Semi-automatic Feedback for Improving Architecture Conformance to Microservice Patterns and Practices

Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas
Faculty of Computer Science, Research Group Software Architecture
University of Vienna
Vienna, Austria
firstname.lastname@univie.ac.at

Sebastian Geiger
Siemens Corporate Technology
Vienna, Austria
sebastian.a.geiger@siemens.com

***Abstract*—Microservices are one of the most recommended architectural styles for distributed applications that support independent development and deployment, enable rapid release, and are highly scalable and polyglot. Many well-established patterns and best practices have been documented in the literature. As there are many such guidances, they have numerous interdependencies, and system designs must adhere to many other architecture constraints, too, implementations do not always conform to those guidances. In complex or large systems, it can be hard and tedious to spot violations. Our work aims to offer automated support for architecting during the continuous evolution of microservice-based systems. More specifically we aim to provide the foundations for an automated approach for architecture reconstruction, assessing conformance to patterns and practices specific for microservice architectures, and detect possible violations. Based on this, we provide actionable options to architects for improving architecture conformance as part of a continuous feedback loop. That is, our goal is to support architecting in the context of continuous delivery practices, where architecture violations are continuously analyzed and fix options are continuously suggested.**

***Index Terms*—Microservices, patterns, best practices, continuous architecture improvement, feedback loop**

I. Introduction

Microservices are one of many service-based architecture decomposition approaches (see e.g. [1], [2], [3], [4]). Microservices should not share their data with other services, can be highly polyglot, and communicate via message-based remote APIs in a loosely coupled fashion. This enables their independent deployment in lightweight containers or other virtualized environments, as well as the rapid evolution of individual microservices independently of one another. These features make microservices ideal for modern DevOps practices (see e.g. [5], [6]).

Many architectural patterns and other recommended "best practices" for microservices have been published in the literature [3], [7], [8]. So far little attention has been paid to providing usable means to enforce these practices during the evolution of microservice-based systems. This is problematic as it is hard to handle conformance manually, especially in large and/or complex microservice architectures. Best practices have dependencies among each other, meaning that improvement of one practice can lead to issues with another one. Many other system architecture and implementation requirements influence the architectures in ways that might lead to unwanted or accidental violations of microservice best practices, too. Finally, in the context of continuous delivery and DevOps practices, it is to be expected that the architecture will change rapidly and often without central coordination.

This study focuses on providing a set of actionable solutions to violations of loose coupling related microservice best practices. In particular, we investigate three major Architectural Design Decisions (ADD) related to microservice coupling: *Persistent Data Storage of Services* related to data sharing via shared data storage; *Service Interconnections* related to the effects of intermediate system components, such as API gateways, as well as of asynchronous integration; and *Dependencies through Shared Services* related to direct, transitive, and cyclic dependencies between individual microservices. These decisions have been modeled based on an empirical study of existing best practices and patterns used by practitioners from our prior work [9]

To address the outlined challenges, we propose a novel architecture refactoring approach that is specific for architectural design in the context of the microservice ADDs, and uses the empirically validated metrics proposed in our prior work [9]. These metrics enable us to study, for each of the above named ADDs, precisely how much a microservice architecture model conforms to favored or less favored design options. For each possible design option in the ADDs, we propose to systematically specify each possible violation. Based on those specifications, we propose automated violation detection algorithms. From the combination of possible ADD options, the chosen option, possible violations, and the detected violations, in each design situation we can calculate all possible next decision options by applying possible solutions to the violations. This leads to a search tree of possible next architecture design models, which we each assess using our metrics. With this we can compare architecture conformance of the current design and possible refactorings, and suggest

to an architect all possible improvements in the three ADDs. Please note that this approach is designed to be continuously applicable during each run of a continuous delivery pipeline. This paper aims to study the following research questions:

- **RQ1** What are the possible architecture violations related to the above-mentioned coupling-related architectural design decisions and how can they be automatically detected?
- **RQ2** How can architects be guided in fixing those violations in a continuous feedback loop, while retaining enough flexibility for architect's to chose between possible options, e.g. because other architecture trade-offs need to be considered?

To evaluate our approach we utilized a set of 27 models based on microservice-based systems originally created or described by practitioners (see Table I) as our main data set. We implemented the automated violation detection and refactoring algorithms to detect the possible violations and to generate all the possible fixes for addressing each violation. We then calculate our metrics on coupling in microservices to judge the improvements compared with the initial version. Our result is that in at most 4 refactoring steps, each of the violations found in the 27 models can be fully resolved leading to optimal metric values, usually with many suggested optimal models provided as options for architects to choose from.

This paper is structured as follows: In Section II we explain the decisions considered in this paper. We also explain the related patterns and practices, as well as the corresponding metrics as the background of our work. Section III discusses and compares to related work. Next, we describe the research methods and the tools we have applied in our study in Section IV. We then describe the approach details in Section V. In Section VI we explain the evaluation process of our work. Section VII discusses the RQs regarding the evaluation results. In Section VIII, we then analyse the threats to validity. Finally, in Section IX we draw conclusions and discuss future work.

II. Background

In this section, we briefly introduce the three coupling-related decisions, their decision options (i.e. the possible patterns and practices that can be chosen). This information comes from two of our prior works: a) The decisions have been modeled based on an empirical study of existing best practices and patterns by practitioners [9], [10]. This study also contained a detailed analysis of possible decision drivers (forces, quality attributes) of the decision options. b) We have defined and empirically validated metrics to assess, for a given system model, how well it conforms to the patterns and best practices modeled in our decision model. Based on the reported positive and negative decision outcomes on the mentioned decision drivers, we could further assess which of the options are more or less favored in the microservice practitioner literature. For evaluation we used 27 microservice component architecture models, summarized in Table I and described in Section IV.

A. Decisions

1) Decision: Persistent Data Storage of Services: This decision is about how persistent data storage is handled for services, if any is needed. The following decision options can be chosen: *No Persistent Data Storage* is applicable only for services whose functions are performed on transient data. The most recommended option is *Database per Service* pattern [3], in which each service has its own database and manages its own data independently. Another option, is to use a *Shared Database* [3]: two or more services read to and write from a common database. This option has two alternatives: *Data Shared via Shared Database* in which multiple services share the same table, resulting in a strongly coupled system. In contrast in the *Database Shared but no Data Sharing* option, each service writes to and reads from its own tables, which has a lesser impact on coupling.

In our previous work, we have empirically defined two metrics that can be used to assess conformance to each of the decision options:

- *Database Type Utilization* to measure the proportion of services that are using individual databases.
- *Shared Database Interactions* to measure the proportion of interconnections via a *Shared Database* among the *total number of service interconnections*.

2) Decision: Service Interconnections: Another important aspect in microservices is how the services communicate between each other. The decision is about how tightly service are coupled via their interconnections. *No Service Interconnection* is an optimal option but in reality this is not applicable. One other option is *Synchronous Service Interconnections* which is usually not the favored option in microservice systems. A number of asynchronous alternative options exist. One of these is *Asynchronous Direct Interconnections*, in which all the services communicate asynchronously via direct invocations. Another option is asynchronous communication through intermediary components. These can be *Pub/Sub Interconnections* [11], in which services publish and subscribe to events between each other, maybe combined with *Event Sourcing*; *Messaging Interconnections* [11], in which services produce and consume messages that are stored in a message broker; *Asynchronous Interconnections via API Gateway* [3], where services route asynchronous invocations via the API Gateway; *Shared Database Interconnection*, in which services interact via a shared database—every communication that is happening in this way is considered as asynchronous. Please note that the last option is beneficial over synchronous invocations in this decision, but leads to other problems, including shared-database interactions from the previous decision.

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

- *Asynchronous Communication Utilization* to measure the proportion of asynchronous service interconnections in the system.

- *Service Interactions via Intermediary Components* to measure the proportion of service interconnections via asynchronous relay architectures, such as *Message Brokers*, *Publish/Subscribe*, or *Stream Processing*.

3) Decision: Dependencies through Shared Services: Optimally, in a microservice-based system, services should not share other services all together at least not in a strongly coupled fashion. There are many patterns that are related to system structures avoiding service sharing. Especially in large scale systems, service sharing can lead to additional issues such as a chain of transitive dependencies between services and severe maintenance issues. We have identified four decision options for this decision: First, the optimal case, the might be *No Shared Services*. There are three cases containing some service sharing: *Directly Shared Service* in which two or more services are directly connected to other service(s); *Transitively Shared Service* in which a service is linked to other services via at least one intermediary service creating a transitive chain; and *Cyclic Dependency* [12] in which services create a direct or transitive path that leads back to the initial service. Cyclic dependencies are considered as highly problematic since the services can no longer be changed, understood, or tested in isolation.

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

- *Direct Service Sharing* to measure the proportion of directly shared elements in the system.
- *Transitively Shared Services* to measure the proportion of transitively shared elements in the system.
- *Cyclic Dependencies Detection* to detect the presence of *at least one* cyclic dependency in the system.

III. Related Work

Microservice best practices have been widely examined in various studies. A collection of microservice patterns and practices has been published by Richardson [3] and another collection of practices related to event-driven microservice architectures has been published by Skowronski [8]. Zimmermann et al. [7] introduced patterns related to microservices APIs. Fowler and Lewis [13] have discussed microservice fundamentals and best practices. Pahl and Jamshidi [1] have summarized many of those in a mapping study. Microservice "bad smells" have been studied by Taibi and Lenarduzzi [14], which correspond to violations in our work.

There is a number of studies that focus on techniques for detecting design or architecture smells, which are considered as violations in our case, but most of them are not specific to the microservices domain. A catalog of architectural bad smells using a format has been published in Garcia et al.'s [15], [16] studies. These studies also propose possible techniques for identification of these architecture smells. Le et al. [17] examined the relations between smells and a project's issues. They further examined the detection of multiple types of architecture smells. A number of detection strategies that take advantage of metrics-based rules for detecting design flaws have been presented by Marinescu [18]. Garcia et al. [19] present a machine learning-based technique for recovering an architectural view containing a system's components and connectors, which aims at detecting architecture drift or erosion. A prototypical tool for architecture recovery of microservice-based systems (MicroART) is presented by Granchelli et al. [20]. Alshuqayran et al. [21] suggest a microservice architecture recovery approach based on a meta-model and rules for mapping artifacts to it. A multivocal literature review, focused on identifying architectural smells for independent deployability, horizontal scalability, fault isolation and decentralisation of microservices, as well as suggesting refactorings to resolve them, is presented by Brogi et al. [22].

Although these works study various aspects of architecture violation detection, and some of them investigate aspects related to the microservice domain, none covers detecting and addressing coupling-related violations in a microservice context. In contrast, our work investigates in detail coupling-related aspects such as data persistence, communication types, and shared service dependencies. As a direct benefit of this, we expect that, in the context of loose coupling, our work produces more accurate detections of decision-specific violations and more targeted suggestions for fixes than those other works possibly could. As a downside, our work 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 Figure 3). 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.

IV. Research and Modeling Methods

In this section, we summarize the main research and modeling methods applied in our study. For reproducibility, all the code and models produced in this study will be made available online, as an open access data set in a long-term archive[1].

A. Research Method

Figure 2 shows the research steps of this study. In Section II we have already explained in detail the architectural decisions and the model-based metrics on which this study is based. In Section V we present a) precise definitions and algorithms for the detection of possible violations for each decision option, and b) precise definitions and algorithms for the possible fixes for each violation.

To evaluate our approach, we have applied it on a dataset of 27 models, summarized in Table I. This dataset comprises microservice-based systems from 9 independent sources, developed by practitioners and published in public repositories

[1]https://doi.org/10.5281/zenodo.4491583

and practitioners' blogs. We assume that these systems are, or reflect, real-world practical examples of microservice architectures. However, as many are open-source systems for demonstrating practices or technologies, they are, at most, of medium size and modest complexity. For the specification of our Microservice Component Architectures meta-model and the calculation of all metrics, violation detection, and fixes, we used CodeableModels[2], a Python tool for the precise specification of meta-models, models, and model instances in code. Based on the meta-model, we manually created model instances for every collected system, and realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all model instances (such as the one in Figure 3 used in an example below).

Our approach is designed to detect all violations for every model in our data set, and perform all possible suggested architecture refactorings (fixes) to it. This we did recursively, 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 the refactored model was identical to a previous version. In the latter case, this means that it is not possible to fix all violations, since resolving one violation will require introducing 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.

B. Using the Approach in a Continuous Delivery Pipeline

Figure 1 shows the position of our approach in a *delivery pipeline* in which every commit triggers an iterative loop of improvements. We place our approach after initial tests have been run, i.e. where usually code coverage and similar checks are run. In this stage we perform the *metrics calculation process* and if a violation is detected we determine the specific type of violation and provide a set of *fix options*. The *architect* or *developer* can select the optimal fix or to perform no more fixes. This triggers the *automatic architecture refactoring process* and a new version of the system component is generated. The *metrics calculation process* is performed again for the new version to evaluate the improvements. If there is no more violation after the fix process we continue in pipeline process. Of course, the approach can be equally applied to more complex systems with multiple delivery pipelines, for example, by mining docker files or other runtime logs in order to reconstruct the architecture.

V. Approach Details

In this section, we first give an overview of the violations and possible fixes we have identified, as well as the algorithms we have developed to detect violations or enact fixes. Also, we give detailed examples from the *Dependencies through Shared Services* decision to illustrate the approach.

We base our violation and fix definitions on the notion of a microservice-based 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**.

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

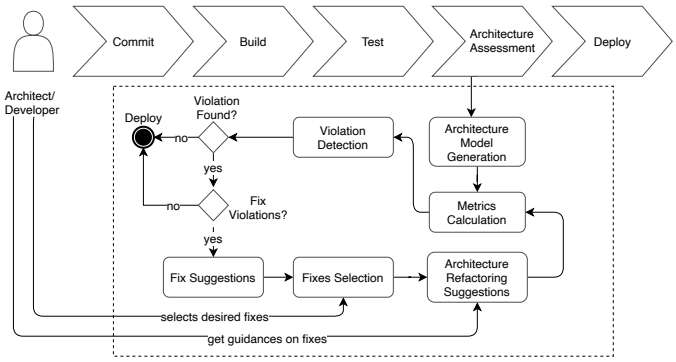

Fig. 1: Placing of our approach in a delivery pipeline

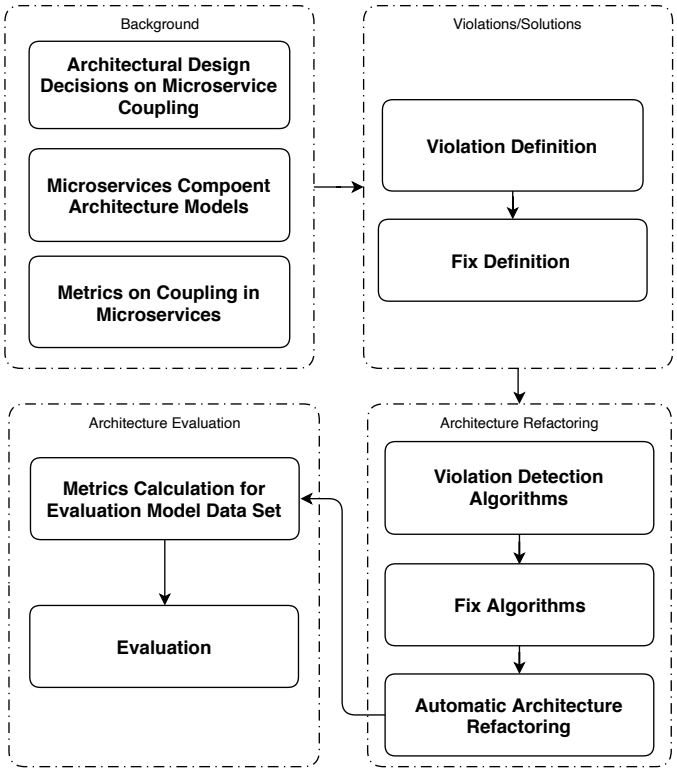

Fig. 2: Overview diagram of the research method followed in this study

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 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 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 I: Selected Models: Size, Details, and Sources

- 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.

A. Violation Detection

Table II summarizes the possible violations we have identified for each of the decisions. The table also describes in detail how the algorithms work that we use for detecting the violations in models based on our model definition above. In Algorithm 1 we exemplary detail the algorithm for detecting the Directly Shared Services Violation of Decision D3. It returns a list of violations, each described by a set of two services connectors in which two services s_i and s_j share a service s_m. Its sibling for the Transively Shared Services Violation is shown in Algorithm 2. This one returns a list of all service sets in which two services s_i and s_j share a service s_m via intermediaries.

Algorithm 1: Detect Directly Shared Services Violation

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

Algorithm 2: Detect Transitively Shared Services Violation

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

B. Fixes

Table III 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 with default values or approaches fully automatically. Many of them require human review by the architect and sometimes a human decision to be applicable. For example, the architects can be presented with a choice of an intermediary component to use to replace cyclic links or which of a set of transitive connectors should be deleted. That is, our fix approach is intended to be used as guidance to architects in a feedback loop (as illustrated in Figure 1), not to replace them.

Please note that some obvious details of the algorithms that are repetitive have been omitted for space reasons. For example, if connectors are replaced, existing stereotypes and annotations on them that are not related to the type of provided replacement must be retained on the new connectors. To illustrate this consider a *RESTful HTTP, Synchronous* connector is replaced with a *RESTful HTTP, Asynchronous* connector. Obviously, the stereotype *Synchronous* changes to *Asynchronous* during the fix, but also it must be ensured that the *RESTful HTTP* annotation is retained.

The Algorithms 3–6 exemplary present the fixes for Decision D3 and its Violation V1 in detail. They represent the identically named fixes D3.V1.F2–D3.V1.F5 respectively. For explanations of each fix, please study Table III.

Algorithm 3: Remove Connectors of Directly Shared Services

```
input: Model M, Set<Tuple> violation
output: –
begin
  for (s_i, s_m) ∈ violation:
    if (s_i, s_m) ∈ service_connectors(M):
      delete_connector(M, (s_i, s_m))
end
```

Algorithm 4: Remove Connectors of Directly Shared Services

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

Algorithm 5: Integrated Shared Services into Calling Service

```
input: Model M, Set<Tuple> violation
output: –
begin
  integration_annotations ← ∅
  for (s_i, s_m) ∈ violation:
    integration_annotations ← integration_annotations ∪
      "integrated functionality from: " +
      get_service_name(M, s_i)
    for connector ∈ incoming_connections(M, s_i):
      change_target(model, connector, s_m)
    delete_service(M, s_i)
  add_annotations(M, s_m, integration_annotations)
end
```

Algorithm 6: Integrated Calling Service into Calling Services

```
input: Model M, Set<Tuple> violation
output: –
begin
  for (s_i, s_m) ∈ violation:
    for connector ∈ outgoing_connections(M, s_m):
      change_source(M, connector, s_i)
    add_annotations(M, s_i,
      {"integrated functionality from: " +
       get_service_name(M, s_m)})
  delete_service(M, s_m)
end
```

Violation	*Violation Detection Algorithm Summary*
D1: Persistent Data Storage of Services	
D1.V1: Services have a shared database, but no data is shared via the shared database	The models are traversed for finding database accesses. Database accesses by more than one service are inspected for the data entities that are read and written. If none of those data entities are shared by two services, this violation is raised. All violating data accesses (including services, databases, and connectors involved) are returned by the detector operation.
D1.V2: Services have a shared database and data is shared via the shared database	The models are traversed for finding database accesses. Database accesses by more than one service are inspected for the data entities that are read and written. If *at least one* of those data entities is shared by *at least two services*, this violation is raised. All violating data accesses (including services, databases, and connectors involved) are returned by the detector operation.
D2: Service Interconnections	
D2.V1: System services communicate synchronously	All service connectors in the model are traversed. If any synchronous connector is encountered, the violation is raised and the list of all synchronous connectors is returned by the detector operation.
D3: Dependencies through Shared Services	
D3.V1: Directly shared services	All services in the model are traversed, and it is checked whether two services share another service via directly linking connectors. If this is the case, a violation is raised. Each pair of shared service connectors that is found is returned by the detector operation.
D3.V2: Transitively shared services	All services in the model are traversed, and it is checked whether two services share another service via transitively linking connectors. Transitive means here via any number of intermediary other services. If this is the case, a violation is raised. Each pair of shared service connectors that is found is returned by the detector operation.
D3.V3: Cyclic Dependency	On the graph of services and connectors in the model we run a cycle detection based on a *depth-first search algorithm*. If we detect a cycle, a violation is raised. All detected cycles are returned as a list of sets of connectors participating in the respective cycle.

TABLE II: Identified Violations and Violation Detection Algorithms

C. Violation Detection and Fixes Example

In Figure 3 the model TH1 from Table I is shown. As an illustrative example, we use it here to demonstrate the *Directly Shared Services Violation (D3.V1)* and possible fixes. In this model the *Payment* service is called directly by services *Passenger Management* and *Drive Management*. Here, the *Payment* service is considered as *shared service* causing the *Directly Shared Services Violation*. It would be triggered in our approach by providing a bad metric value, which would trigger the detailed detection, which would return the $\{(\textit{Passenger Management, Payment}), (\textit{Passenger Management, Payment})\}$ set of tuples. If we run our fix algorithms the resulting model fix suggestions are for instance:

- *Applying Fix D3.V1.F2:* The architect can decide that the connectors or one of them is not really needed or can be replaced by some other manual refactoring. If this is the case, the connectors can get removed by this fix.
- *Applying Fix D3.V1.F3: Payment* service will be disconnected from *Passenger Management* and *Drive Management* services and connected to *API Gateway* (all interactions will be happening via the API Gateway). Alternatively, this fix could also introduce a new *intermediary component* (e.g., Pub/Sub) and all the involved services will be connected to it with *publish* and *subscribe* operations for the data exchange.
- *Applying Fix D3.V1.F4:* The *Passenger Management* and *Drive Management* services can be integrated into *Payment* service.
- *Applying Fix D3.V1.F5:* The *Payment* service can be integrated to *Passenger Management* and *Drive Management* services, if that's possible.

In all these fixes the identified *directly shared services violation* would get fixed.

VI. Evaluation

For evaluating our work, we have fully implemented our algorithms for detecting violations and performing fixes, as well as generating the set of metrics described in Section II 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 TH1 Model (see Figure 3). TH1 violates two of the decisions—"System services communicate synchronously" (D2.V1) fully and "Directly shared services" (D3.V1) partially, as indicated by the two respective measures (0.00 and 0.67 respectively) in Table IV. The incremental refactoring process is illustrated in Figure 4. At the first iteration step, there are two branches, depending on which violation is dealt with first. The first iteration step results in 7 possible model variants, one for each fix option from Table III. Out of those, the model variant F shows no further violations (i.e., the fix for D2.V1 has also coincidentally fixed violation D3.V1 and thus optimally resolved all violations). In model variant G the violation D3.V1 was also coincidentally fixed, but this fix introduced a new "Services have a shared database and data is shared via the shared database" (D1.V2) violation. Thus, after the first iteration, there are 6 new model variants that still contain a violation.

The second iteration step results in further 18 models. In turn, 4 of the resulting models now exhibit violation D1.V2, requiring a third step to be resolved. At the end of the third step, we have 23 suggested model variants (A1–A2, A3_1–A3_2, B1–B2, B3_1–B3_2, C1–C2, C3_1–C3_2, D1–D2, D3_1–D3_2, E1–E4. F, G1–G2) which all optimally 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* 27 system models in Table I. The resulting number of intermediate models and violation instances per step, and the number of final suggested models with an optimal assessment of 1.00, are given in Table IV, along with the initial violations and architecture assessment values for each model. Please note that the metrics reported here are the ones associated to each of the violations; most metrics from Section II match one-to-

Violation	*Fix*	*Fix and Fix Algorithm Summary*
D1: Persistent Data Storage of Services		
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 new service-specific databases and migrate database accesses	Disconnect services from the shared database and introduce a new database per service. Migrate each service-specific database access to the respective service-specific database. The architect needs to check if this fix is possible and, if applied, whether the original database can be deleted after the migration.
	D1.V1.F3: Merge services with shared database into a single service using one database	Merge the services using the same shared database into a single service using that database. The architect needs to check whether such integration is possible and can provide annotations for implementers about the details of the envisaged service integration.
D1.V2	*D1.V2.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.V2.F2: Migrate to new database and add consistency mechanism for the shared data	Same as Fix D1.V1.F3. In addition, add consistency mechanism for the data shared between services. The architect needs to select the consistency mechanism applied (e.g., eventual consistency via an event store). Then connectors will be extended with respective stereotypes and exchange of data-related events.
	D1.V2.F3: Merge Services with Shared Database into a single service using one Database	Same as Fix D1.V1.F3.
D2: Service Interconnections		
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: Replace synchronous direct interconnection with asynchronous direct interconnection	Disconnect all the synchronously connected services and connect them using asynchronous connectors with the same stereotypes as the synchronous ones. Delete the synchronous connectors.
	D2.V1.F3: Replace synchronous direct interconnection 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 synchronous direct interconnections with asynchronous interconnections via this component. Delete the synchronous connectors.
	D2.V1.F4: Introduce communication by writing to and reading from common databases	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 synchronous connectors, introduce communication by writing to and reading from this database. Delete the synchronous connectors. Please note, while this fix repairs this violation, it leads to a violation of D1.
D3: Dependencies through Shared Services		
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 connectors of directly shared services	Change the connections between services and remove connectors between the involved services. This fix is only applicable, if a solution makes sense that performs the same functionality without those connectors. This must be judged by a human architect.
	D3.V1.F3: Replace direct links 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 direct interconnections with asynchronous interconnections via this component. Delete the direct connectors.
	D3.V1.F4: Integrate shared services into calling service	Integrate the responsibility and functionality of the *shared services* (i.e., the services that are called by two or more services) into *calling services* (services that call a shared service), and delete the shared services and any connectors accessing them. Here we add annotations that functionality has been added to the calling service, so that implementers later on can realize this functionality.
	D3.V1.F5: Integrate calling service into shared service	Integrate the responsibility and functionality of the *calling services* into *shared services*, delete the calling service, and rewire their clients directly to the shared services. Here we add annotations that functionality has been added to the shared services, so that implementers later on can realize this functionality. Which functionality goes to which shared service can be further annotated by the architects.
D3.V2	*D3.V2.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.V2.F2: Remove connectors of transitively shared services	The architect needs to decide which of the connectors in each transitive link path are safe to delete. It must then be checked that this is enough to break up the transitive sharing. Then: Same as D3.V1.F2 for selected connectors.
	D3.V2.F3: Replace direct links via an intermediary component, e.g., API Gateway, Pub/Sub, Message Broker	As in D3.V2.F2 the architect needs to select connectors, and a check that sharing is broken needs to be performed. Then: Same as D3.V1.F3 for the selected connectors.
	D3.V2.F4: Integrate transitively shared services into a calling service	The architect has to select which services on the transitive link path are to integrated into the calling service. It must then be checked that this is enough to break up the transitive sharing. Then: Same as D3.V1.F4 for the selected services. The
	D3.V2.F5: Integrate transitively calling service into shared service	As in D3.V2.F4 the architect needs to select services, and a check that sharing is broken needs to be performed. Then: Same as D3.V1.F5 for the selected services.
D3.V3	*D3.V3.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.V3.F2: Replace cyclic relations 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 all connectors in a cycle with connectors to this component. Delete the cyclic connectors.
	D3.V3.F3: Remove connectors from a cycle until there is no cycle	The architect selects connectors that can be deleted in a cyclic path. It is then checked whether the cycle is broken by those deletions. Then the selected connectors are deleted. If selected by the architect, the steps from D3.V3.F2 can then be followed to introduce links via an intermediary component instead.
	D3.V3.F4: Integrate all functionality of services involved to a cycle into one service	Integrate the responsibility and functionality of the services in the cycle into one *integration service*, selected by the architect. This can also be a new service, introduced by the architect. Rewire the cyclic services' clients directly to the integration service. Here we add annotations that functionality has been added to the integration service, so that implementers later on can realize this functionality.

TABLE III: Identified Fixes And Fix Algorithms

one, only D2.V1 has two metrics associated (i.e., both from Section II-A2). Please also note that for D2.V1 to be fixed, it is enough that one of the two metrics is optimal (1.00); this is why the models BM3, CO2, and EC2 require no refactoring steps, even though they score less than 1.00 in the other of the two metrics. The metrics measure the degree of the violation, with *1.00* when *no violation exists* and *0.00* where the worst possible option is selected and not even partial conformance is measured. 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 (as explained in detail in the TH1 example above). As can be seen in Table IV, *all* models are *fully resolved*—i.e., all assessment metrics are 1.00—after *at most* four steps.

VII. Discussion of Research Questions

To answer **RQ1** we have systematically specified a number of decision-based violations related to each possible decision

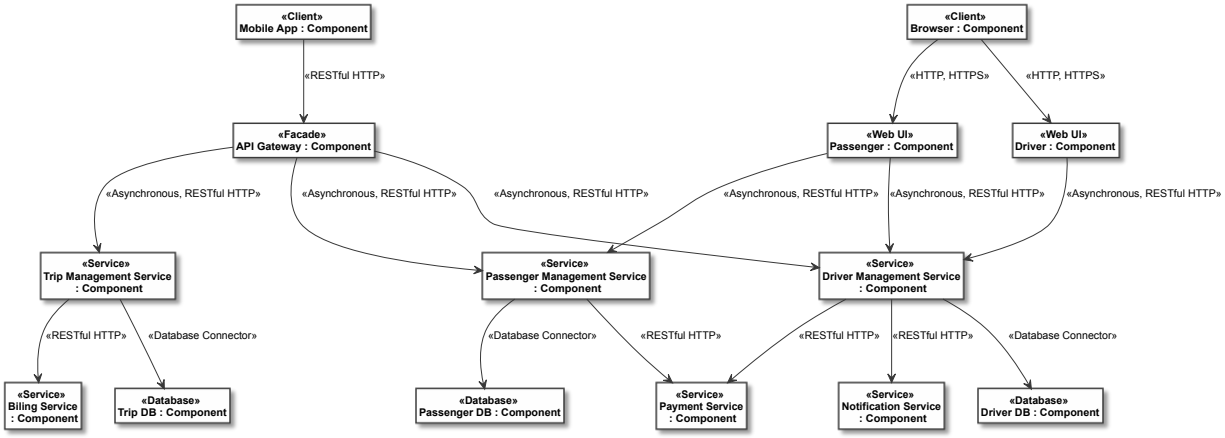

Fig. 3: Example of an Architecture Component Model of model TH1 in Table I: this architecture violates the Service Interaction (D2.V1) and Dependencies through Shared Services (D3.V1) decisions (cf. Table II).

Model ID	Initial Model Assessments						Models Generated / Remaining Violation Instances per Refactoring Step				Resulting Suggested (Optimal) Models
	D1.V1	*D1.V2*	*D2.V1*	*D3.V1*	*D3.V2*	*D3.V3*	*Step 1*	*Step 2*	*Step 3*	*Step 4*	
BM1	0.33	1.00	1.00, 0.00	1.00	1.00	1.00	2 / 0	–	–	–	2
BM2	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
BM3	1.00	1.00	0.00, 1.00	1.00	1.00	1.00	–	–	–	–	–
CO1	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
CO2	1.00	1.00	1.00, 0.00	1.00	1.00	1.00	–	–	–	–	–
CO3	0.60	0.00	0.00, 1.00	1.00	1.00	1.00	2 / 0	–	–	–	2
CI1	1.00	1.00	0.00, 0.00	0.75	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
CI2	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
CI3	1.00	1.00	0.00, 0.00	0.70	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
CI4	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
EC1	1.00	1.00	0.00, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
EC2	1.00	1.00	1.00, 0.00	1.00	1.00	1.00	–	–	–	–	–
EC3	0.00	0.00	0.00, 1.00	1.00	1.00	1.00	2 / 0	–	–	–	2
ES1	1.00	1.00	0.60, 0.00	0.73	1.00	1.00	7 / 1	2 / 0	–	–	8
ES2	0.00	1.00	0.00, 0.00	0.66	1.00	1.00	9 / 10	26 / 11	28 / 2	4 / 0	45
ES3	0.33	1.00	0.00, 0.00	0.66	1.00	1.00	9 / 10	26 / 11	28 / 2	4 / 0	45
FM1	1.00	1.00	0.00, 0.00	0.38	1.00	1.00	7 / 2	6 / 0	–	–	11
FM2	1.00	1.00	0.00, 1.00	0.70	1.00	1.00	4 / 0	–	–	–	4
FM3	1.00	1.00	0.00, 1.00	1.00	0.77	0.00	7 / 0	–	–	–	7
HM1	1.00	1.00	0.00, 0.42	0.80	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
HM2	1.00	1.00	0.80, 0.20	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4
RM1	0.00	1.00	1.00, 0.00	1.00	1.00	1.00	2 / 0	–	–	–	2
RM2	1.00	1.00	0.00, 0.00	1.00	0.82	1.00	7 / 6	18 / 4	8 / 0	–	23
RM3	1.00	1.00	0.00, 0.00	1.00	0.82	0.00	10 / 7	30 / 7	14 / 0	–	38
RS	0.66	1.00	0.11, 0.11	0.63	1.00	1.00	9 / 14	37 / 14	61 / 5	10 / 0	89
TH1	1.00	1.00	0.00, 0.00	0.67	1.00	1.00	7 / 6	18 / 4	8 / 0	–	23
TH2	1.00	1.00	0.66, 0.00	1.00	1.00	1.00	3 / 1	2 / 0	–	–	4

TABLE IV: 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. Figure 4 for a detailed example).

option, summarized in Table II. As we have empirically shown in our prior work [9] that the metrics described in Section II can reliably distinguish favored or less favored design options, the role of the violation detectors is to find the precise spots 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 III). 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

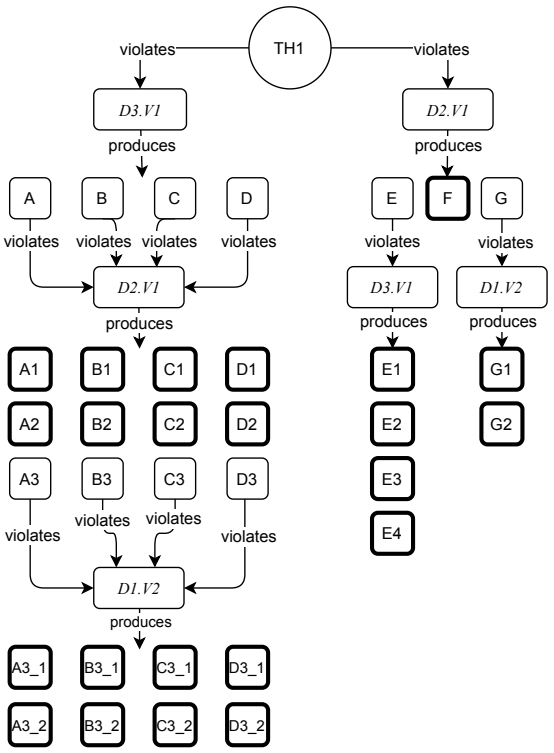

Fig. 4: Example of an exhaustive iterative application of our approach in the TH1 model. Final (i.e., optimally resolved) resulting models are thickly outlined.

have shown (cf. Table IV) that an iterative approach of using our algorithms successively, 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.

VIII. Threats to Validity

To increase the internal validity of our approach, throughout our study, as well as in the previous works on which it builds, we have relied on a large number of systems produced by third parties for testing and evaluation (cf. Table I). Likewise, the solutions we propose are derived from best practices and patterns in the published (grey) literature: our work is limited to gathering, systematizing, and applying them to the given set of systems. Any omissions can be added to our model without invalidating the fundamental approach. One possible threat to the internal validity of our algorithms is that they depend on the particular modelling approach we have adopted. However, this approach is by design generic, based on typical component-and-connector models, and should be both capable of dealing with most microservice systems, as well as be easy to extend or adapt if required.

Our approach presently has some limitations: it operates at a relatively high level of abstraction, does not consider any decision parameters other than the metrics evaluating pattern conformance, and limits itself to specific, coupling-related patterns. We consider that these limitations can be addressed in future work, by applying the same fundamental, metrics-based approach to finer-granularity parameters , enriching our decision support model with additional data sources , and adding more patterns to our decision model. In terms of generalizability, we have not considered industrial-scale systems with hundreds of services, but are confident that the fundamental approach is sound. The challenge would be one of adapting our method and increase its 'intelligence' so as to target contiguous portions of large-scale systems at a time, so that the generated refactored models would be realistically feasible. The solutions proposed may also not be always optimal, as we aim to present generically applicable solutions. It remains possible that an architect will devise a custom, hybrid solution that optimizes a system in ways that an automated approach cannot. Again, however, this is a matter of extending the present approach with the metrics and the 'intelligence' required to model these hybrid solutions.

IX. Conclusion and Future Work

In this paper we have presented a set of violations for three ADDs and mapped them to existing, empirically validated metrics judging the outcomes of these decisions. We have defined automatic detectors for these violations, which provide the precise places in a model where the violations occur. Bases on this, we have defined a set of possible fixes for each violation, as part of an approach for ensuring pattern and best practice conformance in microservice-based architectures. We have evaluated our approach on a set of 27 models showing various degrees of pattern violations and architecture complexity, and have shown that our approach is capable of resolving these violations in at most 4 refactoring steps that can largely be automated. As both metric calculation and violation detection are fully automated, the approach can be applied as an additional "architecture assessment test" in a continuous delivery pipeline, as was our goal. Fixes are automated, too, but architects have to decide which of the suggested options should be applied (if any) and sometimes need to provide some input. Thus the approach is still flexible enough to let the architect make 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.

X. Acknowledgments

This work was supported by: FFG (Austrian Research Promotion Agency) project DECO, no. 864707; FWF (Austrian Science Fund) project API-ACE: I 4268.

References

[1] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," in *6th International Conference on Cloud Computing and Services Science*, 2016, pp. 137–146.

[2] C. Pautasso and E. Wilde, "Why is the web loosely coupled?: a multi-faceted metric for service design," in *18th Int. Conf. on World wide web*. ACM, 2009, pp. 911–920.

[3] C. Richardson, "A pattern language for microservices," http://microservices.io/patterns/index.html, 2017.

[4] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in *Int. Conf. on the Quality of Software Architectures*. Springer, 2007, pp. 15–32.

[5] O. Zimmermann, "Microservices tenets," *Computer Science - Research and Development*, vol. 32, no. 3, pp. 301–310, Jul 2017.

[6] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1: Reality check and service design," *IEEE Software*, vol. 34, no. 1, pp. 91–98, Jan 2017.

[7] O. Zimmermann, M. Stocker, U. Zdun, D. Luebke, and C. Pautasso, "Microservice API patterns," https://microservice-api-patterns.org, 2019.

[8] J. Skowronski, "Best practices for event-driven microservice architecture," https://hackernoon.com/best-practices-for-event-driven-microservice-architecture-e034p21lk, 2019.

[9] E. Ntentos, U. Zdun, K. Plakidas, S. Meixner, and S. Geiger, "Assessing architecture conformance to coupling-related patterns and practices in microservices," in *14th European Conference on Software Architecture (ECSA), 2020*, September 2020. [Online]. Available: http://eprints.cs.univie.ac.at/6478/

[10] ——, "Metrics for assessing architecture conformance to microservice architecture patterns and practices," in *18th International Conference on Service Oriented Computing (ICSOC 2020)*, December 2020. [Online]. Available: http://eprints.cs.univie.ac.at/6479/

[11] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[12] M. Goldstein and D. Moshkovich, "Improving software through automatic untangling of cyclic dependencies." New York, NY, USA: Association for Computing Machinery, 2014.

[13] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," http://martinfowler.com/articles/microservices.html, Mar. 2004.

[14] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

[15] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Identifying architectural bad smells," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 255–258.

[16] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic, "Toward a catalogue of architectural bad smells," in *Architectures for Adaptive Software Systems*, R. Mirandola, I. Gorton, and C. Hofmeister, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 146–162.

[17] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic, "An empirical study of architectural decay in open-source software," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 176–17 609.

[18] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359.

[19] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Yuanfang Cai, "Enhancing architectural recovery using concerns," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 552–555.

[20] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Microart: A software architecture recovery tool for maintaining microservice-based systems," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 298–302.

[21] N. Alshuqayran, N. Ali, and R. Evans, "Towards micro service architecture recovery: An empirical study," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 47–4709.

[22] D. Neri, J. Soldani, O. Zimmermann, and A. Brogi, "Design principles, architectural smells and refactorings for microservices: a multivocal review," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1-2, p. 3–15, Sep 2019. [Online]. Available: http://dx.doi.org/10.1007/s00450-019-00407-8