ph.D. thesis (2000). AAI9980887
5. Freeman, S., Pryce, N.: Growing Object-Oriented Software, Guided by Tests, 1st edn. Addison-Wesley Professional (2009)
6. ISO/IEC 25010: ISO/IEC 25010 2011, systems and software engineering - systems and software quality requirements and evaluation (square) - system and software quality models (2011)
7. Jander, K., Braubach, L., Pokahr, A.: Defense-in-depth and role authentication for microservice systems. In: Proceedings 9th International Conference on Ambient Systems, Networks and Technologies. Procedia Computer Science (open-access), pp. 456–463. Elsevier Science (2018)
8. Jander, K., Pokahr, A., Braubach, L., Kalinowski, J.: Service discovery in megascale distributed systems. In: Ivanovic, M., Badica, C., Dix, J., Jovanovic, Z., Malgeri, M., Savic, M. (eds.) Intelligent Distributed Computing XI, pp. 273–284. Springer (2017)
9. Nadareishvili, I., Mitra, R., McLarty, M., Amundsen, M.: Microservice Architecture: Aligning Principles, Practices, and Culture, 1st edn. O'Reilly Media, Inc. (2016)
10. Narkhede, N., Shapira, G., Palino, T.: Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale. O'Reilly UK Ltd. (2017)

[4] https://vertx.io/.

11. Newman, S.: Building Microservices - Designing Fine-Grained Systems. O'Reilly Media (2015)
12. Newman, S.: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith. O'Reilly Media, Incorporated (2019)
13. Object Management Group (OMG): The Common Object Request Broker: Architecture and Specification, revision 2.0 edn. (1995)
14. Ragan, T.: Navigating the microservice deathstar with deployhub (2019)
15. Richardson, C.: Introduction to microservices - the drawbacks of microservices (2015)