Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices

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. Microservices are the go-to architectural style for building applications that are polyglot, support high scalability, independent development and deployment, and are rapidly adaptable to changes. Among the core tenets for a successful microservice architecture is high independence of the individual microservices, i.e. loose coupling. A number of patterns and best practices are well-established in the literature, but most actual microservice-based systems do not, as a whole or in part, conform to them. Assessing this conformance manually is not realistically possible for large-scale systems. This study aims to provide the foundations for an automated approach for assessing conformance to coupling-related patterns and practices specific for microservice architectures. We propose a model-based assessment based on generic, technology-independent metrics, connected to typical design decisions encountered in microservice architectures. We demonstrate and assess the validity and appropriateness of these metrics by performing an assessment of the conformance of real-world systems to patterns through statistical methods.

1 Introduction

Microservice architectures [14,15,22] describe an application as a collection of autonomous and loosely coupled services, typically modeled around a domain. Key microservice tenets are development in independent teams, cloud-native technologies and architectures, polyglot technology stacks including polyglot persistence, lightweight containers, loosely coupled service dependencies, high releasability, and continuous delivery [22]. Many architectural patterns that reflect recommended "best practices" in a microservices context have already been published in the literature [18,19,23]. The fact that microservice-based systems are complex and polyglot means that an automatic or semi-automatic assessment of their conformance to these patterns is difficult: real-world systems feature combinations of these patterns, and different degrees of violations of the

A. Jansen et al. (Eds.): ECSA 2020, LNCS 12292, pp. 3–20, 2020.
https://doi.org/10.1007/978-3-030-58923-3_1

same; and different technologies in different parts of the system implement the patterns in different ways, making the automatic parsing of code and identification of the patterns a haphazard process.

This work focuses on describing a method for assessing architecture conformance to coupling-related patterns and practices in microservice architectures. Coupling between microservices is caused by existence of dependencies, e.g. whenever one service calls another service to fulfill a request or share data. Loose coupling is an established topic in service-oriented architectures [22] but the application to the specific context of microservice architectures has not, to our knowledge, been examined so far.

Strong coupling is conflicting with some of the key microservice tenets mentioned above. In particular, releasability, which is a highly desirable characteristic in modern systems due to the emergence of DevOps practices, relies on the rapid and independent release of individual microservices, and is compromised by strong dependencies between them. For the same reason, development in independent teams becomes more difficult, and independent deployment of individual microservices in lightweight containers is also impeded. This work covers three broad coupling aspects: *Coupling through Databases*, resulting from reliance on commonly accessed data via shared databases; *Coupling through Synchronous Invocations*, resulting from synchronous communication between individual services; and *Coupling through Shared Services*, which arises through the dependence on common shared services (for details see Sect. 3).

In reality, of course, no microservice system can support *all* microservice tenets well at the same time. Rather the *architectural decisions* for or against the use of specific patterns and practices must reflect a trade-off between ensuring the desired tenets and other important quality attributes [12,22]. From these considerations, this paper aims to study the following research questions:

- **RQ1** How can we automatically assess conformance to loose coupling-related patterns and practices in the context of microservice architecture decision options?
- **RQ2** How well do measures for assessing coupling-related 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?

In pursuing of these questions, we surveyed the relevant literature (Sect. 2) and gathered knowledge sources about established architecture practices and patterns, their relations and tenets in form of a *qualitative study on microservice architectures*. This enabled us to create a meta-model for the description of microservice architectures, which was verified and refined through iterative application in modelling a number of real-world systems, as outlined in Sect. 4. We manually assessed all models and model variants on whether each decision option is supported, thereby deriving an objective *ground truth* (Sect. 5). As the basis for an automatic assessment, we defined a number of generic, technology-independent metrics to measure architecture conformance to the decision options, i.e. at least one metric per major decision option (Sect. 6).

These metrics (and combinations thereof) were applied on the models and model variants to derive a numeric assessment, and then compared to the ground truth assessment via an ordinal regression analysis (Sect. 7). Section 8 discusses the results of our approach, as well as its limitations and potential threats to validity. Finally, in Sect. 9 we draw our conclusions and discuss options for future work.

2 Related Work

Many studies focus on best practices for microservice architectures. Richardson [18] has published a collection of microservice patterns related to major design and architectural practices. Patterns related to microservice APIs have been introduced by Zimmermann et al. [23], while Skowronski [19] collected best practices for event-driven microservice architectures. Microservice fundamentals and best practices are also discussed by Fowler and Lewis [14], and are summarized in a mapping study by Pahl and Jamshidi [16]. Taibi and Lenarduzzi [20] study microservice "bad smells", i.e. practices that should be avoided (which would correspond to violations in our work).

Many software metrics-related studies for evaluating the system architecture and individual architectural components exist, but most of them are not specific to the microservices domain. Allen et al. [1,2] study component metrics for measuring a number of quality attributes, e.g. size, coupling, cohesion, dependencies of components, and the complexity of the architecture. Additional studies for assessing quality attributes related to coupling and cohesion have been proposed and validated in the literature [3,4,6,11]. Furthermore, a small number of studies [5,17,21] propose metrics specifically for assessing microservice-based software architectures. Although these works study various aspects of architecture, design metrics, and architecture-relevant tenets such as coupling and independent deployment, their approach is usually generic. None of the works covers all the related software aspects for measuring coupling in a microservice context: the use of databases, system asynchronicity, and shared components. This is the overarching perspective of our work, and the chief contribution of this paper.

3 Decisions

In this section, we briefly introduce the three coupling-related decisions along with their decision options (i.e. the relevant patterns and practices) which we study in this paper. We also discuss the impact on relevant microservice tenets, which we later on use as an argumentation for our manual ground truth assessment in Sect. 5.

Inter-Service Coupling Through Databases. One important decision in microservice-based systems is data persistence, which needs to take into account qualities such as reliability and scalability, but also adhere to microservice-specific best practices, which recommend that each microservice should be

loosely coupled and thus able to be developed, deployed, and scaled independently [14]. At one extreme of the scale, one option is *No Persistent Data Storage*, which is applicable only for services whose functions are performed on transient data. Otherwise, the most recommended option is the *Database per Service* pattern [18]: each service has its own database and manages its own data independently. Another option, which negatively affects *loose coupling*, is to use a *Shared Database* [18]: a service writes its data in a common database and other services can read these data when required. There are two different ways to implement this pattern: in *Data Shared via Shared Database* multiple services share the same table, resulting in a strongly coupled system, whereas in *Databased Shared but no Data Sharing* each service writes to and reads from its own tables, which has a lesser impact on coupling.

Inter-Service Coupling Through Synchronous Invocations. Service integration is another core decision when building a microservice-based system. A theoretically optimal system of independent microservices would feature no communication between them. Of course, services need to communicate in reality, and so the question of integrating them so as to not result in tight inter-service coupling becomes paramount. The recommended practice is that communication between the microservices should be, as much as possible, asynchronous. This can be achieved through several patterns which are widely implemented in typical technology stacks: the *Publish/Subscribe* [13] pattern, in which services can subscribe to a channel to which other services can publish; the use of a *Messaging* [13] middleware, which decouples communication by using a queue to store messages sent by the producer until they are received by the consumer; the *Data Polling* [18] pattern, in which services periodically poll other services for data changes; and the *Event Sourcing* [18] pattern, that ensures that all changes to application state are stored as a sequence of events; *Asynchronous Direct Invocation* technique, in which services communicate asynchronously via direct invocations. Applying these patterns ensures loose coupling (to different degrees), and increases the system reliability.

Inter-Service Coupling Through Shared Services. Many of the microservice patterns focus on system structure, i.e. avoiding services sharing other services altogether, or at least not in a strongly coupled way. An optimal system in terms of architecture quality should not have any shared service. In reality, this is often not feasible, and in larger systems service sharing leads to chains of transitive dependencies between services. This is problematic when a service is unaware of its transitive dependencies, and of course for the shared service itself, where the needs of its dependents must always be taken into account during its evolution. We define three cases: a *Directly Shared Service* is a microservice which is directly linked to and required by more than one other service; a *Transitively Shared Service* is a microservice which is linked to other services via at least one intermediary service; and a *Cyclic Dependency* [10] which is formed when there is a direct or transitive path that leads back to its origin, i.e. that allows a service to be ultimately dependent on itself after a number of intermediary services. Cyclic dependencies often emerge inadvertently through increasing complexity

over the system's lifecycle, and require extensive refactoring to resolve. All three cases are inimical to the principle of *loose coupling* as well as to system qualities such as *performance*, *modifiability*, *reusability*, and *scalability*.

4 Research and Modeling Methods

In this section, we summarize the research and modeling methods applied in our study. The code and models used in and produced as part of this study have been made available online for reproducibility[1].

4.1 Research Method

Figure 1 shows the research steps from initial data collection to final data analysis. For the data collection phase we conducted a multi-vocal literature study using *web resources*, *public repositories*, and *scientific papers* as sources [9]. We then analyzed the data collected using qualitative methods based on *Grounded Theory* [7] coding methods, such as open and axial coding, and extracted the three core architectural decisions described in the previous section along with their corresponding decision drivers and impacts. As data for our further research we used generated models taken from the *Model Generation* process, described below. We defined a rating scheme for systematic assessment based on support or violation of core practices and tenets. From these we derived a ground truth for our study (the ground truth and its calculation rules are described in Sect. 5) as well as a set of metrics for automatically calculating conformance to each individual pattern or practice per decision. 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, which is applicable in a variety of domains. For the ordinal regression analysis we used the *lrm* function from the *rms* package in R [8].

4.2 Model Generation

We began by performing an iterative study of a variety of microservice-related knowledge sources, and we gradually refined a meta-model which contains all the required elements to help us reconstruct existing microservice-based systems. In order to investigate the ontology, and to evaluate the meta-model's efficiency, we gathered a number of microservice-based systems, summarized in Table 1. Each is either a system published by practitioners (on GitHub and/or practitioner blogs) or a system variant adapted from a published example according to discussions in the relevant literature in order to explore the possible decision space. Apart from the specific variations described in Table 1 all other system aspects remained the same as in the base models.

[1] https://bit.ly/2WmFP3N.

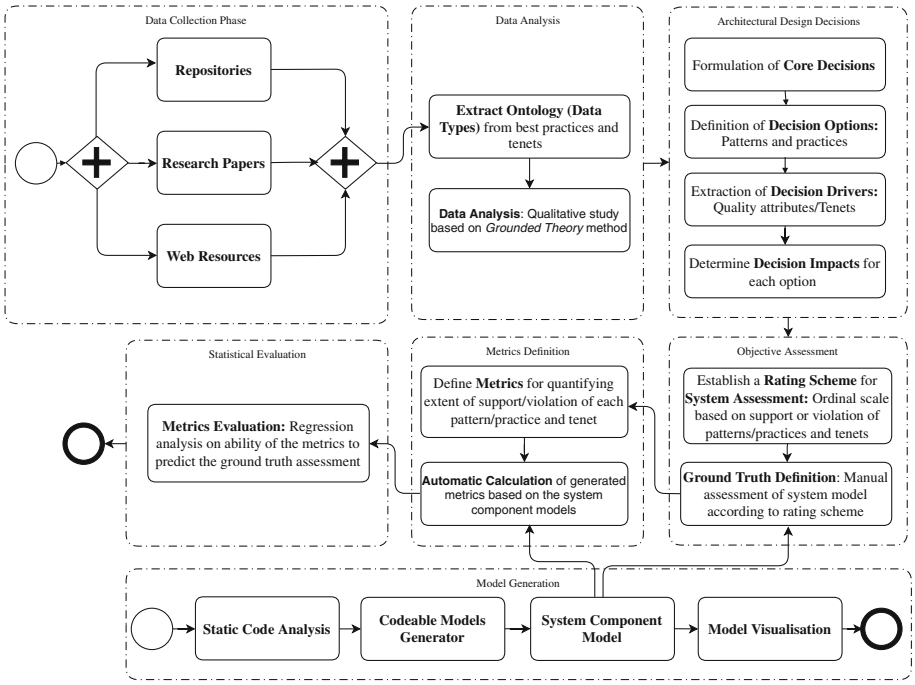

Fig. 1. Overview diagram of the research method followed in this study

The systems were taken from 9 independent sources in total. They were developed by practitioners with microservice experience, and they are representative of the best practices summarized in Sect. 3. We performed a fully manual static code analysis for those models where the source code was available (7 of our 9 sources; two were modeled from documentation published by the practitioners).

To create our models, we used our existing modeling tool CodeableModels[2], a Python implementation for the precise specification of meta-models, models, and model instances in code. Based on CodeableModels, we specified meta-models for components, connectors and relationships. We then manually created model instances for each of the systems in Table 1. In addition, we realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models.

The result is a set of precisely modeled component models of the examined software systems (modeled using the techniques described below). This resulted in a total of 27 models summarized in Table 1. We assume that our evaluation systems are, or reflect, real-world practical examples of microservice architectures. As many of them are open source systems with the purpose of demonstrating practices or technologies, they are at most of medium size and modest complexity, though.

[2] https://github.com/uzdun/CodeableModels.

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 for each component and a set of connector types for each connector. For modeling microservice architectures we followed the method reported in our previous work [21].

5 Ground Truth Calculations

In this section, we present and describe the calculation of the ground truth assessment for each of the decisions from Sect. 3. The results of those assessments are reported in Table 2. The assessment begins with a manual evaluation by the authors on whether each of the relevant patterns (decision options) is either Supported, Partially Supported, or Not Supported (**S**, **P**, **N** in Table 2). Based on this and informed by the description of the impacts of the various decision options in Sect. 3, we combined the outcome of all decision options to derive an ordinal assessment on how well the decision as a whole is supported in each model, using the ordinal scale: [++: very well supported, +: well supported, **o**: neutral, **-**: badly supported, **--**: very badly supported]. This was done according to best practices documented in literature. For instance, following the ordinal scale the assessment for the model BM1 is +: well supported, since a) option *Database per Service* is not supported, b) some services have a shared database, but c) they do not share data via the shared database.

For the *Inter-Service Coupling through Databases* decision, we derive the following scoring scheme for our ground truth assessment:

- **++:** All services (which require data persistence) have individual databases *Database per Service*.
- **+:** Some services have *Shared Databases* and no *Data Shared via the Shared Databases*.
- **o:** All services have *Shared Databases* and no *Data Shared via the Shared Databases*.
- **-:** Some services have *Shared Databases* and *Data Shared via the Shared Databases*.
- **--:** All services have *Shared Databases* and *Data Shared via the Shared Databases*.

From the *Inter-Service Coupling through Synchronous Invocations* decision, we derive the following scoring scheme for our ground truth assessment:

- **++:** All services communicate asynchronously via *Message Brokers* or *Publish/Subscribe* or *Stream Processing*
- **+:** All services communicate asynchronously via *API Gateway* or *HTTP Polling* or *Direct Asynchronous calls*, or (some) via *Message Brokers* or *Publish/Subscribe* or *Stream Processing*.
- **o:** None or some services communicate asynchronously and all other services communicate asynchronously via *Data Sharing* (e.g. Shared DB).

Table 1. Overview of modelled systems (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.
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. 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.
FM3	13 components 15 connectors	Variant of FM1 which demonstrates a cyclic dependency case, uses the store service as an API composition and asynchronous inter-service communication
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.
RM1	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`).
RM2	14 components 14 connectors	Variant of RM1 which contains transitively shared services, API Gateway for client services communication, database per service and direct communication between service.
RM3	14 components 15 connectors	Variant of RM1 which demonstrates a cyclic dependency case, API Gateway for client services communication, database per service and direct communication between service.
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.

Table 2. Ground truth assessment results

	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2	EC3	ES1	ES2	ES3	FM1	FM2	FM3	HM1	HM2	RM1	RM2	RM3	RS	TH1	TH2
Database-based inter-service coupling																											
Database per Service	N	S	S	S	S	P	S	S	S	S	S	S	N	S	N	P	S	S	S	S	S	N	S	S	N	S	S
Shared Database	P	N	N	N	N	P	N	N	N	N	N	N	S	N	S	P	N	N	N	N	N	S	N	N	S	N	N
Data Shared via Shared DB	N	N	N	N	N	P	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Assessments	+	++	++	++	++	-	++	++	++	++	++	++	--	++	o	+	++	++	++	++	++	o	++	++	o	++	++
Inter-service coupling through synchronous invocations																											
Asynchronous Direct Interconnections	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	S	P	P	N	N	N	P	N	N
PubSub/Event Sourcing Interconnections	S	N	N	N	N	N	N	N	N	N	N	S	N	P	N	N	N	N	N	N	S	P	N	N	N	N	P
Asynch Inter-communication via API GW	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Shared Database Interconnections	N	N	N	N	N	P	N	N	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Messaging Interconnections	N	N	N	N	S	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	P	N	N	P	N	N
Assessments	++	--	-	--	++	-	--	--	--	--	--	++	o	+	--	--	--	+	+	-	++	+	--	--	o	--	+
Inter-service coupling through shared services																											
Direct Service Sharing	N	N	N	N	N	N	P	N	P	N	N	N	N	P	P	P	P	P	P	P	N	N	N	N	P	P	N
Transitively Shared Services	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	P	N	N	N	S	N	N	N	N
Cyclic Dependencies	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	S	N	N	N	N	S	N	N	N
Assessments	++	++	++	++	++	++	o	++	o	++	++	++	++	o	o	o	o	o	--	o	++	++	o	--	o	o	++

- **-:** None or some services communicate asynchronously, none or some communicate asynchronously via *Data Sharing*, some services communicate synchronously.
- **--:** All services communicate synchronously.

Finally, from the *Inter-Service Coupling through Shared Services* decision, we derive the following scoring scheme for our ground truth assessment:

- **++:** None of the services is a *Directly Shared Service* or *Transitively Shared Service* and no *Cyclic Dependencies* exist.
- **+:** Some of the services are *Transitively Shared Services*, but none are *Directly Shared Services* and no *Cyclic Dependencies* exist.
- **o:** Some or none of the services are *Transitively Shared Services* and some are *Directly Shared Services*, but no *Cyclic Dependencies* exist.
- **-:** Some of the services are *Transitively Shared Services* and all other services are *Directly Shared Services*, but no *Cyclic Dependencies* exist.
- **--:** There are *Cyclic Dependencies* or all the services are *Transitively Shared Components* and all the services are *Directly Shared Components*.

6 Metrics

In this section, we describe the metrics we have hypothesized for each of the decisions described in Sect. 3. 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.

6.1 Metrics for Inter-Service Coupling Through Databases Decision

Database Type Utilization (DTU) Metric**.** This metric returns the number of the connectors from *Services* to *Individual Databases* in relation to the total number of *Service-to-Database* connectors. This way, we can measure how many services are using individual databases.

$$DTU = \frac{Database\,per\,Service\,Links}{Total\,Service\text{-}to\text{-}Database\,Links}$$

Shared Database Interactions (SDBI) Metric**.** Although a *Shared Database* is considered as an anti-pattern in microservices, there are many systems that make use of it either partially or completely. To measure its presence in a system, we count the number of interconnections via a *Shared Database* compared to the *total number of service interconnections*.

$$SDBI = \frac{Service\,Interconnections\,with\,Shared\,Database}{Total\,Service\,Interconnections}$$

6.2 Metrics for Inter-Service Coupling Through Synchronous Invocations Decision

Service Interaction via Intermediary Component (SIC) Metric**.** We defined this metric to measure the proportion of service interconnections via asynchronous relay architectures such as *Message Brokers*, *Publish/Subscribe*, or *Stream Processing*. These represent the best current practices, and are not exhaustive; should any new architectures emerge, these should be added to this list.

$$SIC = \frac{Service\,Interconnections\,via\,[Message\,Brokers \mid Pub/Sub \mid Stream]}{Total\,Service\,Interconnections}$$

Asynchronous Communication Utilization (ACU) Metric**.** This metric measures the proportion of the sum of asynchronous service interconnections (via *API Gateway/HTTP Polling/Direct calls/Shared Database*) to the total number of service interconnections.

$$ACU = \frac{Asynchronous\,Service\,Interconnections\,via\,[API \mid Polling \mid Direct\,Calls \mid Shared\,DB]}{Total\,Service\,Interconnections}$$

6.3 Metrics for Inter-Service Coupling Through Shared Services Decision

Direct Service Sharing (DSS) Metric**.** For measuring DSS we count all the directly shared services and set this number in relation to the total number of system services. To this add all the shared services connectors in relation to the total number of services interconnections. This gives us the proportion of the directly shared elements in the system.

$$DSS = \frac{\frac{Shared\,Services}{Total\,Services} + \frac{Shared\,Services\,Connectors}{Total\,Service\,Interconnections}}{2}$$

Transitively Shared Services (TSS) Metric**.** For measuring TSS we count all the transitively shared services and set this number in relation to the total number of system services. To this we add all the transitively shared service connectors in relation to the total number of service interconnections. This gives us the proportion of the transitively shared elements in the system.

$$TSS = \frac{\frac{Transitively\,Shared\,Services}{Total\,Services} + \frac{Transitively\,Shared\,Services\,Connectors}{Total\,Service\,Interconnections}}{2}$$

Cyclic Dependencies Detection (CDD) Metric**.** Let $SG = (S, C)$ be the service graph, S the set of service nodes, and C the set of connector edges in a microservice model. Based on the generic definition of closed paths, we define a *closed service path* in SG as a sequence of services $s_1, s_2, \ldots, s_n$ (each service $\in S$) such that $(s_n, s_n + 1) \in C$ is a directed connector between services for $i = 1, 2, \ldots, n$ and $s_1 = s_n$. A *service cycle* is a closed service path in which no service node is repeated except the first and last, and which contains at least two distinct service nodes. Let $ServiceCycles()$ return the set of all service cycles in a service graph. CDD returns 1 (True) if there is at least one cyclic dependency in the model:

$$CDD = \begin{cases} 1: & \text{if } |ServiceCycles(SG)| = 0 \\ 0: & otherwise \end{cases}$$

6.4 Metrics Calculation Results

We note that for the *Inter-Service Coupling through Shared Services* decision as well as *SDBI* metric, our metrics scale is reversed in comparison to the other two decisions, 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.

The metrics results for each model per decision metric are presented in Table 3.

7 Ordinal Regression Analysis Results

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 and summarized in Table 3 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 objective of the analysis is to predict the likelihood of the dependent outcome variable for each of the decisions by using the relevant metrics for each decision.

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.

The statistical significance of each regression model is assessed by the p-value; the smaller the p-value, the stronger the model. A p-value smaller than 0.05 is

Table 3. Metrics calculation results

Metrics	BM1	BM2	BM3	CO1	CO2	CO3	CI1	CI2	CI3	CI4	EC1	EC2	EC3	
Database-based inter-service coupling														
DTU	0.33	1.00	1.00	1.00	1.00	0.60	1.00	1.00	1.00	1.00	1.00	1.00	0.00	
SDBI	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	
Inter-service coupling through synchronous invocations														
SIC	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	
ACU	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	
Inter-service coupling through shared services														
DSS	0.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	0.38	0.00	0.00	0.00	0.00	
TSS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
CDD	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	
Metrics	ES1	ES2	ES3	FM1	FM2	FM3	HM1	HM2	RM1	RM2	RM3	RS	TH1	TH2
Database-based inter-service coupling														
DTU	1.00	0.00	0.33	1.00	1.00	1.00	1.00	1.00	0.00	1.00	1.00	0.66	1.00	1.00
SDBI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Inter-service coupling through synchronous invocations														
SIC	0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.80	1.00	0.00	0.00	0.11	0.00	0.60
ACU	0.00	0.00	0.00	0.00	1.00	0.08	0.50	0.20	0.00	0.00	0.00	0.11	0.00	0.00
Inter-service coupling through shared services														
DSS	0.27	0.34	0.34	0.62	0.47	0.55	0.52	0.00	0.00	0.00	0.00	0.36	0.33	0.00
TSS	0.00	0.00	0.00	0.00	0.00	0.18	0.00	0.00	0.00	0.18	0.16	0.00	0.00	0.00
CDD	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00

generally considered statistically significant. 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.

Table 4. Regression analysis results

Intercepts/coefficients	Value	Model p-value
Database-based inter-service coupling		
Intercept (≥ Badly Supported)	2.6572	1.706019e−06
Intercept (≥ Neutral)	0.8789	
Intercept (≥ Well Supported)	−1.3820	
Intercept (≥ Very Well Supported)	−3.1260	
Metric Coefficient (DTU)	6.4406	
Metric Coefficient (SDBI)	−3.7048	

(*continued*)

Table 4. (*continued*)

Inter-service coupling through synchronous invocations		
Intercept (≥Badly Supported)	−2.6973	6.705525e−11
Intercept (≥Neutral)	−4.4087	
Intercept (≥Well Supported)	−5.8513	
Intercept (≥Very Well Supported)	−15.3677	
Metric Coefficient (SIC)	17.3520	
Metric Coefficient (ACU)	6.5520	
Inter-service coupling through shared services		
Intercept (≥Neutral)	59.4089	1.625730e−10
Intercept (≥Very Well Supported)	9.7177	
Metric Coefficient (DSS)	−82.4474	
Metric Coefficient (TSS)	−122.2583	
Metric Coefficient (CDD)	−57.4650	

8 Discussion

In this section, we first discuss what we have learned in our study that helps to answer the research questions and then discuss potential threats to validity.

8.1 Discussion of Research Questions

To answer **RQ1** and **RQ2**, we proposed a set of generic, technology-independent metrics for each coupling-related decision, and to each decision option corresponds at least one metric. We objectively assessed for each model how well patterns and/or practices are supported for establishing the ground truth, and extrapolated this to how well the broader decision is supported. We formulated metrics to numerically 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 our objectively evaluated assessment with high accuracy. 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.

Here, we make the assumption that the source code of a system can be mapped to the models used in our work. To enable this, we used rather simplistic modeling means, which can rather easily be mapped from a specific source code to the system models. However, it should be noted that full automation of this mapping is an additional effort that needs to be considered and is the subject of ongoing work on our part.

Regarding **RQ3**, we consider that existing modeling practices can be easily mapped to our microservice meta-model and there is no need for major extensions. More specifically, for completing the modeling of our evaluation system 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 decisions *Inter-Service Coupling through Databases* and *Inter-Service Coupling through Shared Services* require to model at least the *Service* and the *Database* component types and the technology-related connector types (e.g. *Database Connector*, *RESTful HTTP* and *Asynchronous Connector*) and the read/write process which explicitly modeled in the *Database Connector* type. The *Inter-Service Coupling through Synchronous Invocations* decision requires a number of additional components (e.g. *Event Sourcing, Stream Processing, Messaging, PubSub*) and the respective connectors (e.g. *Publisher, Subscriber, Message Consumer*, *Messages Producer*, *RESTful HTTP* and *Asynchronous Connector*) 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 (including substantial industry experiences), 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 carefully made sure only to change specific aspects in a variant and keep all other aspects stable. That is, while the variants do not represent actual systems, they are reasonable evolutions of the original designs.

The modeling process is also considered as source of internal validity threat. The models of the systems were repeatedly and independently cross-checked by the author team that has considerable experience in similar methods, but the possibility of some interpretative bias remains: other researchers might have coded or modeled differently, leading to different models. As a mitigation, we also offer the whole models and the code as open access artifacts for review. Since

we aimed 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

Our approach considered that it is achievable to develop a method for automatically assessing coupling related tenets in microservice decisions based on a microservice system's component model. We have shown that this is possible for microservice decision models that contain patterns and practices as decision options. In this work, we first modeled the key aspects of the decision options using a minimal set of component model elements. These could be possibly 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. The statistical analysis shows that each decision related metrics are quite close to the manual, pattern-based assessment.

There are many studies related on metrics for component model and other architectures so far, but specifically for microservice architectures and their coupling related tenets have not been studied. Based on our discussion in Sect. 2, assessing microservice architectures using general metrics it is not very helpful. Our approach is one of the first that studies a metrics-based assessment of coupling-related tenets in the microservices domain. We aim to 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 improvement, stability, or deterioration in microservice architecture conformance. With small changes, our approach could also be applied, for instance, 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. Allen, E.B., Gottipati, S., Govindarajan, R.: Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach. Softw. Qual. J. (2), 179–212. https://doi.org/10.1007/s11219-006-9010-3
2. Allen, E.B., Gottipati, S., Govindarajan, R.: Measuring size, complexity, and coupling of hypergraph abstractions of software: an information-theory approach. Softw. Qual. J. **15**, 179–212 (2006)
3. Bansiya, J., Davis, C.G.: A hierarchical model for object-oriented design quality assessment. IEEE Trans. Softw. Eng. **28**(1), 4–17 (2002)
4. Basili, V.R., Briand, L.C., Melo, W.L.: A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. **22**(10), 751–761 (1996)
5. Bogner, J., Wagner, S., Zimmermann, A.: Towards a practical maintainability quality model for service-and microservice-based systems, pp. 195–198, September 2017
6. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)
7. Corbin, J., Strauss, A.L.: Grounded theory research: procedures, canons, and evaluative criteria. Qual. Sociol. **13**, 3–20 (1990)
8. Frank, E., Harrell, J.: Regression Modeling Strategies: With Applications to Linear Models, Logistic and Ordinal Regression, and Survival Analysis, 2nd edn. Springer, Heidelberg (2013)
9. Garousi, V., Felderer, M., Mäntylä, M.V.: Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering. CoRR
10. Goldstein, M., Moshkovich, D.: Improving software through automatic untangling of cyclic dependencies. In: Association for Computing Machinery, New York, NY, USA (2014)
11. Harrison, R., Counsell, S.J., Nithi, R.V.: An evaluation of the mood set of object-oriented software metrics. IEEE Trans. Softw. Eng. **24**(6), 491–496 (1998)
12. 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
13. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Boston (2003)
14. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term, March 2004. http://martinfowler.com/articles/microservices.html
15. Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, Sebastapol (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. 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)
18. Richardson, C.: A pattern language for microservices (2017). http://microservices.io/patterns/index.html
19. Skowronski, J.: Best practices for event-driven microservice architecture (2019). https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk

20. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018)
21. 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
22. Zimmermann, O.: Microservices tenets. Comput. Sci. - Res. Dev., 301–310 (2016). https://doi.org/10.1007/s00450-016-0337-0
23. Zimmermann, O., Stocker, M., Zdun, U., Luebke, D., Pautasso, C.: Microservice API patterns (2019). https://microservice-api-patterns.org