Evaluating and Improving Microservice Architecture Conformance to Architectural Design Decisions

Evangelos Ntentos[1](✉), Uwe Zdun[1], Konstantinos Plakidas[1], 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.geiger@siemens.com

Abstract. Microservices are a commonly used architectural style targeting independent development, deployment, and release of services, as well as supporting polyglot capabilities and rapid release strategies. This depends on the presence of certain software architecture qualities. A number of architecture patterns and best practices that support the required qualities have been proposed in the literature, but usually in isolation of one another. Additionally, in real-world systems, assessing conformance to these patterns and practices and detecting possible violations is a significant challenge. For small-scale systems of a few services, a manual assessment and violation detection by an expert is probably both accurate and sufficient. However, for industrial-scale systems of several hundred or more services, manual assessment and violation detection is laborious and likely leads to inaccurate results. Furthermore, manual assessment is impractical for rapidly evolving and frequently released system architectures. In this work we examine a subset of microservice-relevant patterns, and propose a method for the semi-automatic detection and resolution of conformance violations. Our aim is to assist the software architect by providing a set of possible fix options and generating models of "fixed" architectures.

1 Introduction

Microservices are one of many service-based architecture decomposition approaches (see e.g. [1–4]). The chief features of microservices are that they communicate via message-based remote APIs in a loosely coupled fashion, and that they can be highly polyglot; ideally, microservices should not share their data with other services. This allows the rapid evolution of individual microservices independently of one another, and their independent deployment in lightweight containers or other virtualized environments. These features make microservices ideal for DevOps practices (see e.g. [5,6]).

While a large body of literature has examined architectural patterns and recommended "best practices" in a microservice context [3,7,8], translating these theoretical insights into usable tools to assist the architectural evolution of actual microservice-based systems has lagged behind. While the theoretical tenets proposed in the literature are easy to grasp and maintain in small-scale systems, ensuring conformance in large,

© Springer Nature Switzerland AG 2021
H. Hacid et al. (Eds.): ICSOC 2021, LNCS 13121, pp. 188–203, 2021.
https://doi.org/10.1007/978-3-030-91431-8_12

complex, as well as rapidly and independently evolving systems quickly becomes a laborious affair requiring considerable manual work and resulting in extensive overhead effort. Furthermore, patterns have mutual dependencies, meaning that improvement in one area can result in deterioration in another. Real-world architectures are also impacted by a number of non-microservice-specific requirements, which also can lead to unintended violations of microservice best practices.

This work provides a set of actionable solutions to violations on different aspects of microservice architectures, as part of a larger study on the topic. Three architectural design decisions (ADDs) were selected as representing very different aspects of architecting microservices, so as to demonstrate the wide applicability of our approach. Other ADDs have already been covered in our prior work. More specifically, for covering the best practices of client-system communication we chose the External API decision; for the guaranteed delivery of messages, a critical aspect of many business-critical microservice systems, we used the Inter-Service Message Persistence decision to examine the relevant recommended practices; finally, to cover the logging and monitoring practices that ensure observability of the microservices and their complex interactions, we used the End-to-End Tracing decision. In this context, we aim to study the following research questions:

- **RQ1.** What are the possible architecture violations related to the above-mentioned ADDs and how can they be automatically detected?
- **RQ2.** What are the possible fixes for the violations found in RQ1 and how can architects be assisted in choosing the appropriate solutions and applying them?

We propose a novel architecture refactoring approach that uses empirically validated metrics proposed in our prior work [9] to evaluate the degree of architecture conformance for each of the given ADDs. For every ADD design option, we define every possible violation and propose a corresponding, automated violation detection algorithm, as well as a set of possible fixes. For each microservice-based system, the sets of ADD options, violations, and fixes leads to a search tree of possible architecture designs that partly or entirely enforce conformance to best practices, which we can continually assess using our metrics.

To evaluate our approach we utilized a set of 24 models of microservice-based systems from third-party practitioners (see Table 1). For each of these, we implemented the automated violation detection and refactoring (fix) algorithms to detect the possible violations and to generate all the possible fixes for addressing each violation, resulting in a set of models. Using our metrics, we evaluated the improvements compared with the original version, as well as any outstanding issues. This process was iteratively repeated until all violations were resolved. Each of the violations found in the 24 models can be fully resolved leading to optimal metric values within *at most* 3 refactoring steps, usually with many suggested optimal models provided as options for architects to choose from.

This paper is structured as follows: In Sect. 2 we analyze the ADDs examined in this work, the associated patterns and practices, and the corresponding metrics. Section 3 discusses and compares our approach to existing studies in the literature. Our research methods and the tools we have applied in our study are described in Sect. 4, followed by a detailed explanation of our approach in Sect. 5. The evaluation process is given at Sect. 6, the results are discussed in Sect. 7, and the threats to validity in Sect. 8. Finally, in Sect. 9 we draw conclusions and discuss future work.

2 Background: Decisions and Metrics

In this section, we briefly introduce the three ADDs and the corresponding patterns and practices as decision options, based on our prior work. The decisions have been modeled based on an empirical study of existing best practices and patterns by practitioners [10], while the metrics used to assess the pattern conformance of each given system derive from [9].

External API Decision. A fundamental decision in microservice-based systems is how external clients are connected to the system services. This can affect aspects related to loose coupling, releasability, independent development and deployment, and continuous delivery. The simplest method, but with the highest negative impact, occurs when the *clients can call into system services directly*, resulting in high coupling that impedes releasing, developing, and deploying the clients and system services independently of each other. Another option, that solves possible problems caused by client-service direct connections, is the *API Gateway* [3], which provides a common entry point for the system (Facade component) and all client requests are routed via this component. 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, etc. (see [7]). The *Backends for Frontends* pattern [3] is another variant of *API Gateway* that specializes in handling different types of clients (e.g., mobile and desktop clients). Alternatively, the *API Composition* pattern [3] describes a service that shields other services from the clients by actively gathering and composing their data. In our previous work [9], we have empirically defined two metrics that can be used to assess conformance to each of the decision options:

- *Client-side Communication via Facade utilization metric* measures 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*) compared to the total number of unique client links.
- *API Composition utilization metric* measures the proportion of clients connected services which are possibly composing an *External API* using *API Composition*.

Inter-service Message Persistence Decision. The persistence or missing persistence of the inter-service messages is another decision with considerable impact on the qualities of the system. Many real-world systems use *no inter-service message persistence*, while options that support message persistence are the *Messaging* pattern [11], in which persistent message queuing is used to store a producer's messages until the consumer receives them, or alternatively *Stream Processing* [8] components (e.g. Apache Kafka). Another option is *Interaction through a Shared Database*, since it supports some level of message persistence, but not the automated support of *Messaging*. A technique that is more microservice-relevant and able to support a lower level of persistence to *Messaging* or a *Shared Database* is the combination of the *Outbox* and the *Transaction Log Tailing* patterns [3]. A persistence more tailored to event-driven or eventually consistent microservice architectures can be achieved following the *Event Sourcing* pattern [3].

For this decision, too, we have empirically defined three metrics that can be used to assess conformance to each of the decision options:

- *Service Messaging Persistence utilization metric* measures the proportion of all service interconnections that are made persistent through a supporting technology (i.e. *Messaging* or *Stream Processing*).
- *Shared Database utilization metric* measures the proportion of all interconnections via a *Shared Database*.
- *Outbox/Event Sourcing utilization metric* measures the proportion of all interconnections with *Outbox/Event Sourcing*.

End-to-End Tracing Decision. End-to-end tracing is an important aspect in microservice architectures since they are usually highly distributed and polyglot systems with complex interactions. One option, like in the other decisions, is to offer *no tracing support*. Alternatively, traces can be recorded on either the services themselves or facade components (or both) via *Distributed Tracing* [3]. A less comprehensive level of tracing can be achieved when service communication is routed through a central component, which stores some, but not all inter-service communication (e.g., *Publish/Subscribe*, *Message Broker* [11], *API Gateway* or *Event Logging* [3,8]); the exception is *Event Sourcing*, which temporarily stores *all* service events.

For this decision, too, we have empirically defined three metrics that can be used to assess conformance to each of the decision options:

- *Services and Facades Support Distributed Tracing metric* measures the proportion of all services and facades that support distributed tracing.
- *Service Interaction via Central Component utilization w/o Event Sourcing metric* measures the proportion of all service interactions through a central component other than *Event Sourcing*.
- *Service Interaction via Central Component with Event Sourcing metric* measures the proportion of all service interactions through a central component via *Event Sourcing*.

3 Related Work

The fundamentals of the term "microservices" were first discussed by Fowler and Lewis [12], and fundamental tenets by Zimmermann [5]. Richardson [3] has published a collection of microservice patterns and practices, while a mapping study by Pahl and Jamshidi [1] has summarized much of the previous literature on patterns. Skowronski [8] has examined event-driven microservice architectures specifically, and microservice API patterns were studied by Zimmermann et al. [7].

A number of studies have focus on techniques for detecting design or architecture "bad smells" (violations). Taibi and Lenarduzzi [13] defined a list of microservice-specific smells, while Neri et al. [14] have presented an extensive examination of architectural smells for independent deployability, horizontal scalability, fault isolation, and decentralisation of microservices, as well as suggesting refactorings to resolve them. Most similar studies are more generic, but still useful. Le et al. [15] proposed a classification of architectural smells and their impact on different quality attributes. Catalogs

of smells have been published by Garcia et al. [16,17] and Azadi et al. [18]. Detection strategies for smell categories related to our study are discussed by Brogi et al. [19], Le et al. [20], Marinescu [21], and especially Neri et al. [14], along with suggested refactorings for resolving them. Although these works study various aspects of architecture violations detection, and some investigate aspects related to the microservice domain, none covers detecting and addressing violations specifically associated with the ADDs covered in this work (external API, persistent messaging, and end-to-end tracing) in a microservice context, which our work investigates in detail.

As a result, we expect that our work produces more accurate detection of decision-specific violations and more targeted suggestions for fixes. On the other hand, our approach requires a model in which the component and connector roles in a microservice architecture have been modeled (as for instance done with stereotypes in the model introduced in Fig. 2). That is, our work requires additional insight into a system's architecture, and some effort in encoding the corresponding models; however, this knowledge is at a relatively high level of abstraction and the resulting models are not impacted by changes in service implementation. We are currently working on a semi-automatic approach for architecture reconstruction and modelling that relies on reusable code abstractions and is thus suitable for complex systems with short delivery cycles.

4 Research and Modeling Methods

In this section, we summarize the main research methods applied in our study. These have been more extensively described in our previous work [22]. For reproducibility, all the code of the algorithms' implementation and the models produced in this study will be made available online, as an open-access dataset in a long-term archive.[1]

4.1 Research Method

Figure 1 shows the structure of the research process of this study. In Sect. 2 we have already explained in detail the architectural decisions and the model-based metrics on which this study is based. In Sect. 5 we present precise definitions and algorithms a) for the detection of possible violations per decision option, and b) for the possible fixes (architecture refactorings) for each violation.

We have tested our approach by applying the algorithms to the 24 models in our data set. First all violations present in each model were detected, and then all possible fixes for each violation were applied in an iterative-exhaustive manner, i.e., on the resulting, refactored models for each violation fix, we again performed *all* violation detection algorithms and applied *all* possible refactorings, until either no more violations were detected, or we arrived at a refactored model identical to a previous version. In the latter case, which we did not encounter here, this would have meant that a violation could not be entirely resolved, as its fix introduced other violations. For each of the final models (the 'leaves' of the iteration tree), we assessed pattern conformance through our metrics on microservice coupling, to judge the improvement compared to the original model.

[1] https://doi.org/10.5281/zenodo.5549978.

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
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

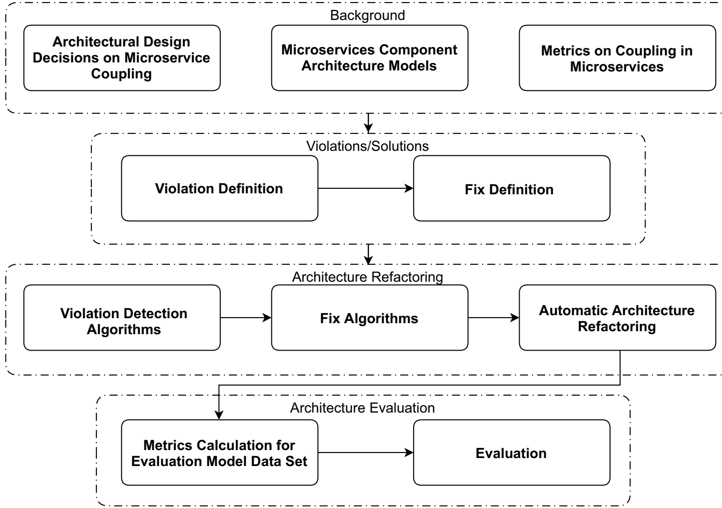

Fig. 1. Overview diagram of the research method followed in this study

5 Architecture Refactoring Approach

From an abstract point of view, a microservice-based system is composed of components and connectors, with distinct sets of component types and connector types. This applies also to indirect or implicit relationships between components, such as indirect dependencies, which can be described as a special set of connectors. For example, in Fig. 2, two components are indirectly linked via the API gateway.

We base our definitions of the violations and fixes on the notion of an architecture model consisting of a directed components and connectors graph. This can be expressed formally as: A microservice architecture model M is a tuple (CP, CN, CPT, CNT, ST) where:

- CP is a finite set of **component nodes**. The operation $components(M)$ returns all components in M.
- $CN \subseteq CP \times CP$ is an ordered finite set of **connectors**. $connectors(M)$ returns all connectors in M.
- CPT is a set of **component types**. The operation $services(M)$ returns all components of type $service$ in M. The operation $service_connectors(M)$ returns all connectors of components of type $service$ in M.
- CNT is a set of **connector types**.
- ST is a finite set of **stereotype nodes**. The operation $cp_stereotypes(CP)$ returns all stereotypes of component CP. The operation $cn_stereotypes(CN)$ returns all stereotypes of connector CN. Stereotypes can be applied to components to denote their type, such as *Service*, *API Gateway*, etc. Stereotypes can be applied to connectors to denote their type, such as *Read_Data*, *RESTful HTTP*, or *Asynchronous*. Some are specialized with tagged values (details omitted here for space reasons).

- $cp_annotations : CP \rightarrow \{String\}$ is a function that maps an component to its set of annotations. Annotations are used in our approach (in some of the fixes) to document aspects that need further consideration or maybe manual refactoring.
- $cn_annotations : CN \rightarrow \{String\}$ is a function that maps a connector to its set of annotations.

Please note that we define many additional model traversal operations not detailed here for space reasons.

5.1 Violations and Detection Algorithms

Table 2. Identified violations and violation detection algorithms

Violation	Violation detection algorithm summary
D1: External API	
D1.V1: Services are directly connected to clients	All services in the model are traversed, and it is checked whether services are directly connected to clients or web UIs. If this is the case, a violation is raised. Each service-client connector that is found is returned by the detector operation
D2: Persistent Messaging for Inter-Service Communication	
D2.V1: Services communicate without using an intermediary component that is able to persist the communication (e.g., Message Brokers or a persistent Publish/Subscribe or Stream Processing or Event Sourcing or Outbox/Transaction Log Tailing or Database) and no persistent messaging occurs between them	All service connectors in the model are traversed. If no intermediary component is found, the violation is raised and the list of all relevant connectors is returned by the detector operation
D3: End-to-End Tracing	
D3.V1: Distributed Tracing is not supported on services and/or facades or services communicate without using a central intermediary component (e.g., Message Brokers or persistent Publish/Subscribe or Stream Processing or Event Sourcing or Outbox/Transaction Log Tailing or API Gateway)	All services, facades and the corresponding connectors in the model are traversed, and it is checked whether services and/or facades support tracing or whether an intermediary component is presented. If no intermediary component or tracing support on services/facades is found, the violation is raised and the list of all relevant connectors is returned by the detector operation

Table 2 summarizes the possible violations we have identified for each of the decisions. The table also describes in detail how the algorithms that we use for detecting the violations in the models work. As a detailed example, Algorithm 1 detects the *Services communicate without using an intermediary component* violation of Decision D2. It returns a list of connected service pairs s_i and s_j, that are *not* connected via an intermediary component.

Algorithm 1: Services Communicate w/o Intermediary Component Violation

```
input: Model M
output: Set<Tuple> Component intermediary
begin
  violations ← ∅
  for s_i ∈ services(M):
  for s_j ∈ services(M):
    if (s_i, s_j) ∈ direct_service_connectors(M):
      violations ← violations ∪ (s_i, s_j)
  return violations
end
```

5.2 Fix Options and Algorithms

Table 3 details all the fixes for each identified violation, along with a summary of the fix algorithm. Please note that many algorithms can only be applied fully automatically with their default values. Many of them require human review and decision by the architect. For example, the architects can be presented with a choice of an intermediary component to use to replace services links.

Table 3. Identified fixes and fix algorithms

Violation	Fix	Fix and fix algorithm summary
D1: External API		
D1.V1	*D1.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical
	D1.V1.F2: Introduce a new API Gateway and connect client to services via it	Disconnect client(s) from the services and introduce a new API Gateway. Connect the client(s) to the API Gateway and the API to each former client-connected service
	D1.V1.F3: Introduce API Composition service or service with reverse proxy capabilities and connect client(s) to the services via this component	Disconnect client(s) from the services and introduce a new API composition service. Connect the client(s) to the API composition service and the latter to each former client-connected service
D2: Persistent Messaging for Inter-Service Communication		
D2.V1	*D2.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical
	D2.V1.F2: Remove the non-persistent connectors between services and replace them with persistent messaging-based connectors	Replace non-persistent interconnections with interactions via an intermediary component (e.g., API Gateway, Pub/Sub, Message Broker). The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace non-persistent interconnections with persistent interconnections via this component
	D2.V1.F3: Remove the non-persistent connectors between services and replace them by writing to and reading from a common database	The architect has to select if an existing database can be used for the fix, or a new one has to be created For each connector, introduce communication by writing to and reading from this database. Delete the non-persistent interconnections
D3: End-to-End Tracing		
D3.V1	*D3.V1.F1: Do not fix the violation*	The architect should have the option to *not* fix the violation, e.g. because it is not critical
	D3.V1.F2: Remove the connectors that don't support end-to-end tracing between services and replace them with interactions via an intermediary component (e.g., API Gateway, Pub/Sub, Message Broker)	The architect has to select if an existing intermediary component can be used for the fix, or a new one has to be created. Replace interconnections that don't support end-to-end tracing with interconnections via this component
	D3.V1.F3: Connect services and facades that don't support end-to-end tracing with a tracing component (e.g., Zipkin)	The architect has to select if an existing tracing component can be used for the fix, or a new one has to be created. Introduce interconnections from service and facades to tracing component

The Algorithms 2 and 3 respectively present the fixes F2 and F3, for Decision D2 and its Violation V1. For explanations of each fix, please study Table 3.

Algorithm 2: Remove the non-persistent connectors between services and replace them with persistent messaging-based connectors

```
input: Model M, Set<Tuple> violation , Component intermediary_component
output: -
begin
  for (s_i, s_j) ∈ violation:
    add_connector(s_i , intermediary_component ,
      get_applicable_stereotypes(M, (s_i , s_j)))
    add_connector(intermediary , s_m ,
      get_applicable_stereotypes(M, (s_i , s_j)))
  delete_direct_connector(M, (s_i, s_j))
end
```

Algorithm 3: Remove the non-persistent connectors between services and replace them by writing to and reading from a common database

```
input: Model M, Set<Tuple> violation , Component database
output: -
begin
  for (s_i, s_j) ∈ violation:
    add_connector(s_i , database ,
      get_applicable_stereotypes(M, (s_i , s_j)))
    add_connector(s_j , database ,
      get_applicable_stereotypes(M, (s_i , s_j)))
  delete_direct_connector(M, (s_i, s_j))
end
```

5.3 Example Application

In Fig. 2 the model CI4 from Table 1 is shown as an illustrative example to demonstrate all three violations and possible fixes. In this model the *Cinema Catalog* service is connected directly with *Movie* and *Booking* services, causing D2.V1 and D3.V1, while *Client* is connected directly with *Cinema Catalog* service, causing D1.V1. In contrast, *Booking Payment* and *Notification* services are connected to each other and with the *Client* through the *API Gateway*, resulting in no violation. If we run our fix algorithms, some of the resulting refactoring suggestions are:

- *Applying Fix D1.V1.F2:* The architect can choose the existing *API Gateway* and connect *Client* to *Cinema Catalog* and *Movie* services through it. The current connectors are removed by this fix.
- *Applying Fix D1.V1.F3:* The architect can introduce an *API composition* service or service with reverse proxy capabilities and connect *Client* to *Cinema Catalog* and *Movie* services through it. The current connectors are removed by this fix.
- *Applying Fix D2.V1.F2:* All services with non-persistent connectors are disconnected and connected to a *Message-based persistent mechanism* (all interactions will be happening via this component). For example, this fix can introduce a new *Pub-/Sub intermediary component* (alternatively *Message Broker* or *API Gateway*), to which all involved services will be connected with *publish* and *subscribe* operations supporting persistent communications.
- *Applying Fix D2.V1.F3:* All services with non-persistent connectors are disconnected from each other as well as from their existing databases and connected to a new *shared database* with *read* and *write* operations.

– *Applying Fix D3.V1.F2: Cinema Catalog*, *Movie* and *Booking* services that don't support end-to-end tracing will be disconnected from each other and connected to a new (or existing) *intermediary component* (e.g., *Pub/Sub*, *Message Broker* or *API Gateway*).
– *Applying Fix D3.V1.F3:* A new *tracing component* (e.g., *Zipkin*) is introduced and connected to all services and the *API Gateway*.

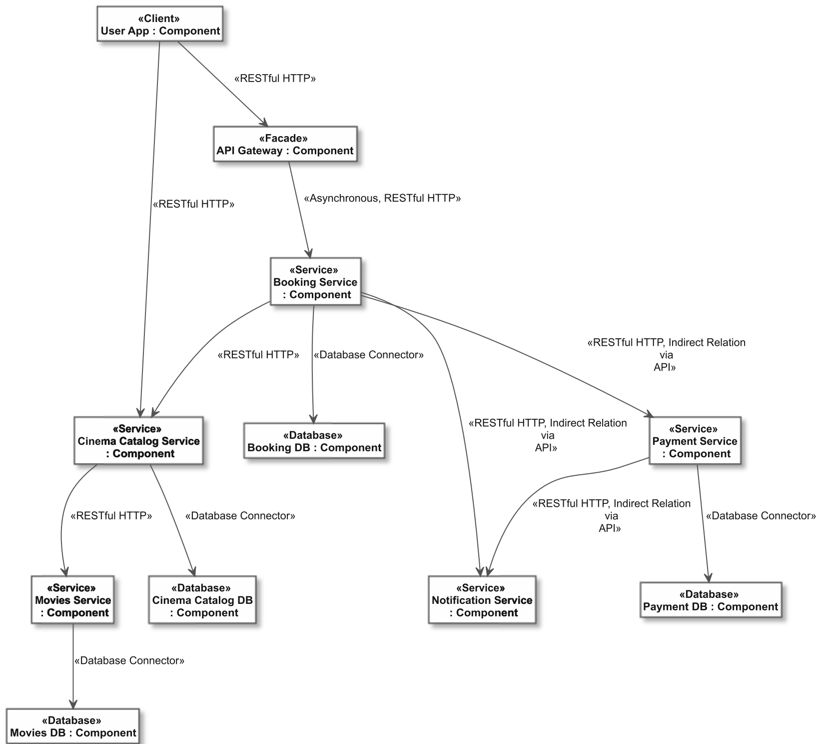

Fig. 2. Example of an architecture component model (CI4 in Table 1): this architecture violates all three ADDs

6 Iterative Application and Evaluation

To evaluate our work, we have fully implemented our algorithms for detecting violations and performing fixes, as well as generating the set of metrics described in Sect. 2 to measure the improvements and the presence of remaining violations, in our model set. In case multiple violations are present in a model, then the algorithms can be employed iteratively, until all violations have been fully resolved.

As an example, let us illustrate this exhaustive iterative refactoring for the previously mentioned CI4 Model (see Fig. 2). CI4 violates all the three decisions as indicated

by the corresponding decision-related measures in Table 4. The incremental refactoring process is illustrated in Fig. 3. At the first iteration, there are three branches, indicating the respective violations. The first refactoring step produces 6 possible model variants, one for each fix option from Table 3. All resulting models have resolved the respective violation, but have the other two unresolved, requiring another refactoring step that produces 18 new model variants. In turn, 7 of the resulting models still violate D1.V1 and D2.V1, requiring a third step to be resolved. At the end of the third step, we have 29 suggested model variants (M1_1, M2_1, M2_3, M1_2_1–M1_2_2, M2_1_1–M2_2_2, M2_4_1–M2_4_2, M3_1, M3_2_1–M3_2_2, M4_1, M4_2_1–M4_2_2, M4_3_1–M4_3_2, M5_1–M5_2, M4_4_1–M4_4_2, M6_1_1–M6_2_2, M6_2_1–M6_2_2, M6_3_1–M6_3_2, M6_4_1–M6_4_2) which all fully resolve the violations (i.e., scoring 1.00 in our assessment scale). The architect can choose the refactoring sequence, and from among those final optimal model variants, but can also choose to not apply certain fixes, e.g. due to other constraints that are outside of the scope of our study.

For evaluation purposes, we have performed this procedure for *all* 24 system models in Table 1. The resulting number of intermediary models and violation instances per step, and the number of final suggested models with an optimal assessment of 1.00, are given in Table 4, along with the initial violations and architecture assessment values for each model. Please note that the metrics reported here are the ones associated with each of the decisions in Sect. 2. Please also note that for each violation to be fixed, it

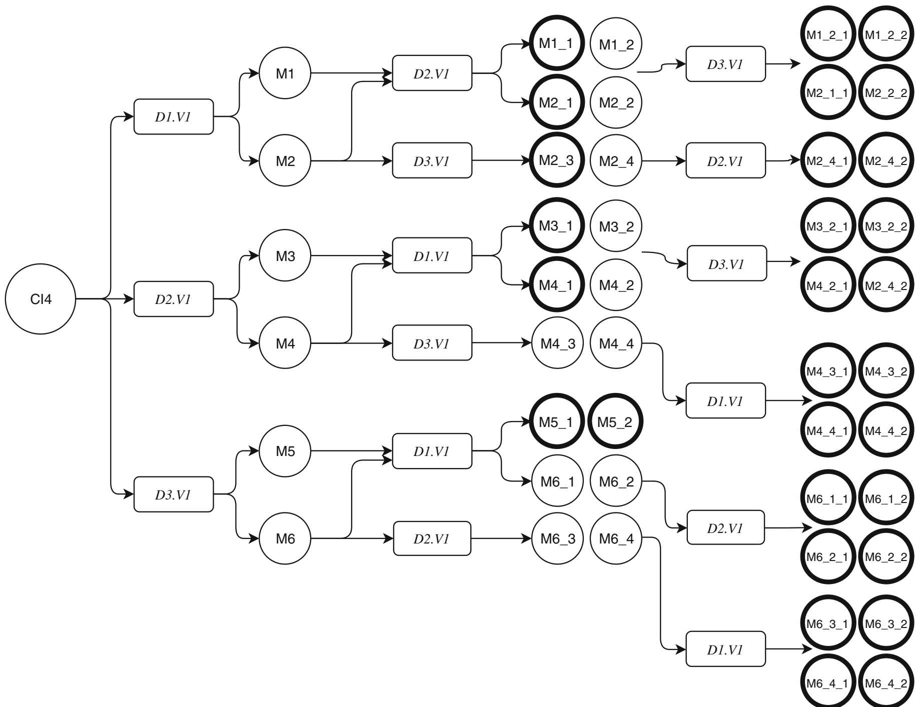

Fig. 3. Example of an exhaustive iterative application of our approach in the CI4 model. Final (i.e., fully resolved) resulting models are thickly outlined.

is enough that at least one of the corresponding metrics is optimal (1.00). Obviously, the number of steps required to reach optimal models depends on a) the number of the violations present in the initial model and b) on the possible appearance of new violations during the refactoring process, which did not occur in the present case. As can be seen in Table 4, *all* models are *fully resolved*—i.e., all assessment metrics are 1.00—after *at most* three steps.

Table 4. This table shows a) the architecture assessment (per decision/violation pair) of the original models used in our study, b) the number of models generated at each step of an iterative application of our algorithms, and c) the number of violation instances (generated models $\times$ violations per model) still remaining, or introduced, after each iteration, plus d) the resulting number of suggested (optimal) models at the end (cf. Fig. 3 for a detailed example).

Model ID	Initial model assessments			Models generated/remaining violation instances per refactoring step			Resulting suggested (optimal) Models
	D1.V1	*D2.V1*	*D3.V1*	*Step 1*	*Step 2*	*Step 3*	
BM1	1.00, 0.00	0.00, 0.00, 1.00	0.00, 0.00, 1.00	–	–	–	–
BM2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2/0	–	–	2
BM3	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2/0	–	–	2
CO1	0.00, 0.00	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6/9	18/11	22/0	29
CO2	1.00, 0.00	1.00, 0.00, 0.00	1.00, 1.00, 0.00	–	–	–	–
CO3	0.00, 0.00	0.00, 1.00, 0.00	1.00, 0.00, 0.00	2/0	–	–	2
CI1	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.14, 0.00	4/2	4/0	–	6
CI2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	2/0	–	–	2
CI3	0.00, 0.30	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6/9	18/11	22/0	29
CI4	0.50, 0.10	0.00, 0.00, 0.00	0.00, 0.60, 0.00	6/9	18/11	22/0	29
EC1	0.25, 0.00	0.00, 0.00, 0.00	0.00, 1.00, 0.00	4/4	8/0	–	8
EC2	0.25, 0.00	1.00, 0.00, 1.00	0.00, 0.00, 1.00	2/0	–	–	2
EC3	0.25, 0.00	0.00, 1.00, 0.00	0.00, 0.00, 0.00	4/2	4/0	–	4
ES1	1.00, 0.00	0.60, 0.00, 0.60	0.00, 0.60, 0.00	4/2	4/0	–	6
ES2	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.45, 0.00	4/2	4/0	–	6
ES3	1.00, 0.00	0.00, 0.00, 0.00	0.00, 0.45, 0.00	4/2	4/0	–	6
FM1	0.00, 0.25	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6/9	18/11	22/0	29
FM2	0.00, 0.50	0.00, 0.00, 0.00	1.00, 0.00, 0.00	4/4	8/0	–	8
HM1	0.00, 0.70	0.00, 0.00, 0.00	0.90, 0.00, 0.00	6/9	18/11	22/0	29
HM2	0.00, 0.70	0.80, 0.00, 0.80	0.90, 0.00, 0.80	6/9	18/11	22/0	29
RM	1.00, 0.00	1.00, 0.00, 0.00	0.14, 1.00, 0.00	–	–	–	–
RS	1.00, 0.00	0.11, 0.00, 0.00	0.62, 0.11, 0.00	4/2	4/0	–	6
TH1	0.25, 0.12	0.00, 0.00, 0.00	0.00, 0.00, 0.00	6/9	18/11	22/0	29
TH2	0.25, 0.04	0.66, 0.00, 0.66	0.00, 0.00, 0.66	6/9	18/11	22/0	29

7 Discussion

To answer **RQ1** we have systematically specified a number of decision-based violations related to each possible decision option, summarized in Table 2. As we have empirically shown in our prior work [9] that the metrics described in Sect. 2 can reliably distinguish favored or less favored design options, the role of the violation detectors is to find the precise locations in the models where the violations occur. For each system model in our

evaluation dataset it was possible to suggest fixes that bring the architecture to optimal values, meaning that the algorithms have found the right place(s) to apply the fixes.

Regarding **RQ2** we defined a number of algorithms addressing every possible violation, with multiple fix options (cf. Table 3). If all options are tried out, this results in a search tree of possible architecture models, which can in turn be assessed, using our metrics, to measure improvements to the initial architecture and detect any remaining violations. We have shown (cf. Table 4) that an iterative approach results, within a few steps, in a sufficient variety of possible architecture models that remove all detected violations and ensure pattern conformance of the system architecture. The multiple optimal model variants that result from our approach give architects substantial levels of freedom in their design decisions. As detection is fully automated and human expertise is limited to the fix process, the approach is well suited to be run in a continuous delivery environment, which was one of our research goals.

8 Threats to Validity

The basis material of our study derives from third-party sources: the solutions we propose are gathered from the best practices recommended in the published literature, and our evaluation dataset is a fairly representative set of systems (cf. Table 1), derived from nine different sources and published with the express purpose of demonstrating microservice architecture features. One possible threat to the internal validity of our algorithms is that they depend on the particular modelling approach we have adopted. However, our approach is by design abstract and generic, based on typical component-and-connector models used widely in the literature. The author team, with considerable experience in modeling methods, performed the system modeling as well as, repeatedly and independently cross-checked all models. As the main modelling criterion was the ability to adequately represent the context of our systems, we cannot exclude that other teams might arrive at different interpretations, but we are confident that any resulting models would be broadly similar and compatible with our results. Furthermore, the algorithms we specified could easily be adapted to a different model, as they operate on the level of basic architectural constructs.

Nevertheless, some limitations remain. In order to remove the obstacles provided by the polyglot nature of microservice-based systems, we have chosen to apply our metrics and tools at a relatively high level of abstraction. We also limited our evaluation in the present paper to the patterns, metrics, and concerns applying to the given three ADDs, which in a real-world architecture would be insufficient. This point is addressed in previously published and ongoing parts of our work, which extend the coverage to additional ADDs, and aim to extend and test our approach in a larger set of patterns, design requirements, and more granular parameters. The same concern applies as to the lack of evaluation of the applicability of our approach on larger and more complex systems that are commonly found in industry, but which were not accessible to us for study. The lack of full automation is also a major obstacle to practical application, as the process still requires considerable input by the architect. At the same time, our approach can not match the ability of an experienced architect, familiar with the system, to devise a much more optimal solution. This is a limitation of all generic architecture

assistance approaches, and one we intend to improve on. We want to emphasize that the present approach is a starting point from which the question of evaluating and improving microservice architectures can be examined, facilitating and building up to more complex and nuanced methods as more systems and decisions are modelled and tested. The generated models are also not optimal, as they are not evaluated, for example, on the coding/refactoring effort required to implement them. Nevertheless, the existence of a semi-automatic approach that detects and analyzes violations in an architecture remains of great value, since practitioners often ignore best practices, systems are often developed without a conscious effort to follow best practices, or are allowed to drift from the original architecture specifications over time.

9 Conclusion and Future Work

In this paper we present a set of violations for three microservice-related ADDs. Building on previous work, we have defined automatic detectors, which return the location where the violations occur, a set of possible fixes for each violation, and automatic algorithms for refactoring the system in order to fix the violations. We have evaluated our approach on a set of 24 models of various degrees of pattern violations and architecture complexity, and have shown that our approach is capable of resolving these violations in at most 3 refactoring steps. Both metric calculation and violation detection are fully automated, but the choice of fixes and refactoring sequence remains with the human architect. Thus the approach is still flexible enough to let the architect make meaningful architectural design choices.

In our future work, we aim to broaden the set of ADDs and violations included in our approach, enrich it with runtime metrics and other architecture aspects such as deployment environments, and extend our model dataset to include larger and more complex systems. In addition, we hope to experimentally validate our approach by employing it in real-world delivery pipelines as part of a feedback loop.

Acknowledgments. This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 864707; FWF (Austrian Science Fund) project API-ACE: I 4268; FWF (Austrian Science Fund) project IAC^2: I 4731-N. Our work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952647 (AssureMOSS project).

References

1. Pahl, C., Jamshidi, P.: Microservices: a systematic mapping study. In: 6th International Conference on Cloud Computing and Services Science, pp. 137–146 (2016)
2. 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)
3. Richardson, C.: A pattern language for microservices (2017). http://microservices.io/patterns/index.html
4. Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., Schuster, N.: Reusable architectural decision models for enterprise application development. In: Overhage, S., Szyperski, C.A., Reussner, R., Stafford, J.A. (eds.) QoSA 2007. LNCS, vol. 4880, pp. 15–32. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-77619-2_2

5. Zimmermann, O.: Microservices tenets. Comput. Sci. - Res. Dev. **32**(3), 301–310 (2016). https://doi.org/10.1007/s00450-016-0337-0
6. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 1: reality check and service design. IEEE Softw. **34**(1), 91–98 (2017)
7. Zimmermann, O., Stocker, M., Zdun, U., Luebke, D., Pautasso, C.: Microservice API patterns (2019). https://microservice-api-patterns.org
8. Skowronski, J.: Best practices for event-driven microservice architecture (2019). https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk
9. Ntentos, E., Zdun, U., Plakidas, K., Meixner, S., Geiger, S.: Metrics for assessing architecture conformance to microservice architecture patterns and practices. In: Kafeza, E., Benatallah, B., Martinelli, F., Hacid, H., Bouguettaya, A., Motahari, H. (eds.) ICSOC 2020. LNCS, vol. 12571, pp. 580–596. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-65310-1_42
10. Bures, T., Duchien, L., Inverardi, P. (eds.): ECSA 2019. LNCS, vol. 11681. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29983-5
11. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Boston (2003)
12. Lewis, J., Fowler, M.: Microservices: a definition of this new architectural term, March 2004. http://martinfowler.com/articles/microservices.html
13. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Softw. **35**(3), 56–62 (2018)
14. Neri, D., Soldani, J., Zimmermann, O., Brogi, A.: Design principles, architectural smells and refactorings for microservices: a multivocal review. SICS Software-Intensive Cyber-Physical Systems **35**(1), 3–15 (2019). https://doi.org/10.1007/s00450-019-00407-8
15. Le, D.M., Carrillo, C., Capilla, R., Medvidovic, N.: Relating architectural decay and sustainability of software systems. In: 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 178–181 (2016)
16. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Identifying architectural bad smells. In: 2009 13th European Conference on Software Maintenance and Reengineering, pp. 255–258 (2009)
17. Garcia, J., Popescu, D., Edwards, G., Medvidovic, N.: Toward a catalogue of architectural bad smells. In: Mirandola, R., Gorton, I., Hofmeister, C. (eds.) QoSA 2009. LNCS, vol. 5581, pp. 146–162. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02351-4_10
18. Azadi, U., Fontana, F., Taibi, D.: Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 88–97 (2019)
19. Brogi, A., Neri, D., Soldani, J., et al.: Freshening the air in microservices: resolving architectural smells via refactoring. In: Yangui, S. (ed.) ICSOC 2019. LNCS, vol. 12019, pp. 17–29. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45989-5_2
20. Le, D.M., Link, D., Shahbazian, A., Medvidovic, N.: An empirical study of architectural decay in open-source software. In: 2018 IEEE International Conference on Software Architecture (ICSA), pp. 176–17609 (2018)
21. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings, pp. 350–359 (2004)
22. Ntentos, E., Zdun, U., Plakidas, K., Geiger, S.: Semi-automatic feedback for improving architecture conformance to microservice patterns and practices. In: 18th IEEE International Conference on Software Architecture(ICSA 2021), March 2021. http://eprints.cs.univie.ac.at/6763/