Metrics for Assessing Architecture Conformance to Microservice Architecture Patterns and Practices

Evangelos Ntentos[1(✉)], Uwe Zdun[1], Konstantinos Plakidas[1], Sebastian Meixner[2], and Sebastian Geiger[2]

[1] Faculty of Computer Science, Research Group Software Architecture, University of Vienna, Vienna, Austria
{evangelos.ntentos,uwe.zdun,konstantinos.plakidas}@univie.ac.at
[2] Siemens Corporate Technology, Vienna, Austria
{sebastian.meixner,sebastian.geiger}@siemens.com

Abstract. Many contemporary service-based systems follow the microservice approach, particularly in DevOps or continuous delivery contexts. They share a set of important tenets such as independent development and deployment, high releasability, polyglot technology support, and loose coupling. A number of best practices for microservice architectures have been codified as patterns, which embody those tenets. However, no real-world microservices system can support all patterns and practices well, but rather architectural decisions making trade-offs among them are needed. Conformance to the patterns and practices selected in such decisions is hard to ensure and assess automatically, especially in large-scale, complex, and evolving systems. In this work, we propose a model-based approach based on generic, technology-independent metrics, tied to typical architectural design decisions in the microservice domain. With this approach we can measure conformance to the patterns and related tenets. We demonstrate and assess the validity and appropriateness of these metrics in performing an assessment of a system's conformance to patterns through statistical methods.

1 Introduction

Microservices architectures [10,19] structure an application as a collection of autonomous services, modeled around a domain. They share a set of important *tenets* such as development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, high releasability, end-to-end tracing and monitoring, and continuous delivery [9,10,19]. This work examines ways to ensure architecture conformance to these microservice tenets while applying established patterns and practices. That is, many architectural patterns that reflect recommended "best practices" in a microservices context have already been published in the literature [14,15,20]. Conformance to these

E. Kafeza et al. (Eds.): ICSOC 2020, LNCS 12571, pp. 580–596, 2020.
https://doi.org/10.1007/978-3-030-65310-1_42

patterns impacts how far a microservice system supports the desired microservices tenets.

Unfortunately, as real-world, industrial microservice-based systems are usually highly complex, often highly polyglot, and rapidly changed and released (see, e.g. [2,8]), an automatic or semi-automatic assessment of their pattern conformance is difficult: real-world systems feature various combinations of these patterns and different degrees of violations of the same. Different technologies in various parts of the system implement the patterns in different ways, and these implementations are continuously changing at a high pace. Making matters even more challenging, a high level of automation is required for complex systems. While for small-scale systems of a few services, a manual assessment by an expert is probably as quick and as accurate as an automated one, that is not true for industrial-scale systems of several hundred or more services, which are being developed by different teams or companies, evolving at different paces. In that case, manual assessment is laborious and inaccurate, and a more automated method would vastly improve cost-effectiveness. Another major challenge is that no microservice system can support all microservice tenets well at once. Rather the *architectural decisions* for or against a set of related patterns and practices need to make a trade-off among the desired tenets and important other quality attributes [6,19]. Under these considerations, this paper aims to study the following research questions:

- **RQ1.** How can conformance to the tenets embodied in microservice architecture decision options (i.e. patterns and practices) be automatically assessed?
- **RQ2.** How well do measures for assessing decision options and their associated tenets perform?
- **RQ3.** What is a set of minimal elements needed in a microservice architecture model to compute such measures?

Our approach to address these challenges is to define a set of metrics for each microservice decision associated to the decision's options, i.e. at least one metric per major decision option. Based on a manual assessment of a small set of models and model variants that is representative for the possible decision options and option combinations of the studied decisions, we derive a ground truth. The ground truth is established by objectively assessing whether each decision option is supported. By combining the outcome of all options of a decision, we can then derive an ordinal assessment of how well the decision is supported in each model. We then use the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis. In this paper, we propose an architectural component model based approach which uses only modeling elements that can be derived from the system's source code. For this reason, it is important to be able to work with a minimal set of modeling elements, else it might be difficult to continuously parse them from the source code.

To study the research questions we selected and modeled three major decisions, which represent important aspects in architecting microservices. To illustrate our approach we selected by purpose very different aspects of microservices

architecture, in particular: the decision for an external API, message persistence, and end-to-end tracing. For each of these we hypothesized a number of generic, technology-independent metrics to measure conformance to the respective decisions. For the evaluation of these metrics, we modeled 24 architecture models taken from the practitioner literature and assessed each of them manually regarding its support of the patterns and practices contained in each decision. We then compared the results in depth and statistically over the whole evaluation model set. The results show that a subset of each decision related metrics are quite close to the manual, pattern-based assessment.

This paper is structured as follows: Sect. 2 compares to related work. In Sect. 3 we explain the decisions considered in this paper and the related patterns/practices. Next, we describe the research methods and the tools we have applied in our study in Sect. 4. In Sect. 5 we report how the ground truth data for each decision is calculated. Section 6 introduces our hypothesized metrics. Section 7 describes the metrics calculations results for our models and the results of the ordinal regression analysis. Section 8 discusses the RQs regarding the evaluation results and analyses the threats to validity. Finally, in Sect. 9 we draw conclusions and discuss future work.

2 Related Work

Much research has been conducted in collecting and systematizing microservice patterns. For instance, Richardson [14] collected microservice patterns related to major design and architectural practices. Zimmermann et al. [20] introduce microservice API related patterns. Skowronski [15] collected best practices for event-driven microservice architectures. Microservice fundamentals and best practices are also discussed by Fowler and Lewis [9], and are summarized in a mapping study by Pahl and Jamshidi [11]. Taibi and Lenarduzzi [16] study microservice bad smells, i.e. practices that should be avoided (which would correspond to metrics violations in our work).

Many of the works on service metrics today are focused on runtime properties (see e.g. [13]). A number of studies has used metrics to assess microservice-based software architectures, e.g. [1,12,18], but each is focused on narrow sets of architecture-relevant tenets (e.g. loose coupling), and no general approach for an assessment across different microservice tenets exists. Pautasso and Wilde [12] propose a composite, facet-based metric for the assessment of loose coupling in service-oriented systems. Zdun et al. [18] study the independent deployment of microservices by defining metrics to assess architecture conformance to microservice patterns, focused on two aspects: independent deployment and shared dependencies of services. Bogner et al. [1] propose a maintainability quality model which combines eleven easily extracted code metrics into a broader quality assessment. Engel et al. [3] also propose a method of using real-time system communication traces to extract metrics on conformance to recommended microservice design principles such as loose coupling and small service size.

These studies focus on treating microservice architectures as a question of components and connectors, factoring in the technologies used, and producing

assessments that combine different assessment parameters (i.e. metrics). Such metrics, if automatically collected, can be utilized as part of larger assessment models/frameworks during design and development time. Our work broadly follows the same approach, but extends it to different architecture tenets relevant to microservice-specific design decisions. Once metrics can be checked automatically, our approach can be classified as a metrics-based, microservice-specific approach for software architecture conformance checking. In general, approaches for architecture conformance checking are often based on automated extraction techniques [5,17]. Techniques that are based on a broad set of microservice-related metrics to cover multiple microservice tenets do not yet exist.

3 Background

External API Decision. One central decision in microservice-based systems is how the external API is offered to clients. This is tightly coupled to the loose coupling, releasability, independent development and deployment, and continuous delivery tenets, as it determines the coupling between client and internal system concerns. In some service-based systems, the *clients can call into system services directly*, meaning high coupling and thus difficulties in releasing, developing, and deploying the clients and system services independently of each other. A better decoupling level might be reached through an *API Gateway* [14], a pattern that describes a common entry point for the system through which all requests are routed. It is a specialized variant of a *Reverse Proxy*, which covers only the routing aspects of an *API Gateway* but not further API abstractions such as authentication, rate limiting, and so on (see [20]). A variant of *API Gateway* for servicing different types of clients (e.g., mobile and desktop clients) is the *Backends for Frontends* pattern [14], which offers a fine-grained API for each specific type of client. A variant where clients can call into system services directly, but are still decoupled is *API Composition* [14], i.e. a service which can invoke other microservices and provides an API for the connected services.

Inter-service Message Persistence Decision. In many business-critical microservice systems, an important concern is that no messages get lost. This concern directly influences the communication between services, and, depending on which option is chosen, the coupling between services, their releasability, their independent development and deployment, as well as their continuous delivery are impacted. Many systems choose communication means that offer *no inter-service message persistence*. Some patterns better support the related aspects of the microservice tenets: The *Messaging* pattern [7] describes service communication, in which persistent message queuing is used to store a producer's messages until the consumer receives them. Many *Stream Processing* [15] components (e.g. Apache Kafka) offer a very similar message persistence level. These solutions offer optimal inter-service message persistence, in the sense that the technology is designed for providing support for it. Some other solutions applied in the microservice field can be used (or adapted) to support it: *Interaction through a*

Shared Database, even though frowned upon with regard to other microservice tenet aspects, supports some level of message persistence as well, but not the automated support of *Messaging*. A more microservice-style technique that supports this level of database-based persistence is the combination of the *Outbox* and the *Transaction Log Tailing* patterns [14] in which each service that sends messages has an outbox database table. As part of the database transaction, the service sends messages by inserting them into the outbox table. A message relay component reads the outbox table and publishes the messages to a message broker. Using the *Event Sourcing* pattern [14] every change to the state of the system should be contained in an event object and stored sequentially in order to be accessible over time. The events are persisted in an event store. This way at least a temporary message persistence is achieved.

End-to-End Tracing Decision. Logging and monitoring are standard practices for creating observability of microservices. As microservice architectures are used for highly distributed and polyglot systems with complex interactions, many of them go one step further and realize end-to-end tracing. It supports tracing and monitoring tenets directly, as well as understandability concerns during independent development and deployment, mastering complexity of highly decoupled services, and thus indirectly releasability and continuous delivery. Like in the other decisions, one option is to offer *No Tracing Support*. In contrast, *Distributed Tracing* [14] is a method used to profile and monitor applications through recording traces on the distributed components. It can either be supported on the microservices of a system, on the gateways of a system, or on both. If both support *Distributed Tracing*, this is optimal, as all relevant traces in ingress, egress, and inter-service communication can be recorded. If it is not supported, a lower level of tracing and monitoring can be reached by routing the service communication through a central component, such as a *Publish/Subscribe* or *Message Broker* component [7]. This can also be achieved if all internal inter-service communication is routed through the *API Gateway*, or if *Event Sourcing* or *Event Logging* [14,15] are used, which store all events temporarily. None of the later techniques has the same level of support as *Distributed Tracing*, but all of them can – with some programming or manual effort – be used to reconstruct traces.

4 Research and Modeling Methods

4.1 Model Selection Methods

This study focuses on architecture conformance to microservice patterns and practices. To be able to study this, we first performed an iterative study of a variety of microservice-related knowledge sources, and we refined a metamodel which contains all the required elements to help us reconstruct existing microservice-based systems. For problem investigation and as an evaluation model set for eventually creating a ground truth for our study, we have gathered a number of microservice-based systems, summarized in Table 1. Each of them is either taken directly from a system published by practitioners (on GitHub

and/or practitioner blogs) or a system variant adapted according to discussions in the relevant literature. The systems were taken from 9 independent sources. They were developed by practitioners with microservice experience, and they provide a good representation of the microservices best practices summarized in Sect. 3. We performed a fully manual static code analysis for those models where the source code was available (i.e. 7 of our 9 sources; two were modeled based on documentation created by the practitioners). The result is a set of precisely modeled component models of the software systems (modeled using the techniques described below). Variations were modeled to cover the complete design space of our three decisions described in Sect. 3, according to the referenced practitioner sources. Apart from the variations described in Table 1 all other system aspects remained the same as in the base models. This resulted in a total of 24 models summarized in Table 1. We assume that our evaluation models are close to models used in practice and real-world practical needs for microservices. As many of them are open source systems with the purpose of demonstrating practices, they are at most of medium size, though.

Table 1. Selected models: size, details, and sources

Model ID	Model size	Description/Source
BM1	10 components 14 connectors	Banking-related application based on CQRS and event sourcing (from https://github.com/cer/event-sourcing-examples)
BM2	8 components 9 connectors	Variant of BM1 which uses direct RESTful completely synchronous service invocations instead of event-based communication
BM3	8 components 9 connectors	Variant of BM1 which uses direct RESTful completely asynchronous service invocations instead of event-based communication
CO1	8 components 9 connectors	The common component model E-shop application implemented as microservices directly accessed by a Web frontend (from https://github.com/cocome-community-case-study/cocome-cloud-jee-microservices-rest)
CO2	11 components 17 connectors	Variant of CO1 using a SAGA orchestrator on the order service with a message broker. Added support for Open Tracing. Added an API gateway
CO3	9 components 13 connectors	Variant of CO1 where the reports service does not use inter-service communication, but a shared database for accessing product and store data. Added support for Open Tracing
CI1	11 components 12 connectors	Cinema booking application using RESTful HTTP invocations, databases per service, and an API gateway (from https://codeburst.io/build-a-nodejs-cinema-api-gateway-and-deploying-it-to-docker-part-4-703c2b0dd269)
CI2	11 components 12 connectors	Variant of CI1 routing all interservice communication via the API gateway
CI3	10 components 11 connectors	Variant of CI1 using direct client to service invocations instead of the API gateway

(contniued)

Table 1. *(contniued)*

Model ID	Model size	Description/Source
CI4	11 components 12 connectors	Variant of CI1 with a subsystem exposing services directly to the client and another subsystem routing all traffic via the API gateway
EC1	10 components 14 connectors	E-commerce application with a Web UI directly accessing microservices and an API gateway for service-based API (from https://microservices.io/patterns/microservices.html)
EC2	11 components 14 connectors	Variant of EC1 using event-based communication and event sourcing internally
EC3	8 components 11 connectors	Variant of EC1 with a shared database used to handle all but one service interactions
ES1	20 components 36 connectors	E-shop application using pub/sub communication for event-based interaction, a middleware-triggered identity service, databases per service (4 SQL DBs, 1 Mongo DB, and 1 Redis DB), and backends for frontends for two Web app types and one mobile app type (from https://github.com/dotnet-architecture/eShopOnContainers)
ES2	14 components 35 connectors	Variant of ES1 using RESTful communication via the API gateway instead of event-based communication and one shared SQL DB for all 6 of the services using DBs. However, no service interaction via the shared database occurs
ES3	16 components 35 connectors	Variant of ES1 using RESTful communication via the API gateway instead of event-based communication and one shared database for all 4 of the services using SQL DB in ES1 However, no service interaction via the shared database occurs
FM1	15 components 24 connectors	Simple food ordering application based on entity services directly linked to a Web UI (from https://github.com/jferrater/Tap-And-Eat-MicroServices)
FM2	14 components 21 connectors	Variant of FM1 which uses the store service as an API composition and asynchronous interservice communication. Added Jaeger-based tracing per service
HM1	13 components 25 connectors	Hipster shop application using GRPC interservice connection and OpenCensus monitoring & Tracing for all but one services as well as on the gateway (from https://github.com/GoogleCloudPlatform/microservices-demo)
HM2	14 components 26 connectors	Variant of HM1 that uses publish/subscribe interaction with event sourcing, except for one service, and realizes the tracing on all services
RM	11 components 18 connectors	Restaurant order management application based on SAGA messaging and domain event interactions. Rudimentary tracing support (from https://github.com/microservices-patterns/ftgo-application)
RS	18 components 29 connectors	Robot shop application with various kinds of service interconnections, data stores, and Instana tracing on most services (from https://github.com/instana/robot-shop)
TH1	14 components 16 connectors	Taxi hailing application with multiple frontends and databases per services from (https://www.nginx.com/blog/introduction-to-microservices/)
TH2	15 components 18 connectors	Variant of TH1 that uses publish/subscribe interaction with event sourcing for all but one service interactions

4.2 Metrics Definition, Ground Truth Calculation, and Statistical Evaluation Methods

To measure conformance to the respective patterns and practices in the design decisions from Sect. 3, we defined a set of metrics for each microservice decision associated to the decision's options, i.e. at least one metric per major decision option. Based on the manual assessment of the models from Table 1, we derived a ground truth for our study (the ground truth and its calculation rules are described in Sect. 5). The ground truth is established by objectively assessing whether each decision option is supported, partially supported, or not supported. By combining the outcome of all options of a decision, we then derived an ordinal assessment on how well the decision is supported in each model, using the scale: [++: very well supported, +: well supported, *o*: neutral, −: badly supported, −−: very badly supported]. Our scale does not assume equal distances (i.e. it is not a Likert scale), but it assumes the given order. We then used the ground truth data to assess how well the hypothesized metrics can possibly predict the ground truth data by performing an ordinal regression analysis.

Ordinal regression is a widely used method for modeling an ordinal response's dependence on a set of independent predictors. For the ordinal regression analysis we used the *lrm* function from the *rms* package in R [4].

4.3 Methods for Modeling Microservice Component Architectures

From an abstract point of view, a microservice-based system is composed of components and connectors with a set of component types and a set of connector types. Our paper has the goal to automate metrics calculation and assessment based on the component model of a microservice system. That is, if the system is manually modeled or the model can be derived automatically from the source code, our approach is applicable. For modeling microservice architectures we followed the method reported in our previous work [18]. All the code and models used in and produced as part of this study have been made available online for reproducibility[1].

5 Ground Truth Calculations for the Study

In this section, we report for each of the decisions from Sect. 3 how the ground truth data is calculated based on manual assessment whether each of the relevant patterns is either Supported (**S** in Table 2), Partially Supported (**P** in Table 2), or Not-Supported (**N** in Table 2). The ordinal results of those assessments are then reported in the Assessments rows of Table 2.

Following the argumentation, which decision option explained in Sect. 3 has which impact on the *External API Decision* related tenets, we can derive the following scoring scheme for our ground truth assessment of this decision:

[1] https://doi.org/10.5281/zenodo.3999477.

Table 2. Ground truth data

External API																									
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Reverse Proxy	S	S	S	N	S	N	S	S	N	P	P	P	P	S	S	S	N	N	N	N	S	S	P	P
API Gateway	S	S	S	N	S	N	S	S	N	P	P	P	P	S	S	S	N	N	N	N	S	S	P	P
Backends for Frontends	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	S	N	N	N	N	N	N	N	N
API Composition	N	N	N	N	N	N	N	N	P	P	N	N	N	N	N	N	N	S	S	S	N	N	P	P
Assessments	++	++	++	--	++	--	++	++	-	o	o	o	o	++	++	++	--	+	+	+	++	++	o	o
Persistent Messaging for Inter-Service Communication																									
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Messaging or Persistent PubSub	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	P	N	N
Shared Database Interaction	N	N	N	N	N	S	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N
Outbox and Trans. Log Tailing	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Event Sourcing	S	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	N	N	N	P	N	N	N	P
All Service Comm. Persistent	S	N	N	N	S	S	N	N	N	N	N	S	N	N	N	N	N	N	N	S	N	N	N	P
Assessments	+	--	--	--	++	+	--	--	--	--	--	+	-	--	--	--	--	--	--	-	++	o	--	o
End-to-End Tracing																									
	BM1	**BM2**	**BM3**	**CO1**	**CO2**	**CO3**	**CI1**	**CI2**	**CI3**	**CI4**	**EC1**	**EC2**	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
Distributed Tracing on Services	N	N	N	N	S	S	N	N	N	N	N	N	N	N	N	N	N	S	P	S	P	P	N	N
Distributed Tracing on Gatew.	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	N	N	N	N
Pub/Sub, Messaging	S	N	N	N	S	N	N	N	N	N	N	S	N	P	N	N	N	N	N	P	S	P	N	S
Inter-service comm. via Gatew.	N	S	S	N	N	N	P	S	N	P	S	N	N	N	P	P	N	N	N	N	N	N	N	N
Event Sourcing/Logging	S	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	N	N	N	P	S	N	N	S
Assessments	o	-	-	--	++	++	--	-	--	--	-	o	--	--	--	--	--	+	+	++	o	o	--	o

– ++: All client traffic is routed through an *API Gateway* or *Backends for Frontends*.
– +: All client-connected services provide *API Composition* or only *Reverse Proxy* capabilities.
– **o**: Some client traffic is routed through *API Gateway* or *Backends for Frontends*.
– **-**: Some client-connected services provide *API Composition* or only *Reverse Proxy* capabilities.
– **--**: All client traffic is directly connected to backend services and no *API Composition* happens.

From the argumentation for the *Inter-service Message Persistence Decision*, we can derive the following scoring scheme for our ground truth assessment:

– +: *Message Brokers* or a persistent *Publish/Subscribe* or *Stream Processing* component are used for all inter-service communication.
– +: All interservice communication is persisted by some combination of partial *Message Brokers*, persistent *Publish/Subscribe*, or persistent *Stream Processing* or partial or full coverage with *Shared Database*, *Event Sourcing*, *Outbox/ Transaction Log Tailing*.
– **o**: A part of the interservice communication is persisted by partial coverage with *Message Brokers*, persistent *Publish/Subscribe*, or persistent *Stream Processing*.
– **-**: A part of the interservice communication is persisted by partial coverage with *Shared Database*, *Event Sourcing*, *Outbox/ Transaction Log Tailing*.

– **--**: None of the above is supported.

Finally, from the argumentation for the*End-to-end Tracing Decision*, we can derive the following scoring scheme for our ground truth assessment:

– **++**: *Distributed Tracing* is fully supported on all services and gateways.
– **+**: *Distributed Tracing* is fully supported on either the services or the gateways.
– **o**: *Distributed Tracing* is partially supported or *Event Sourcing/Event Logging* are fully supported.
– **-**: *Publish/Subscribe*, *Message Broker*, or *Invocations Routed Via API Gateway* are fully supported for service interactions or those patterns are partially supported and at the same time *Event Sourcing/Event Logging* are supported.
– **--**: None of the above is supported.

6 Metrics

All metrics, unless otherwise noted, are a continuous value with range from 0 to 1, with 1 representing the optimal case where a set of patterns is fully supported, and 0 the worst-case scenario where it is completely absent. For instance, in EC1 client traffic is partially routed through API Gateway resulting CCF = 0.25. The metrics results for each model per decision metric are presented in Table 3.

6.1 Metrics for the External API Decisions

Client-side Communication via Facade utilization metric (CCF). This metric returns the number of the connectors from *Clients* to *Facade* components set in relation to the total number of unique *Client* connectors. This way, we can measure how many unique client links are using the External API used by one of the Facade components (i.e. offered through patterns such as *API Gateway*, *Reverse Proxy*, *Backends for Frontends*).

$$CCF = \frac{Number\,of\,Client\,to\,Facade\,Links}{Number\,of\,Unique\,Client\,Links}$$

In this metric (and in other metrics below), the number of unique client links is defined as follows:

$$\begin{aligned} Number\,of\,Unique\,Client\,Links = \\ max\{Number\,of\,Facades\,Linked\,to\,Clients, \\ Number\,of\,Clients\,Linked\,to\,Facades\} \\ + Number\,of\,Client\,to\,Non-Facade/Non-Client\,Links \end{aligned}$$

As a result, the only decision option remaining is *API Composition*, for which we formulated the APIC metric.

API Composition utilization metric (APIC). In cases that a client is directly connected to services, it is possible that these services offer an *External API* shielding the interfaces of other services that are connected to them. That is, a client can have access to a system service via other services. To detect such cases, we count the routes from the client to system services via other services and set this number in relation to the total number of system services. That gives us the proportion of services that are accessible by clients via other services. We then divide this number with the unique client links to estimate the proportion of clients connected services which are possibly composing an *External API* using *API Composition*.

$$APIC = \frac{\frac{Number\,of\,Client\,to\,Services\,via\,other\,Services\,Routes}{Total\,Number\,of\,Services}}{Number\,of\,Unique\,Client\,Links}$$

6.2 Metrics for Persistent Messaging for Inter-Service Communication Decision

Service Messaging Persistence utilization metric (SMP). One important aspect in services interconnections is the persistence of the exchanged messages. We defined this metric to measure the proportion of the services interconnections that are made persistent through supporting technology (i.e. *Messaging* or *Stream Processing*).

$$SMP = \frac{Service\,Interconnections\,with\,Messaging\,or\,Stream\,Processing}{Number\,of\,Service\,Interconnections}$$

Shared DataBase utilization metric (SDB). Although a *Shared Database* is considered as an anti-pattern in microservices, there are many systems that use it either partially or completely. The pattern might be beneficial for persistent messaging, but definitely is not the optimal option. To measure its presence in a system, we count the number of interconnections via a *Shared Database* compared to the total number of interconnections. We note that for this metric, our metrics scale is reversed in comparison to the other metrics, because here we detect the presence of an anti-pattern: the optimal result of our metrics is 0, and 1 is the worst-case result.

$$SDB = \frac{Service\,Interconnections\,with\,SharedDB}{Number\,of\,Service\,Interconnections}$$

Outbox/Event Sourcing utilization metric (OES). *Outbox* and *Event Souring* can ensure temporary message persistence. Our metric measures the proportion of the interconnections with *Outbox/Event Sourcing* to the total number of interconnections.

$$OES = \frac{Service\,Interconnection\,with\,Outbox\,or\,Event\,Sourcing}{Number\,of\,Service\,Interconnections}$$

6.3 Metrics for End-to-End Tracing Decision

$$SFT = \frac{Services\,and\,Facades\,Support\,Distributed\,Tracing}{Number\,of\,Services\,and\,Facades}$$

Service Interaction via Central Component utilization metric (SICC) and **Service Interaction with Event Sourcing utilization metric (SIES)**. *Distributed Tracing* can be supported by routing the inter-service communication via a central component (e.g. *Publish/Subscribe, Message Broker* and *API Gateway*). Since *Event Sourcing* also enables tracing by tracking the messages, we distinguish between systems that support *Event Sourcing* (SIES), and systems that do not (SICC).

$$SICC = \frac{Service\,Interaction\,via\,Central\,Component\,w/o\,Event\,Sourcing}{Number\,of\,Service\,Interconnections}$$

Table 3. Metrics calculation results

Metrics	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2
External API												
CCF	1.00	1.00	1.00	0.00	1.00	0.00	1.00	1.00	0.00	0.50	0.25	0.25
APIC	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30	0.10	0.00	0.00
Persistent messaging for inter-service communication												
SMP	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SDB	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
OES	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
End-to-end tracing												
SFT	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
SICC	0.00	1.00	1.00	0.00	1.00	1.00	0.14	1.00	0.00	0.60	1.00	0.00
SIES	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00
Metrics	**EC3**	**ES1**	**ES2**	**ES3**	**FM1**	**FM2**	**HM1**	**HM2**	**RM**	**RS**	**TH1**	**TH2**
External API												
CCF	0.25	1.00	1.00	1.00	0.00	0.00	0.00	0.00	1.00	1.00	0.25	0.25
APIC	0.00	0.00	0.00	0.00	0.25	0.50	0.70	0.70	0.00	0.00	0.12	0.04
Persistent messaging for inter-service communication												
SMP	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.66	0.11	0.00	0.00
SDB	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
OES	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.66
End-to-end tracing												
SFT	0.00	0.00	0.00	0.00	0.00	1.00	0.90	0.90	0.14	0.62	0.00	0.00
SICC	0.00	0.60	0.45	0.45	0.00	0.00	0.00	0.00	1.00	0.11	0.00	0.00
SIES	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.80	0.00	0.00	0.00	0.66

$$SIES = \frac{Service\ Interaction\ via\ Central\ Component\ with\ Event\ Sourcing}{Number\ of\ Service\ Interconnections}$$

7 Ordinal Regression Analysis Results

The metrics calculations for each model per each decision metric are presented in Table 3. The dependent outcome variables are the ground truth assessments for each decision, as described in Sect. 5 and summarized in Table 2. The metrics defined in Sect. 6 are used as the independent predictor variables. The ground truth assessments are ordinal variables, while all the independent variables are measured on a scale from 0.0 to 1.0. The aim of the analysis is to predict the

Table 4. Regression analysis results

Intercepts/Coefficients	Value	Model p-value
External API		
Intercept (≥Badly Supported)	*−3.5690*	*4.423828e−11*
Intercept (≥Neutral)	*−4.5042*	
Intercept (≥Well Supported)	*−10.2692*	
Intercept (≥Very Well Supported)	*−15.7271*	
Metric Coefficient (CCF)	*20.3552*	
Metric Coefficient (APIC)	*18.1419*	
Persistent messaging for inter-service communication		
Intercept (≥Badly Supported)	*−5.6344*	*2.002198e−09*
Intercept (≥Neutral)	*−9.5937*	
Intercept (≥Well Supported)	*−11.2074*	
Intercept (≥Very Well Supported)	*−21.0398*	
Metric Coefficient (SMP)	*94.5503*	
Metric Coefficient (SDB)	*10.4199*	
Metric Coefficient (OES)	*13.3840*	
End-to-end tracing		
Intercept (≥Badly Supported)	*−35.4940*	*4.440892e−15*
Intercept (≥Neutral)	*−53.7947*	
Intercept (≥Well Supported)	*−103.6085*	
Intercept (≥Very Well Supported)	*−135.5906*	
Metric Coefficient (SFT)	*44.6971*	
Metric Coefficient (SICC)	*94.1809*	
Metric Coefficient (SIES)	*125.5634*	

likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics.

Each resulting regression model consists of a *baseline intercept* and the independent variables multiplied by *coefficients*. There are different intercepts for each of the value transitions of the dependent variable ($\geq$*Badly Supported*, $\geq$*Neutral*, $\geq$*Well Supported*, $\geq$*Very Well Supported*), while the coefficients reflect the impact of each independent variable on the outcome. For example, a positive coefficient, such as +5, indicates a corresponding five-fold increase in the dependent variable for each unit of increase in the independent variable; conversely, a coefficient of −30 would indicate a thirty-fold decrease.

In Table 4, we report the p-values for the resulting models, which in all cases are very low, indicating that the sets of metrics we have defined are able to predict the ground truth assessment for each decision with a high level of accuracy.

8 Discussion

8.1 Discussion of Research Questions

For answering **RQ1** and **RQ2**, we suggested a set of generic, technology-independent metrics for each microservice decision, and we associated at least one metric to each major decision option. The ground truth is established by objectively assessing how well a pattern and/or practice is supported in each model, and extrapolating this to how well the broader decision is supported. We formulated metrics to assess a pattern's implementation in each model, and performed an ordinal regression analysis using these metrics as independent variables to predict the ground truth assessment. Our results show that every set of decision-related metrics can predict with high accuracy our objectively evaluated assessment. This suggests that automatic metrics-based assessment of a system's conformance to the tenets embodied in each design decision is possible with a high degree of confidence.

Regarding **RQ3**, we can assess that our microservice meta-model has no need for major extensions and is easy to map to existing modeling practices. More specifically, in order to fully model our evaluation model set, we needed to introduce 25 component types and 38 connector types, ranging from general notions such as the *Service* component type, to very technology-specific classes such as the *RESTful HTTP* connector, which is a subclass of *Service Connector*. Our study shows that for each pattern and practice embodied in each decision and the proposed metrics, only a small subset of the meta-model is required. The decision *External API* requires to model at least the *Service*, *Client*, and the *Facade* component types and the technology-related connector types (e.g. *RESTful HTTP, Synchronous Connector, HTTP, HTTPS*). The *Persistent Messaging for Inter-Service Communication* and *End-to-End Tracing* decisions need a number of additional components (e.g. *Event Sourcing, Stream Processing, Messaging, PubSub*) and the respective connectors (e.g. *Publisher, Subscriber, Message Consumer* and *Messages Producer*) to be modeled.

8.2 Threats to Validity

We deliberately relied on third-party systems as the basis for our study to increase internal validity, thus avoiding bias in system composition and structure. It is possible that our search procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches. Given that our search was not exhaustive, and that most of the systems we found were made for demonstration purposes, i.e. relatively modestly sized, this means that some potential architecture elements were not included in our meta-model. In addition, this raises a possible threat to external validity of generalization to other, and more complex, systems. We nevertheless feel confident that the systems documented are a representative cross-cut of current practices in the field, as the points of variance between them were limited and well attested in the literature. Another potential threat is the fact that the variant systems were derived by the author team. However, this was done according to best practices documented in literature. We made sure only to change specific aspects in a variant and keep all other aspects stable.

Another potential source of internal validity threat is the modeling process itself. The author team has considerable experience in similar methods, and the models of the systems were repeatedly and independently cross-checked, but the possibility of some interpretative bias remains: other researchers might have coded or modeled differently, leading to different models. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study. The ground truth assessment might also be subject to different interpretations by different practitioners. For this purpose, we deliberately chose only a three-step ordinal scale, and given that the ground truth evaluation for each decision is fairly straightforward and based on best practices, we do not consider our interpretation controversial. Likewise, the individual metrics used to evaluate the presence of each pattern were deliberately kept as simple as possible, so as to avoid false positives and enable a technology-independent assessment. As stated previously, generalization to more complex systems might not be possible without modification. But we consider that the basic approach taken when defining the metrics is validated by the success of the regression models.

9 Conclusions and Future Work

In this work we have hypothesized that it is possible to develop a method to automatically assess microservices tenets in microservice decisions based on a microservice system's component model. We have shown that this is possible for microservice decision models comprising patterns and practices as decision options. Our approach first modeled the key aspects of the decision options using a minimal set of component model elements (which could be automatically extracted from the source code). Then we derived at least one metric per decision option and used a small reference model set as a ground truth. We then used

ordinal regression analysis for deriving a predictor model for the ordinal variable. Our statistical analysis shows a high level of accuracy.

While so far many studies on metrics for component model and other architectures exist, the specifics of microservice architectures and their particular tenets have not been studied. As discussed in Sect. 2, only using general metrics does not help much in assessing microservice architectures. Our approach is one of the first that studies a metrics-based assessment of multiple, very different microservice tenets. Our main goal is a continuous assessment, i.e. we envision an impact on continuous delivery practices, in which the metrics are assessed with each delivery pipeline run, indicating improvements, stability, or deteriorations in microservice architecture conformance. With small changes, our approach could also be applied, during early architecture assessment.

As future work, we plan to study more decisions, tenets, and related metrics. We also plan to create a larger data set, thus better supporting tasks such as early architecture assessment in a project.

Acknowledgments. This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 846707; FWF (Austrian Science Fund) project API-ACE: I 4268.

References

1. Bogner, J., Wagner, S., Zimmermann, A.: Towards a practical maintainability quality model for service-and microservice-based systems, pp. 195–198 (2017). https://doi.org/10.1145/3129790.3129816
2. Chen, L.: Microservices: architecting for continuous delivery and DevOps. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 39–397, April 2018. https://doi.org/10.1109/ICSA.2018.00013
3. Engel, T., Langermeier, M., Bauer, B., Hofmann, A.: Evaluation of microservice architectures: a metric and tool-based approach. In: Mendling, J., Mouratidis, H. (eds.) CAiSE 2018. LNBIP, vol. 317, pp. 74–89. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92901-9_8
4. Harrell, F.E.: Regression Modeling Strategies. SSS. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19425-7
5. Guo, G.Y., Atlee, J.M., Kazman, R.: A software architecture reconstruction method. In: Donohoe, P. (ed.) Software Architecture. ITIFIP, vol. 12, pp. 15–33. Springer, Boston, MA (1999). https://doi.org/10.1007/978-0-387-35563-4_2
6. Haselböck, S., Weinreich, R., Buchgeher, G.: Decision models for microservices: design areas, stakeholders, use cases, and requirements. In: Lopes, A., de Lemos, R. (eds.) ECSA 2017. LNCS, vol. 10475, pp. 155–170. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-65831-5_11
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Boston (2003)
8. Knoche, H., Hasselbring, W.: Drivers and barriers for microservice adoption - a survey among professionals in Germany. Enterp. Model. Inf. Syst. Archit. (EMISAJ) Int. J. Conceptual Model. **14**(1), 1–35 (2019). https://doi.org/10.18417/emisa.14.1

9. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term, March 2004. http://martinfowler.com/articles/microservices.html
10. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, Sebastopol (2015)
11. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science, pp. 137–146 (2016)
12. Pautasso, C., Wilde, E.: Why is the web loosely coupled?: a multi-faceted metric for service design. In: 18th International Conference on World Wide Web, pp. 911–920. ACM (2009)
13. Pietrantuono, R., Russo, S., Guerriero, A.: Run-time reliability estimation of microservice architectures. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE), pp. 25–35, October 2018. https://doi.org/10.1109/ISSRE.2018.00014
14. Richardson, C.: A pattern language for microservices (2017). http://microservices.io/patterns/index.html
15. Skowronski, J.: Best practices for event-driven microservice architecture (2019). https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk
16. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018). https://doi.org/10.1109/MS.2018.2141031
17. Van Deursen, A., Hofmeister, C., Koschke, R., Moonen, L., Riva, C.: Symphony: view-driven software architecture reconstruction. In: 4th Working IEEE/IFIP Conference on Software Architecture (WICSA 2004), pp. 122–132. IEEE (2004)
18. Zdun, U., Navarro, E., Leymann, F.: Ensuring and assessing architecture conformance to microservice decomposition patterns. In: Maximilien, M., Vallecillo, A., Wang, J., Oriol, M. (eds.) ICSOC 2017. LNCS, vol. 10601, pp. 411–429. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-69035-3_29
19. Zimmermann, O.: Microservices tenets. Comput. Sci. Res. Dev. 301–310 (2016). https://doi.org/10.1007/s00450-016-0337-0
20. Zimmermann, O., Stocker, M., Zdun, U., Luebke, D., Pautasso, C.: Microservice API patterns (2019). https://microservice-api-patterns.org