Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns

Uwe Zdun[1], Elena Navarro[2], Frank Leymann[3]

[1] Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Austria, Email: uwe.zdun@univie.ac.at
[2] Computing Systems Department, Laboratory of User Interaction and Software Engineering, University of Castilla-La Mancha, Spain, Email: elena.navarro@uclm.es
[3] Institute of Architecture of Application Systems, University of Stuttgart, Germany, Email: frank.leymann@iaas.uni-stuttgart.de

Abstract. Microservice-based software architecture design has been widely discussed, and best practices have been published as architecture design patterns. However, conformance to those patterns is hard to ensure and assess automatically, leading to problems such as architectural drift and erosion, especially in the context of continued software evolution or large-scale microservice systems. In addition, not much in the component and connector architecture models is specific (only) to the microservices approach, whereas other aspects really specific to that approach, such as independent deployment of microservices, are usually modeled in other views or not at all. We suggest a set of constraints to check and metrics to assess architecture conformance to microservice patterns. In comparison to expert judgment derived from the patterns, a subset of these constraints and metrics shows a good relative performance and potential for automation.

1 Introduction

Many approaches have been proposed for service-based architecture decomposition (see e.g. [16,19,21,28]). An approach which evolved from established best practices are *microservices*, as Newman [15] points out: "The microservices approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well." Lewis and Fowler [14] describe microservices as "an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery." More detailed discussions can be found in [27,18].

This paper focuses on architecture decomposition based on the microservices approach. Many required decisions about how to perform the major architecture decomposition into microservices have already been described in form of architectural design patterns [21]. However, those and related patterns can lead to architecture designs in many different variants and combinations of pattern-based design options, making it hard to automatically or semi-automatically judge questions such as: When designing a microservice architecture, how much did a project deviate from the established best practices? After evolving a microservice architecture, are we still in conformance with

the chosen microservice patterns? When moving from a monolithic architecture to a microservice architecture, how big is the gap to a microservice-based design?

For checking or assessing such questions related to pattern conformance of the microservice architecture, a high level of automation would be very useful. While it is possible to judge these questions for a small scale architecture manually, in practice it is rarely done in each architecture evolution step, leading to architectural drift and erosion [20]. For larger scale projects, manual assessment is more difficult. For instance, consider the work of an integration architect judging pattern conformance in hundreds of microservices. Here, manual assessment can only work in a cost-effective way, if every team is very disciplined and assesses their own conformance in each evolution step. Further, without automation, at a larger scale with many different stakeholders, judging pattern conformance objectively and uniformly across teams and stakeholders is difficult. These points have led us to address the following research questions:

RQ1: Which measures can be defined to automatically check or assess pattern conformance in microservice decomposition architectures?

RQ2: How well do such measures perform in relation to expert judgment?

RQ3: Given that many defining aspects of microservices (like independent deployment) are modeled outside of a microservice decomposition architectures, what is a set of minimal elements needed in a microservice decomposition architecture to compute meaningful measures?

Our major contributions are the following. Based on existing microservice patterns [21] we have hypothesized a number of constraints and metrics to make an automated judgment on microservice architecture decomposition. To evaluate those constraints and metrics, we have modeled 13 architecture models taken from the practitioner literature and assessed each of them manually regarding its quality and violations of microservice patterns (following as closely as possible the expert judgment of the pattern authors). We have then compared the results in depth and statistically over the whole evaluation model set. Our results are: A subset of the constraints and metrics are quite close to the pattern-based assessment based on the expert judgment taken from the patterns. We identified only a few necessary modeling elements in microservice decomposition architectures, meaning that they are rather easy to create semi-automatically (e.g. using the approach from [6]). Moreover, in those models not much is (only) specific to microservices so that there is still room for improvement. Such further improvement would require detailed modeling of the microservices and thus more manual effort.

This paper is organized as follows. Section 2 compares to related work. Next, we discuss a minimal formal model as a basis for modeling microservice-based architecture decomposition in Section 3. Section 4 introduces our suggested microservice design constraints and metrics, and Section 5 evaluates them in the context of 13 models from practice. Section 6 discusses the RQs regarding the evaluation results and concludes.

2 Related Work

Many studies currently study microservice-based architectures in the context of DevOps or container-technologies like Docker (see e.g. [8,3,9]). In addition, quite a number of studies analyse the application of microservices in various application domains such as

data centers [12], digital archives [10], or Web apps [25], to name but a few. A recent mapping study [1] confirms that the major interests in these and other studies are mostly the concrete system architectures often in relation to deployment, cloud, monitoring, performance, APIs, scalability, and container-technologies. That is, these studies are related to ours, so far, as their architectures are potential targets for our approach. The additional aspects that are studied in those approaches (like performance, scalability, or deployment aspects) are potential extensions of our approach, as possible future work.

First engineering approaches, specific to microservices are emerging. We have based our work on the microservice patterns by Richardson [21]. For instance, the *API Gateway* pattern is beneficial in a *Microservice Architecture*, but not a must. This pattern proposes "a single entry point for all clients." A variant of *API Gateway* is the *Backend for Frontend* pattern that "defines a separate *API Gateway* for each kind of client." With regard to data stores, the recommended pattern is *Database per Service*, i.e., "an architecture which keeps each microservice's persistent data private to that service and accessible only via its API." Loosely coupled interaction is usually the only intended way how microservices should communicate with each other. This is typically achieved using event-driven communication or messaging [7], in both cases with focus on an eventually consistent approach for communication of data-related operations.

Another set of microservice patterns has been published by Gupta [5], general best practices are discussed in [14], and other similar approaches are summarized in another recent mapping study [16]. So far, however, no automated software engineering tools have been proposed for microservice decomposition in the literature. Engineering approaches rather focus on aspects like support for modeling and composition [11] or migration from monolithic architectures [13]. Related general service design methods focus e.g. on QoS-aware service composition [22] or the involved architecture decisions [28]. While much of the work on service metrics is focused on runtime properties like QoS, some specific design metrics for Web services have been proposed, e.g. focusing on loose coupling [19]. To the best of our knowledge, no general conformance approach for architecture decomposition of microservices – or services in general – exists so far.

Software architecture conformance checking is often based on automated extraction techniques, which could be used as a basis for our approach as well (here following [6]), e.g. using architecture reconstruction approaches [4,24]. Such approaches often can check conformance to architecture patterns [4,6] or other kinds of architectural rules [24]. Other static architecture conformance checking techniques are: dependency-structure matrices, source code query languages, and reflexion models [17]. In such approaches often general software engineering metrics like complexity metrics play a role [17]. Our approach follows the same general strategy like those approaches, but in contrast we focus on specific constraints (or more generally, architecture rules) and metrics derived from microservices best practices – not applicable in a general context, but at the same time more powerful in our specific microservice (or service) context.

3 Modeling Microservice-Based Architecture Decomposition

Figure 1 shows a simple sample microservice decomposition model, as they are modeled in practice (see e.g. [21]). It uses UML2 component model notation with one ex-

tension: a *Directed Connector* is modeled using a directed arrow (not part of UML2). Not much in such a model is (only) specific to microservices, but at the same time many aspects may be modeled in a way which is violating some parts of the microservice patterns. This might lead to severe problems in other views of the architecture or system, such as logical, detailed design or deployment views. For instance, a decomposition that would hinder independent deployment, uses many shared dependencies and is mainly based on strongly coupled connectors, so that it would not be following the microservice best practices well.

From an abstract point of view, a microservice-based architecture decomposition is a decomposition into a directed components and connectors graph with a set of component types for each component and a set of connector types for each connector, expressed formally: An architecture decomposition model M is a tuple $(CP, CN, CPT, CNT, cp\_directtype, cn\_directtype, cp\_supertype, cn\_supertype, cp\_type, cn\_type)$ where:

- CP is a finite set of **component nodes**.
- $CN \subseteq CP \times CP$ is an ordered finite set of **connector edges**.
- CPT is a set of **component types**.
- CNT is a set of **connector types**.
- $cp\_directtype : CP \rightarrow \mathbb{P}(CPT)$ is a function that maps each component node cp to its set of **direct component types**,
- $cp\_supertype : CPT \rightarrow \mathbb{P}(CPT)$ is a function called **component type hierarchy**. $cp\_supertype(cpt)$ is the set of direct supertypes of cpt; cpt is called the subtype of those supertypes. The transitive closure[4] $cp\_supertype^*$ defines the inheritance in the hierarchy such that $cp\_supertype^*(cpt)$ contains the **direct and indirect (aka transitive) supertypes** of cpt. The inheritance hierarchy is cycle free, i.e. $\forall cpt \in CPT : cp\_supertype^*(cpt) \cap \{cpt\} = \emptyset$.
- $cp\_type : CP \rightarrow \mathbb{P}(CPT)$ is a function that maps each component to its set of **direct and transitive component types**, i.e., $\forall cp \in CP, dt \in CPT : dt = cp\_directtype(cp) \Rightarrow cp\_type(cp) = dt \cup cp\_supertype^*(dt)$.
- $cn\_directtype : CN \rightarrow \mathbb{P}(CNT)$ is a function that maps each connector cn to its set of **direct connector types**.
- $cn\_supertype : CNT \rightarrow \mathbb{P}(CNT)$ is a function called **connector type hierarchy**. $cn\_supertype(cnt)$ is the set of direct supertypes of cnt; cnt is called the subtype of those supertypes. The transitive closure $cn\_supertype^*$ defines the inheritance in the hierarchy such that $cn\_supertype^*(cnt)$ contains the **direct and indirect (aka transitive) supertypes** of cnt. The inheritance hierarchy is cycle free, i.e. $\forall cnt \in CNT : cn\_supertype^*(cnt) \cap \{cnt\} = \emptyset$.
- $cn\_type : CN \rightarrow \mathbb{P}(CNT)$ is a function that maps each connector to its set of **direct and transitive connector types**, i.e., $\forall cn \in CN, dt \in CNT : dt = cn\_directtype(cn) \Rightarrow cn\_type(cn) = dt \cup cn\_supertype^*(dt)$.

With this definition, we can rephrase **RQ3** to the question: Which elements of CPT and CNT and which type hierarchy dependencies of those are actually needed in order to compute meaningful constraints and metrics?

[4] All transitive closures in this article are assumed to be calculated with a standard algorithm for transitive closures like Warshall's algorithm.

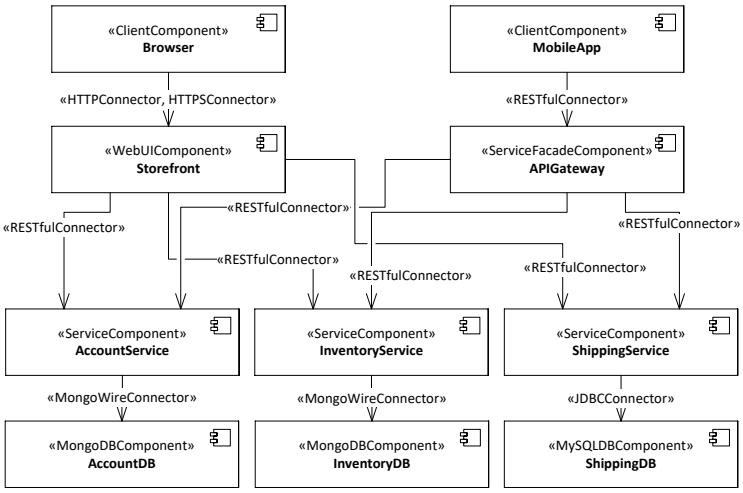

Fig. 1. Sample microservice architecture decomposition model(adapted from [21])

4 Microservice Design Constraints and Metrics

4.1 Constraints and Metrics Based on Independent Deployment

As microservices are emphasized to be independent units of deployment, one hypothesis we have developed was that a good indicator for microservice-based decomposition could be to check whether *all components are independently deployable or to what degree they are independently deployable*. From the viewpoint of an architecture decomposition model, independently deployable means that no components that are part of a microservice have in-memory connectors (or subclasses thereof or similar strongly coupled connectors) to other components that are part of that microservice. In particular, we do not consider external components, as they are not part of a microservice. Finally, microservice should contain components at the same level of abstraction connected only via loosely coupled interfaces. To express this formally, we assume there is a supertype of all in-memory connectors (and similar strongly coupled connectors) *InMemoryConnector* $\in CNT$ and a supertype of all external components *External-Component* $\in CPT$ (with a subtype *ClientComponent*, i.e. $ExternalComponent \in cp\_supertype^*(ClientComponent)$).

- The function $imc : CP \rightarrow \mathbb{P}(CP)$ maps a component to the set of components that are directly connected to the component via connectors typed as $InMemoryConnector$. We call $imc(cp)$ the **direct in-memory cluster** of a component cp with $\forall cp \in CP : imc(cp) = \{co \in CP \,|\, \exists cn \in CN : cn = (cp, co) \wedge InMemoryConnector \in cn\_type(cn)\}$.
- The transitive closure $imc^* : CP \rightarrow \mathbb{P}(CP)$ defines the set of components directly and indirectly connected to a component cp via $InMemoryConnector$ edges. We call $imc^*(cp)$ the **in-memory cluster** of a component cp.

- The function $idcc : CP \rightarrow \mathbb{P}(CP)$ maps a component to its **independently deployable component cluster** such that $\forall cp \in CP : idcc(cp) = \{co \in (\{cp\} \cup imc^*(cp)) \mid ExternalComponent \notin cp\_type(co)\}$.
- The function $idccs : M \rightarrow \mathbb{P}(\mathbb{P}(CP_m))$ maps a model to the set of its **independently deployable component clusters** (i.e., a set of component clusters (CPS) computed with the function $idcc$): $\forall m \in M : idccs(m) = \{CPS \in \mathbb{P}(CP_m) \mid \forall cp \in CP_m : idcc(cp) \in CPS\}$[5].

Based on these definitions we can define the constraint **all components are independently deployable (CAID)**, $CAID : M \rightarrow Boolean$, using the formula below, which computes all independently deployable component clusters CPS in a model m and checks for all CPS that their size is less or equal to 1 using the aggregate function $\mathcal{F}_{count}$. Here, we use the standard aggregate function from relational algebra which counts the number of elements in the collection to compute the size, i.e., it has the same semantics as in SQL. Here, the boolean value 0 means false, i.e. a constraint violation, and 1 means true, i.e. that the constraint is not violated:

$$\forall m \in M : CAID(m) = \begin{cases} 1 \text{ if: } \forall CPS \in idccs(m) : \mathcal{F}_{count}(CPS) \leq 1 \\ 0 \text{ if: } \forall CPS \in idccs(m) : \mathcal{F}_{count}(CPS) > 1 \end{cases}$$

Our implementation of the constraint additionally computes the clusters that have failed to provide precise failure information to the user. Additionally, there is a function for computing the **components violating independent deployability**, $cvid : M \rightarrow \mathbb{P}(CP)$, which simply executes the $CAID$ constraint, and returns an empty set if it is not violated, otherwise all components in the violating clusters. We suggest two metrics that can be derived from this constraint and its underlying functions:

- **Ratio of components violating independent deployability to non-external components (RVID)** is based on the constraint CAID. It used the function $cvid$ to execute the constraint, and returns the number of violating components or an empty set in case of no violation. Then *RVID* sets their number in ratio to the total number of non-external components. $nec : M \rightarrow \mathbb{P}(CP)$ is a helper function returning all components in a model that are not of type *ExternalComponent* (**non-external components**). Here, and in a number of the following metrics counting unique non-external components, we set the component counts in ratio to the model size in terms non-external components, which – compared to the component counts themselves – scales the metric to the interval $[0, 1]$. This, thus, makes metric results for different models more comparable. $RVID : M \rightarrow \mathbb{R}$ is defined as follows:

$$\forall m \in M : RVID(m) = \frac{\mathcal{F}_{count}(cvid(m))}{\mathcal{F}_{count}(nec(m))}$$

[5] We use the notation 'CP_m', 'CN_m' etc. in formulas taking models as input to denote the tuple of elements of the model m; in formulas considering any model, like the previous ones, we omit notation for brevity.

– **Ratio of independently deployable component clusters to non-external components (RIDC)**, $RIDC : M \to \mathbb{R}$, sets the number of independently deployable component clusters in ratio to the size of the model (in terms of non-external components):

$$\forall m \in M : RIDC(m) = \frac{\mathcal{F}_{count}(idccs(m))}{\mathcal{F}_{count}(nec(m))}$$

4.2 Constraints and Metrics Based on Shared Dependencies

Many of the microservice patterns [21] (for a short summary see Section 2) focus on decompositions which *avoid sharing other components or sharing them in a strongly coupled fashion*. Hence, another major idea for constraints and metrics was to base them on the notion of shared components, sharing components, and shared dependencies in the architecture decomposition. With regard to constraints we have envisioned three basic types of constraints: *no shared components* which checks whether there is no shared component; *no sharing components* which checks whether there is no sharing component; *no shared dependencies* which checks whether there is no shared dependency of two components. As typically different clients can share a microservice, and microservices can themselves share third-party microservices, all external components need to be excluded from these constraints (and metrics). All three constraints are based on the same algorithm for finding the set of shared dependencies of each component in the model, requiring the following functions for this:

– $acd : CP \to \mathbb{P}(CP)$ is a function which calculates **all direct component dependencies** of a component. That is, $acd(cp)$ is defined formally as: $\forall cp \in CP : acd(cp) = \{cd \in CP \mid \exists cn \in CN : cn = (cp, cd)\}$. The transitive closure acd^* defines **all direct and indirect component dependencies** of a component cp.
– $ascd : M \to \mathbb{P}(CP_m \times (CP_m \times CP_m))$ is a function which maps a model $m \in M$ to a set of tuples containing a component $cp \in CP_m$ and the set of **all shared component dependencies** of that component cp (excluding external components). Each of these shared component dependencies is itself a tuple (oc, sd) being $oc \in CP_m$ the other component with which cp shares a dependency and $sd \in CP_m$ the component which is shared both by oc and cp, expressed formally: $\forall m \in M : ascd(m) = \{(cp, (oc, sd)) \mid cp, oc, sd \in CP_m \wedge sd \in acd(cp) \wedge sd \in acd(oc) \wedge ExternalComponent \notin cp\_type(cp) \wedge ExternalComponent \notin cp\_type(oc) \wedge ExternalComponent \notin cp\_type(sd)\}$.
– $sic : M \to \mathbb{P}(CP_m)$ is a function that provides the set of all **sharing non-external components**, formally defined as: $\forall m \in M : sic(m) = \{cp \in CP_m \mid \exists oc, sdCP_m : (cp, (oc, sd)) \in ascd(m)\}$.
– $sdc : M \to \mathbb{P}(CP_m)$ is a function that provides the set of all **shared non-external components**, formally defined as: $\forall m \in M : sdc(m) = \{sd \in CP_m \mid \exists oc, cpCP_m : (cp, (oc, sd)) \in ascd(m)\}$.

The closer study of the three types of constraints revealed that they lead to exactly the same violations: as a shared dependency leads to a sharing and a shared component, either all these constraints are violated or none of them. For this reason, it is enough for us to formally define and study one of those constraints. Here,

we define the constraint **no shared non-external component dependencies (NSCD)**, $NSCD : M \rightarrow Boolean$, as (0 = false, i.e. a constraint violation, and 1 = true, i.e. no constraint violation):

$$\forall m \in M : NSCD(m) = \begin{cases} 1 \text{ if: } \forall\, SD \in ascd(m) : \mathcal{F}_{count}(SD) = 0 \\ 0 \text{ if: } \forall\, SD \in ascd(m) : \mathcal{F}_{count}(SD) > 0 \end{cases}$$

Further for this constraint (and all related metrics) below, we suggest – in addition to the basic constraint – three variants.

- *NSCD-F* excludes *Facade* components from the constraint. Many microservice models (as well as monolithic models) contain *Facades*, such as an *APIGateway* in Figure 1, as an acceptable way to share microservice components [2]. We thus assume a class *Facade* $\in CPT$ with classes like *APIGateway* as its subclasses (thus also $\in CPT$ through e.g. $Facade \in cp\_supertype^*(APIGateway)$ and so on). At first we envisioned to automatically compute which components are *Facades*, but unfortunately this design intent is impossible to compute in an unambiguous way. For instance, our evaluation model *RB* (see Table 1) contains microservices that are directly connected to clients, and, without further information, there is no way to automatically distinguish those from a model in which only *Facades* are modeled. For this reason, all **-F* variants of constraints and metrics require *Facades* to be explicitly modeled. The rationale behind the **-F* variants is: If *Facades* are modeled, we hypothesize that excluding them from the constraints and metrics could lead to a better identification of real issues with regard to shared dependencies. For space reasons, we omit the formal definition here, as it is analogous to the functions/constraints defined above, just excluding *Facades* in the functions.
- *NSCD-C* excludes loosely coupled connectors (event-driven, publish/subscribe style interaction, and message queuing) from further investigation. We assume a class *LooselyCoupledConnector* $\in CNT$ with subclasses such as *EventBased-Connector*, *PubSubConnector*, *MessagingConnector* (all also $\in CNT$, using $cn\_supertype^*$ relations). That is, only strongly coupled connectors can lead in **-C* variants of constraints and metrics to constraint violations or lower metrics values. As the patterns suggest to use only loosely coupled interaction in event-driven, publish/subscribe style between microservices, we hypothesize that excluding them from the constraints and metrics could lead to a better identification of a real issue with regard to shared dependencies. We expect that the exclusion of loosely coupled connectors makes the results more comparable for different models in the sense that in this way the same model, modeled at different levels of detail, leads to the same metric values and constraint violations. For space reasons, we omit the formal definition here, as it is analogous to the functions/constraints defined above, just excluding *LooselyCoupledConnectors* in the functions.
- *NSCD-FC* is the combination of *NSCD-F* and *NSCD-C*.

All metrics below are defined analogously in a basic version plus three variants. Here, however, the differences between shared components, sharing components, and shared dependencies play a major role, and it is interesting to study which of those basic

counts is better suited as a foundation for a shared dependency metric. Firstly, we define the **ratio of sharing non-external components to non-external components (RSIC)**, $RSIC : M \rightarrow \mathbb{R}$, based on the count of components returned by the functions sic (defined above) set in relation to the non-external components count (based on nec) as:

$$\forall m \in M : RSIC(m) = \frac{\mathcal{F}_{count}(sic(m))}{\mathcal{F}_{count}(nec(m))}$$

Secondly, we define the **ratio of shared non-external components to non-external components (RSCC)**, $RSCC : M \rightarrow \mathbb{R}$, based on functions sdc and nec:

$$\forall m \in M : RSCC(m) = \frac{\mathcal{F}_{count}(sdc(m))}{\mathcal{F}_{count}(nec(m))}$$

Finally, we suggest a metric **ratio of shared dependencies of non-external components to possible dependencies (RSDP)**, $RSDP : M \rightarrow \mathbb{R}$ based directly on the number of shared dependencies returned by the function $ascd$. Here we scale the metric using the number of all possible dependencies (i.e., the number of counted components squared). As this value has no specific meaning in the context of our model, we have also compared other scalings in our evaluation like no scaling, the model size in terms components, and all component dependencies. We have chosen only the scaling based on all possible dependencies here, as all other metrics perform weaker in our evaluation, and at the same time none of the other options scales the metric to the normed interval [0,1]. As a result, we suggest the metric:

$$\forall m \in M : RSDP(m) = \frac{\mathcal{F}_{count}(ascd(m))}{(\mathcal{F}_{count}(nec(m)))^2}$$

All metrics, defined in this section, also have *-F, *-C, and *-FC variants, with analogous reasoning to the discussion for *NSCD*. The differences in formal definition to the base variants are the following: The metrics must use adapted versions of the functions, analogously to the *NSCD* variants, and the function nec in the divisor of the metrics should be adapted to not consider *Facades* for the two *-F and *-FC variants, as scaling should be done according to the considered components.

5 Evaluation

For performing our evaluation, we have fully implemented our formal model, constraints, metrics, and related algorithms using the Frag Modeling Framework (FMF), a runtime modeling, domain-specific language and generator framework implemented on top of Java/Eclipse which enables us to easier change design decisions made and perform experimentation than in comparable frameworks like the Eclipse Modeling Framework (EMF) (see [26] for more details). Besides extensive test cases, a code generator to generate R scripts has been implemented, used to perform statistical comparison of achieved and expected results for the different constraints and metrics. In addition, we have fully modeled and implemented 13 models in an evaluation model set, summarized in Table 1. Each of the models is either taken directly from a model published

by practitioners or adapted according to discussions on the respective referenced Web sites. While the models taken from 4 independent sources[6] are still examples, they all originate from models developed by practitioners with microservice and monolith implementation experience. Hence, we assume that our evaluation models are close to models used in practice and real practical needs for microservice decomposition (compared e.g. to models created solely by ourselves).

Table 1. Summary of models used for evaluation and manual assessment of pattern compliance

ID	Size	Short Description	Major Violations of Patterns	VMP	MQ
EC1	10 comp., 11 conn.	E-Commerce model with 3 independent microservices, an API gateway, a Web UI, databases per service, inter-service communication not modeled	None	0	1.0
EC2	13 comp., 19 conn.	Similar to EC1; additionally 1 service consists of 4 components which are realizing different business capabilities	A service contains different subdomains/capabilities or is not modeled at the same abstraction level	1	0.6
EC3	11 comp., 17 conn.	Similar to EC1; additionally models inter-service communication using the Event Sourcing pattern	None	0	1.0
EC4	11 comp., 17 conn.	Similar to EC1; additionally models inter-service communication using the Transaction Log Trailing (or Database Trigger) pattern	None	0	1.0
EC5	8 comp., 11 conn.	Similar to EC1; with only one database, which is shared among the microservices	Shared database	1	0.6
EC6	8 comp., 11 conn.	Same components as in EC1 but all in one shared address space, shared database, API gateway, Web UI	No decomposition into multiple services (all other violations are secondary)	1	0.0
EC7	8 comp., 14 conn.	Similar to EC6; with all in-memory component dependencies explicitly modeled	No decomposition into multiple services (all other violations are secondary)	1	0.0
EC8	11 comp., 19 conn.	Similar to EC2; with only one database, which is shared among the microservices	A service contains different subdomains/capabilities or is not modeled at the same abstraction level; shared database	1	0.4
RB	4 comp., 3 conn.	Single service for restaurant booking, no clients modeled, follows CQRS pattern, uses REDIS for fast denormalized querying	None	0	1.0
TH1	18 comp., 17 conn.	Taxi hailing application: 3 microservices with a layer of 3 backend services in addition to 3 databases per service, shared payment component	Shared, strongly coupled component	1	0.6
TH2	18 comp., 17 conn.	Same as TH1, avoids shared component using loosely coupled connectors	None	0	1.0
TH3	15 comp., 19 conn.	Same components as in TH1 but all in one shared address space, 1 shared database, 1 API gateway, 1 Web UI	No decomposition into multiple services (all other violations are secondary)	1	0.0
SA	15 comp., 19 conn.	Web shop app with 7 services, 5 different data stores, 2 modular Web UIs	None	0	1.0

The table also shows our manual, pattern-based assessment of the architecture conformance of each of the models. There are two assessments: *Does the model violate at least one of the microservice patterns (from [21])?* We carefully assessed each model

[6] We have adapted Models EC1-8 from [21]. Model RB is adapted from: `http://eventuate.io/exampleapps.html`. The Models TH1-TH3 are adapted from: `https://www.nginx.com/blog/introduction-to-microservices/`. Model SA is adapted from: `https://www.slideshare.net/smancke/fros-con2014-microservicesarchitecture`. For all models, we aimed to stay close to the original model; adaptation mainly means modeling them using our approach to architecture decomposition modeling and in the model variants introducing the described variations.

for major violations of the patterns. If at least one occurs, we marked it in column Violations of Microservice Patterns (*VMP*) of Table 1 as $true = 1$, otherwise as $false = 0$. In addition, we tried to objectively measure the quality of the model with regard to conformance to the microservice patterns [21]. For this, we use the following rules to compute the Microservice Architecture Quality (Column *MQ* in Table 1) based on a detailed manual inspection of the compliance of the models to the architecture patterns:

- If the *Microservice Architecture* pattern cannot be found at all, that is, the architecture clearly follows a *Monolithic Architecture*, we set *MQ=0*.
- Otherwise we set *MQ=1*, and then if one of the violations listed below (each one can occur multiple times) is found, we reduce *MQ* by 0.4 on the first occurrence, by another 0.2 on the second occurrence (of the same or another pattern), another 0.1 on the third occurrence, and so on. Thus, the violation penalty is divided by factor 2 from one violation occurrence to the next because if such a minor violation occurs, the model should not be better rated than 0.6. But even if multiple minor violations happen, the rating should still stay better than the monolithic score of 0. The violations analyzed are the following: (1) A minor violation of the *Microservice Architecture* pattern occurs, such as some microservices contain components corresponding to multiple different capabilites or subdomains, or not all microservices are modeled at the same abstraction level. (2) Internal components share other internal components not using loosely coupled connectors, e.g. realized using *Event-driven Architecture* (or the realization of an *Event-driven Architecture* violates established patterns for event-based communication among microservices such as *Event Sourcing*, *Transaction Log Tailing*, *Database Triggers*, *Application Publishes Events*, *Command Query Responsibility Segregation*, see [21]). (3) The *Database per Service* pattern is not used, but a *Shared Database*.
- The use of the two API Gateway patterns is beneficial, but does not change the quality assessment. The reason is that API Gateways are also commonly used in monolithic architectures, and a microservice architecture that does not use them is not less well decomposed w.r.t. the microservice patterns. Note that although the API Gateway patterns are still important for our approach, their use is important for calculating some of our constraints and metrics (see discussion on *Facades* below).

We have chosen this scoring scheme because it is close to the suggestions in the patterns and introduces no major subjective bias. In the course of our evaluations, we have compared it to other reasonable scorings, including subjective expert judgment by the authors, and a number of similar mechanical scorings. The sensitivity to those scorings was generally low, as long as we followed the suggestions from the patterns closely. The evaluation of the constraints leads to binary vectors indicating for each model whether the constraint is violated or not. Below we discuss the results of each of these vectors in detail. In addition, we calculated the Jaccard similarity [23] to the vector built from *VMP* values in Table 1 (*JS_VMP* in Table 2) to get a quick estimate of how well the respective constraint performs in relation to the manual, pattern-based assessment for our evaluation model set. The Jaccard similarity is a common index for binary samples, which is defined as the quotient between the intersection and the union of the pairwise compared variables among two vectors.

Metrics evaluation leads to vectors with positive values which should indicate the quality of the microservice decomposition. Again, we discuss them in detail below. In addition, we compute the Cosine similarity with the vector *MQ* from Table 1 (*CS_MQ* in Table 3) to get a quick estimate of how well the respective metric performs in relation to the pattern-based assessment for our evaluation model set. Cosine similarity is a common measure of similarity between two vectors based on the cosine of the angle between them [23]. Some of the metrics below are reversed compared to *MQ* in the sense that their best value is 0.0, with higher values indicating better quality. Consequently, we compared those metrics to the reversed *MQ*, which is defined as *MQR* = *1 – MQ* (below indicated as *CS_MQR*). Alternatively, we could calculate the associated distance metrics, where the distance d is also defined in relation to its associated similarity metric as $d = 1 - s$.

Table 2. Evaluation Results: Constraints (1 - constraint is violated, and 0 - it is not violated)

Constraint	EC1	EC2	EC3	EC4	EC5	EC6	EC7	EC8	RB	TH1	TH2	TH3	RSA	*JS_VMP*
CAID	0	1	0	0	0	1	1	1	0	0	0	1	0	*0.71*
NSCD	1	1	1	1	1	1	1	1	1	1	1	1	1	*0.54*
NSCD-F	0	1	1	1	1	1	1	1	1	1	1	1	1	*0.58*
NSCD-C	1	1	1	1	1	1	1	1	0	1	1	1	0	*0.64*
NSCD-FC	0	1	0	0	1	1	1	1	0	1	0	1	0	*1.0*

Table 3. Evaluation Results: Metrics

Metric	EC1	EC2	EC3	EC4	EC5	EC6	EC7	EC8	RB	TH1	TH2	TH3	SA	*CS_MQ*	*CS_MQR*
RVID	0.0	0.36	0.0	0.0	0.0	0.83	0.83	0.44	0.0	0.0	0.0	0.89	0.0		*0.96*
RIDC	1.0	0.73	1.0	1.0	1.0	0.33	0.33	0.67	1.0	1.0	1.0	0.22	1.0	*0.97*	
RSIC	0.63	0.73	0.56	0.89	0.83	0.83	0.83	0.89	0.5	0.33	0.33	0.56	0.64		*0.73*
RSIC-F	0.0	0.44	0.43	0.86	0.75	0.75	0.75	0.86	0.5	0.22	0.22	0.43	0.5		*0.74*
RSIC-C	0.63	0.73	0.56	0.56	0.83	0.83	0.83	0.89	0.0	0.33	0.33	0.56	0.0		*0.81*
RSIC-FC	0.0	0.44	0.0	0.0	0.75	0.75	0.75	0.86	0.0	0.22	0.0	0.43	0.0		*0.91*
RSCC	0.75	0.82	0.78	0.78	0.67	0.67	0.67	0.78	0.25	0.5	0.5	0.78	0.64		*0.70*
RSCC-F	0.0	0.44	0.14	0.14	0.25	0.25	1.0	0.57	0.25	0.11	0.11	0.29	0.33		*0.80*
RSCC-C	0.75	0.82	0.67	0.67	0.67	0.67	0.67	0.78	0.0	0.5	0.42	0.78	0.0		*0.76*
RSCC-FC	0.0	0.44	0.0	0.0	0.25	0.25	1.0	0.57	0.0	0.11	0.0	0.29	0.0		*0.85*
RSDP	0.38	0.89	0.54	0.99	0.72	0.72	1.56	1.51	0.13	0.11	0.11	0.62	0.35		*0.79*
RSDP-F	0.0	0.37	0.12	0.61	0.38	0.38	0.75	0.98	0.13	0.02	0.02	0.16	0.08		*0.72*
RSDP-C	0.38	0.89	0.3	0.3	0.72	0.72	1.56	1.51	0.0	0.11	0.07	0.62	0.0		*0.85*
RSDP-FC	0.0	0.37	0.0	0.0	0.38	0.38	0.75	0.98	0.0	0.02	0.0	0.16	0.0		*0.79*

5.1 Evaluation for Constraints and Metrics Based on Independent Deployment

Table 2 shows the results for the constraint **all components are independently deployable (CAID)**[7]. We can see an acceptable Jaccard similarity (0.71) of the constraint violation vector to the pattern-based assessment *VMP*. Inspecting the violations closer,

[7] In Table 2, 1 means that the constraint is violated, and 0 that it is not violated.

we can see that two violations are not found (false negatives): the violations in Models *EC5* and *TH1*. That is, the constraint does not work well for non-monolithic structures that share a database as in *EC5* or a component as in *TH1*. The constraint works, however, if this issue is combined with other violations as in Model *EC8*.

We have suggested two metrics based on independent deployment: **Ratio of components violating independent deployability to non-external components (RVID)** and **ratio of independently deployable component clusters to non-external components (RIDC)**. RVID sets the unique components in the violations in ratio; that is, 0 indicates the highest possible quality, and higher values indicate lesser quality. Thus, the metric must be compared to the reversed microservice quality vector *MQR*. The cosine similarity *CS_MQR* shows a very high similarity of 0.96. *RIDC*, in contrast, has values ranging from 0 to 1, with 1 indicating the best possible quality, meaning it must be compared to the microservice quality vector *MQ*. Here, we see an even slightly higher cosine similarity *CS_MQ* of 0.97. As both metrics are based on the functions used in *CAID*, they also have the same weakness of not identifying the shared database/component issues in Models *EC5/TH1*, but the high similarity measures show that the indication of quality with regard to the other microservice patterns is rather good for both metrics, with *RIDC* performing slightly better for our evaluation model set.

5.2 Evaluation for Constraints and Metrics Based on Shared Dependencies

No shared non-external component dependencies (NSCD) is violated by all models (6 false positives) and has only a Jaccard similarity of 0.54; it is not a good match. Its variant *NSCD-F*, which excludes sharing by *Facade* components, is slightly better suited, but still has 5 false positives and a Jaccard similarity of only 0.58; the variant *NSCD-C*, which considers only strongly coupled connectors as leading to shared components, is slightly better with 4 false positives and a Jaccard similarity of 0.64. The combination *NSCD-FC* considering no *Facades* and no loosely coupled connectors produces exactly the same vector as the pattern-based assessment *VMP* (and thus the Jaccard similarity is 1.0). This very good result might be surprising, as the uncombined constraints *NSCD-F* and *NSCD-C* produce rather weak results alone. A closer inspection revealed that in our models there was indeed in each false positive in *NSCD-F* a loosely coupled connector and *NSCD-C* a sharing *Facade* that caused the violation.

For all shared dependencies metrics, the value 0.0 is the best possible value, and higher values indicate lower quality. Thus, the metrics must be compared to the reversed microservice quality vector *MQR*. The **ratio of sharing non-external components to non-external components (RSIC)** shows a moderate cosine similarity *CS_MQR* of 0.73, which is gradually improved by its two variants *RSIC-F* and *RSIC-C* with cosine similarities 0.74 and 0.81, respectively. The combined variant *RSIC-FC* shows the best results with a high cosine similarities of 0.91.

For **ratio of shared non-external components to non-external components (RSCC)** the cosine similarity *CS_MQR* has a moderate value of 0.70. Its variants *RSCC-F* and *RSCC-C* perform better with cosine similarities of 0.80 and 0.76, respectively. Again, the combined variant *RSCC-FC* shows the best results with a high cosine similarities of 0.85, but it is less similar for our evaluation model set than *RSIC-FC*.

Finally, **ratio of shared dependencies of non-external components to possible dependencies (RSDP)** has a good cosine similarity of 0.79 already in its basic variant, but interestingly *RSDP-F* performs weaker with a cosine similarity of only 0.72. Close inspection of the dependencies revealed that this effect is due to the fact that, on the one hand, the *Facade* dependencies make the values for high quality microservice architectures worse, but, on the other hand, they make them much more worse for monolithic architecture, as for them *Facades* have many more dependencies. Thus, monolithic architectures gain in the variant *RSDP-F* comparatively too much. This can, in our numbers for instance, be easily retraced using the values for Models EC1 and EC6. While *RSDP-F* leads to a comparatively better result for EC1 (0.0 instead of 0.38 for *RSDP*), the monolith EC6 improves from 0.72 (which was close to the expected reversed quality of 1.0) to 0.38 (which is much more distant from 1.0). *RSDP-C* leads to the expected improvement with a cosine similarity of 0.85. *RSDP-FC* suffers from the same effect for *Facade* dependencies, and thus has only a moderate cosine similarity of 0.79.

6 Discussion, Threats to Validity and Future Work

Discussion of RQs. With regard to **RQ1** and **RQ2**, we have suggested a number of constraints for checking the quality of microservice decomposition in software architecture models. The variant *NSCD-FC* of the shared dependency based constraints performs best, correctly identifying all constraint violations. The constraint *CAID* based on independent deployment performs worse than *NSCD-FC* (but better than all other *NSCD* variants), as it has issues with correctly identifying violations related to shared databases or components. Nonetheless, both constraints are useful and should be combined in their use. As both identify different lists of violations, inspecting the results of both constraints can help developers to more easily find the root cause of a violation. In addition, our evaluation revealed that CAID has only false negatives; that is, in our evaluation model set, all violations identified are actually violations. Hence, it can be used in addition to *NSCD-FC* with no danger of suggesting non-issues to be fixed. This is not the case for any of the other *NSCD* variants, which yield false positives.

We have also suggested a number of metrics for measuring the quality of microservice decomposition in software architecture models. For both of the metrics based on independent deployment, *RDIC* and *RVID*, we can assess a very high similarity to our pattern-based assessments, and hence they seem to be both good candidates for measuring the quality of microservice decomposition. *RDIC* performs slightly better than *RVID*, but given that the values and interpretations used in the pattern-based quality assessment contain a certain level of subjectivity, our empirical evaluation does not really identify a clear favorite. As they are based on *CAID*, we should be aware that the base function suffers from some false negatives which are part of the metrics' values. Further research would be needed to improve the metrics in this regard.

For the metrics related to shared dependencies, we can assess that none of the metrics is a perfect match for our pattern-based quality assessment, but considering that the values and interpretations used in the pattern-based quality assessment contain a certain level of subjectivity, the achieved similarities of the two metric *RSIC-FC* and

RSCC-FC, with values of 0.91 and 0.85 are actually quite good matches, with *RSIC-FC* performing a bit better for our evaluation model set. It is interesting that all three **-FC* metrics yield the correct value of 0.0 for well-designed microservice models, and never assign the perfect value for a model with a violation. Unfortunately, the strength of the effect of violations on metrics values is not optimal yet in any of the metrics. For instance, in the best matching metric *RSIC-FC*, EC8 is the worst model; however, in our pattern-based assessment we see its violations as less severe than those e.g. in EC6. *RSSC-FC* is more correct in this regard, but assigns a very strong effect to the violation in EC7 (which is actually the same model as EC6, but just models the violation in more detail). It is unfortunate that the metric *RSDP* suffers from the issues related to the strong effect on removing *Facade* dependencies, but its variant *RSDP-C* performs for our evaluation model set just as well as *RSCC-FC*. Therefore, an interesting direction of further research could be to investigate other ways to mitigate the effects of the shared dependencies of the *Facades* instead of excluding them.

Overall, based on our empirical results using one of the metrics *RDIC* or *RVID* seems advisable. The results show that the shared dependency metrics in their current form are inferior. However, our results also indicate that shared dependency constraints and metrics can be improved by modeling more details. Here, we have studied *Facades* and loosely coupled connectors, as they are important structures in the microservice patterns and rather easy to model. Please note that modeling additional details is less needed for constraints and metrics based on independent deployment.

In the context of **RQ3**, we can assess that our decomposition model needs rather minimal extensions (the few component and connector types named above) and is easy to map to existing modeling practices. In particular, in order to fully model our evaluation model set, we needed to introduce 20 component types and 42 connector types, ranging from general notions like *ExternalComponent* and its sub-class *ClientComponent*, to very technology-specific classes like *MongoWireConnector* (a subclass of *DatabaseConnector* connecting to a *MongoDBComponent*, a subclass of *DatabaseComponent*). These would not always be easy to map automatically, but our study has shown that for the suggested constraints and metrics, only a small subset is needed: The constraints on independent deployment require at least that *ExternalComponents* (and its subclass *ClientComponent*) and the connector type *InMemoryConnectors* are modeled. The shared dependencies based constraints require two additional abstractions to be modeled: loosely coupled connectors (as subclasses of *LooselyCoupledConnector*) and *Facade* components. All except *Facade* components are relatively easy to compute automatically, e.g. by inspecting the used technology for a connection. We can assess: our approach can easily be mapped using an automated mapping from the source code to an architecture model that assumes standard component model abstractions, such as those in UML2, e.g. with approaches like our architecture abstraction approach [6].

Future Work. In our approach, we have focused only on modeling additionally details with no to low effort, to enable a high potential for automation and less extra effort compared to existing modeling practices. An interesting direction for future research could be to study how modeling more details could lead to better results in the metrics. For instance, modeling capabilities or subdomains of the microservices, or the detailed domain model, are promising directions to further improve the metrics.

Major Threats to Validity. A threat to validity is that potentially the patterns or our models are not well chosen as study objects and do not represent the domain of microservices well. However, as related practices and similar models have been proposed by many other authors, we judge this threat to be rather low. However, many authors also model other architectural views, and they might have an influence on architecture decomposition – which we want to study as future work. Potentially the authors could have been biased in their judgment, but as we have followed a quite mechanical scoring scheme (based on the patterns, not our own judgment), this threat is mostly limited to our evaluations based on the pattern-based quality assessments (see Section 5). Even though we have aimed to follow the argumentations in the microservice patterns [21] as closely as possible, a major threat remains that at least the evaluation scores introduced are subjective to a certain degree. Note that we have tested in the course of our evaluations some other kinds of reasonable scoring scheme, leading to comparable but slightly different results. The sensitivity to those scores was generally low, as long as we followed the suggestions from the patterns closely. In addition, this potential threat to validity is not necessarily a problem, in the sense that a project aiming to apply the constraints and metrics could easily re-run our evaluations with different values that introduce scores according to the project's needs. As we have used pretty basic and standard statistics, we see no major threats to statistical conclusion validity.
Concluding Remarks. In summary, our results show that a subset of the constraints or metrics are quite close to the pattern-based assessment based on the expert judgment taken from the patterns, and we have also shown where the metrics and constraints could be substantially improved. Our results indicate that the best way to reach this goal seems to be more detailed modeling of the microservices (e.g. based on capabilities, subdomains, domain-specific models, and/or modeling at different abstraction levels). However, each of these possible future works would also mean more manual effort, and less potential for automation, but this might not be an issue in all those application cases where designing a well-defined architecture is the goal. With modest effort our results are applicable to other service decomposition schemes than microservices as well.

Acknowledgment This work was partially supported by Austrian Science Fund (FWF) project ADDCompliance: I 2885-N33; DFG ADDCompliance project: LE 2275/13-1; Spanish Ministry of Economy, Industry and Competitiveness, State Research Agency / European Regional Development Fund, grant Vi-SMARt (TIN2016-79100-R).

References

1. Alshuqayran, N., Ali, N., Evans, R.: A systematic mapping study in microservice architecture. In: IEEE 9th Int. Conf. on Service-Oriented Computing and Applications (SOCA). pp. 44–51. IEEE (2016)
2. De, B.: API Patterns. In: API Management, pp. 81–104. Springer (2017)
3. Guo, D., Wang, W., Zeng, G., Wei, Z.: Microservices architecture based cloudware deployment platform for service computing. In: 2016 IEEE Symposium on Service-Oriented System Engineering (SOSE). pp. 358–363. IEEE (2016)
4. Guo, G.Y., Atlee, J.M., Kazman, R.: A software architecture reconstruction method. In: Software Architecture, pp. 15–33. Springer (1999)

5. Gupta, A.: Microservice design patterns. http://blog.arungupta.me/microservice-design-patterns/ (2017)
6. Haitzer, T., Zdun, U.: Semi-automated architectural abstraction specifications for supporting software evolution. Science of Computer Programming 90, 135–160 (2014)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley (2003)
8. Kang, H., Le, M., Tao, S.: Container and microservice driven design for cloud infrastructure devops. In: 2016 IEEE Int. Conf. on Cloud Engineering (IC2E). pp. 202–211. IEEE (2016)
9. Kratzke, N.: About microservices, containers and their underestimated impact on network performance. In: Proceedings of Cloud Computing. pp. 165–169 (2015)
10. Kurhinen, H., Lampi, M.: Micro-services based distributable workflow for digital archives. In: Archiving Conference. pp. 47–51. No. 1, Society for Imaging Science and Tech. (2014)
11. de Lange, P., Nicolaescu, P., Derntl, M., Jarke, M., Klamma, R.: Community application editor: Collaborative near real-time modeling and composition of microservice-based web applications. In: Modellierung (Workshops). pp. 123–128 (2016)
12. Le, V.D., Neff, M.M., Stewart, R.V., Kelley, R., Fritzinger, E., Dascalu, S.M., Harris, F.C.: Microservice-based architecture for the nrdc. In: 2015 IEEE 13th Int. Conf. on Industrial Informatics (INDIN). pp. 1659–1664. IEEE (2015)
13. Levcovitz, A., Terra, R., Valente, M.T.: Towards a technique for extracting microservices from monolithic enterprise systems. arXiv preprint arXiv:1605.03175 (2016)
14. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term. http://martinfowler.com/articles/microservices.html (Mar 2004)
15. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly (2015)
16. Pahl, C., Jamshidi, P.: Microservices: A systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science. pp. 137–146 (2016)
17. Passos, L., Terra, R., Valente, M.T., Diniz, R., das Chagas Mendonca, N.: Static architecture-conformance checking: An illustrative overview. IEEE software 27(5), 82–89 (2010)
18. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 1: Reality check and service design. IEEE Software 34(1), 91–98 (Jan 2017)
19. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: a multi-faceted metric for service design. In: 18th Int. Conf. on World wide web. pp. 911–920. ACM (2009)
20. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
21. Richardson, C.: A pattern language for microservices. http://microservices.io/patterns/index.html (2017)
22. Rosenberg, F., Celikovic, P., Michlmayr, A., Leitner, P., Dustdar, S.: An end-to-end approach for qos-aware service composition. In: IEEE Int. Conf. on Enterprise Distributed Object Computing Conference (EDOC'09). pp. 151–160. IEEE (2009)
23. Tan, P.N., Steinbach, M., Kumar, V.: Introduction to Data Mining. Addison-Wesley (2005)
24. Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Symphony: View-driven software architecture reconstruction. In: 4th Working IEEE/IFIP Conf. on Software Architecturen(WICSA 2004). pp. 122–132. IEEE (2004)
25. Viennot, N., Lécuyer, M., Bell, J., Geambasu, R., Nieh, J.: Synapse: a microservices architecture for heterogeneous-database web applications. In: 10th European Conf. on Computer Systems. p. 21. ACM (2015)
26. Zdun, U.: A DSL toolkit for deferring architectural decisions in DSL-based software design. Information & Software Technology 52(7), 733–748 (2010)
27. Zimmermann, O.: Microservices tenets. Computer Science - Research and Development 32(3), 301–310 (Jul 2017)
28. Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., Schuster, N.: Reusable architectural decision models for enterprise application development. In: Int. Conf. on the Quality of Software Architectures. pp. 15–32. Springer (2007)