Applying a Multi-platform Architectural Conformance Solution in a Real-world Microservice-based System

Elena A. Araujo
elena.araujo@posgrad.ufla.br
Universidade Federal de Lavras
Lavras, MG

Álvaro M. Espíndola
alvaro.espindola@estudante.ufla.br
Universidade Federal de Lavras
Lavras, MG

Vinicius Cardoso Garcia
vcg@cin.ufpe.br
Universidade Federal de Pernambuco
Recife, PE

Ricardo Terra
terra@ufla.br
Universidade Federal de Lavras
Lavras, MG

ABSTRACT

Microservice architectures are composed of a set of independent microservices that execute well-defined functionalities, allowing each one to be developed in different programming languages and data management technologies. The problem, however, is that such heterogeneity implies in a harder verification process of communication among microservices and the architectural designs of each microservice. Although the state-of-the-art provides several architectural conformance solutions, none formally restricts communications (e.g., over HTTP) between different systems. Even stable and industrial solutions—such as Kubernetes, Terraform, and Docker Compose—provide basic mechanisms to restrict communications between microservices. Thereupon, this paper proposes and evaluates a multi-platform architectural conformance solution for the microservice architecture. For this purpose, (i) we specify an architectural constraint language, called DCL$^+$—adapted from the DCL (Dependency Constraint Language) language; (ii) we propose a multi-platform process that restricts the communication between the microservices and also verifies the architectural projects of each one of them; (iii) we develop DCL+check, a tool that implements the proposed solution; (iv) we apply our process in a medium-size real-world application composed of eleven microservices, developed in two different languages (JavaScript and Java). As result, we found 16 communication and 171 structural design violations. The communication violations occurred in general due to the lack of knowledge of the developers about the restrictions of communication among the modules of the orchestrator system and other microservices, as well as the evolution of two microservices.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; *Software maintenance tools*; **Architecture description languages**; Domain specific languages.

SBCARS '20, October 19–20, 2020, Natal, Brazil

ACM ISBN 978-1-4503-8754-5/20/09…$15.00
https://doi.org/10.1145/3425269.3425270

ACM Reference Format:
Elena A. Araujo, Álvaro M. Espíndola, Vinicius Cardoso Garcia, and Ricardo Terra. 2020. Applying a Multi-platform Architectural Conformance Solution in a Real-world Microservice-based System. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '20), October 19–20, 2020, Natal, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3425269.3425270

1 INTRODUCTION

Microservice architecture is an approach to develop a single application and a complex and wide-scale ecosystem of applications and services, each running on its own processes, communicating, technologies, data schemas among others architectural decisions [2, 8]. Therefore, microservices are fully autonomous, which means they can be developed in different programming languages, using different frameworks, persisting data in different DBMSs, and hosted in different application servers [3].

Such heterogeneity, however, implies in a harder verification process of communication between microservices and the architectural designs of each microservice. For instance, we are not aware of cross-platform solutions whose precise goal is to support developers in enforcing communication and structural design constraints in the microservice architecture.

This paper claims that the lack of conformance solutions disregarding communications between microservices can nullify the benefits provided by such architectural style, such as maintainability, scalability, portability, etc. [14, 18]. This phenomenon is known as software architecture erosion and it is considered a challenging research problem in the software architecture area [4, 5, 12, 21, 24].

As an example, as illustrated in Figure 1, assume an application composed of three microservices: `Ms1` is written in JavaScript through Node.js, stored in a Mercurial repository, and runs in Italy; `Ms2` is written in Java, stored in a Git repository, and runs in Canada; and `Ms3` is written in C#, stored in a SVN repository, and runs in USA. The *problem* is how we can restrict the communication between these microservices?

This paper, therefore, proposes and evaluates an architectural conformance[1] process for the microservice architecture. We provide software architects with a multi-platform solution that allows (i) the

[1]The term *architectural conformance* refers to the checking if the implemented architecture (a.k.a., source code architecture) is in accordance with the planned one.

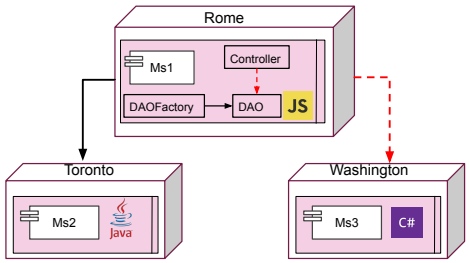


Figure 1: An example of application composed by microservices.

verification of the structural design of each microservice and (ii) the restriction of communication between microservices.

As an example of (i), the planned architecture of `Ms1` specifies that only the `DAOFactory` class can create DAO (*Data Access Object*) objects. If another class instantiates DAO objects, we report a *structural design* violation. There are a bunch of architectural conformance solutions that could detect such violation and our goal is to provide an integrated and multi-platform solution based on the DCL language [22, 23] (*complementary contribution*).

As an example of (ii), a constraint specifies that `Ms1` can communicate *only* with `Ms2`. If `Ms1` also communicates with `Ms3`, our solution reports a *communication* violation (*our main contribution*). Indeed, there are stable and industrial solutions—such as Kubernetes[2], Terraform[3], and Docker Compose[4]—that can somehow address communications violations, besides Istio[5] and Kiali[6] that provide mechanisms to control and visualize the communication between services. Nevertheless, our approach brings some relevant advantages when considering architectural aspects:

(1) *Granularity.* Although the aforementioned solutions can forbid communications from a microservice A to B, our proposed solution can specify which classes inside A can or cannot communicate to B;

(2) *Required communications.* The aforementioned solutions can forbid communications but not ensure that a communication must be established as our proposed solution does;

(3) *On-developing checking.* When you forbid microservice A to communicate with B using the aforementioned solutions, such blocking will occur at runtime. Using our proposed solution, a forbid communication is presented to the developers as soon as they write the code down; and

(4) *More types of restrictions.* Our proposed solution provides other types of constraints using different semantics such as "can-communicate-only" that avoids the creation of several "cannot" constraints.

Our *main contribution for the state-of-the-art* is the solution formally restricts communications (e.g., over HTTP) between different systems. Therefore, this paper focuses mainly on communication aspects. We designed it as multi-platform and also included local structural design conformance to promote better applicability in real-world scenarios.

Our *main contribution for the state-of-the-practice* is an evaluation in a real-world microservice-based system where our solution was effectively able to detect 16 deviations in the communication architecture, besides 171 in the structural design architecture, which were unknown by its architects.

The remainder of this work is organized as follows. Section 2 introduces microservices and the constraint language our proposed language is based on. Section 3 describes our proposed DCL$^+$. Section 4 describes our architectural conformance solution. Sections 5 and 6 define the methodology we follow and evaluate the process in a real-world application composed of one orchestrator system developed in JavaScript and ten microservices developed in Java. Section 7 discusses related work and Section 8 concludes.

2 BACKGROUND

2.1 Microservice Architecture

According to Martin Fowler [7], the microservice architecture is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. The left side of Figure 2 illustrates the monolithic architecture where UI, business logic, and data access components are all together. In contrast, the right side of the figure illustrates the microservices architecture where the actual system is composed of many fully independent and autonomous systems, which are named microservices.

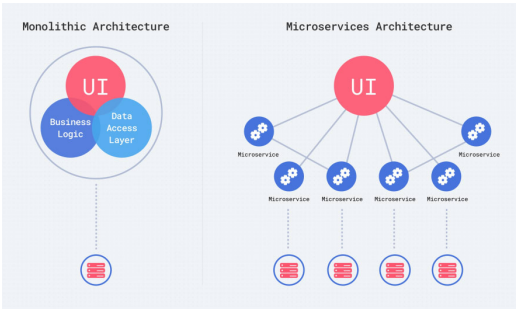


Figure 2: Monolithic vs Microservices.

Sam Newman [16], in a short way, defines as autonomous services that work together. It means they can be hosted in different servers in different countries, developed using different frameworks or programming languages, share or not the same database, etc. When someone modifies the implementation of a microservice, it does not impact other microservices since the communication occurs through well-defined interfaces usually using REST (*REpresentational State Transfer*). It allows microservices to be built, scaled, and tested independently [25].

We can summarize the key benefits as technology heterogeneity, resilience, scaling, ease of deployment, organizational alignment, composability, and optimizing for replaceability [16]. Although so many positive aspects of microservices, there is no way so far to restrict the communication among microservices.

[2]https://kubernetes.io/
[3]https://www.terraform.io/
[4]https://docs.docker.com/compose/
[5]https://istio.io/
[6]https://kiali.io/

2.2 Dependency Constraint Language (DCL)

DCL is a statically checked domain-specific language that allows software architects to restrict the spectrum of structural dependencies, which can be established in object-oriented systems [22, 23]. Particularly, the language provides constraints to capture two types of architectural violations: *divergences* (when a dependency that exists in the source code violates the planned architecture) and *absences* (when the source code does *not* establish a dependency that is prescribed by the planned architecture) [15, 18]. DCL also allows specifying the granularity of dependencies (fine-grained as *access*, *declare*, *create*, *throw*, *extend*, *implement*, and *useannotation*). To illustrate the use of DCL, assume the following constraints:

```
1 only Factory can-create DAO
2 Util can-only-depend Util, $API
3 View cannot-access Model
4 Entity must-implement Serializable
```

These constraints specifies that:

(line #1) only objects from module `Factory` can create objects from module `DAO`, i.e., if any other module of the system creates DAO objects such creation is considered a violation;

(line #2) classes from module `Util` can establish dependencies only with itself and the language API, i.e., if any class from `Util` establishes dependency with any other module such dependency is considered a violation;

(line #3) classes from module `View` cannot access classes in module `Model`, i.e., if a class from `View` accesses any class from `Model` such access is considered a violation; and

(line #4) all classes in module `Entity` must implement the `Serializable` interface, i.e., if a class from `Entity` do not implement `Serializable`, this absence is considered a violation.

3 DCL$^+$

The original DCL, which is currently maintained by our research group, does not addresses the particularities of the microservice architecture. Therefore, in this section, we propose DCL$^+$, which is an adaptation and extension of DCL for the microservice architecture that introduces a new dependency type called *communicate* to specify the set of microservices that a particular microservice is allowed to communicate, as illustrated in Figure 3.

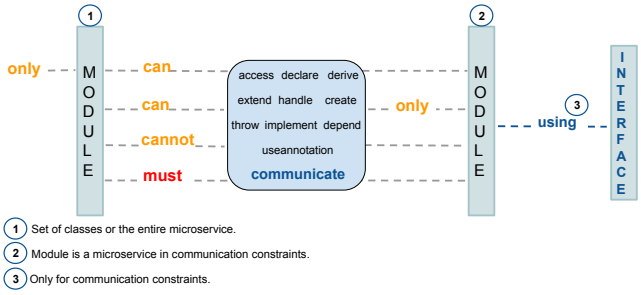


Figure 3: DCL$^+$ syntax.

Let S be the set of all microservices belonging to the application. Let A be a microservice, such that $A \in S$, M_A is the set of all modules belonging to A, and S_{M_A} is a subset of M_A modules. Similarly, let S_B be a set of microservices other than A ($S_B \subseteq S - \{A\}$). In this way, the new type of dependency is defined below.

Divergences: To detect wrong communication, *DCL*$^+$ supports the following types of restrictions between microservices:

- S_{M_A} *cannot-communicate* S_B: modules belonging to the subset S_{M_A} *cannot* communicate with S_B microservices. Formally, violations are defined as follows:
  $$\exists m \; \exists B \; [\; m \in S_{M_A} \;\wedge\; B \in S_B \;\wedge\; communicate(m, B) \;]$$
- S_{M_A} *cannot-communicate* S_B **using** I: modules belonging to the subset S_{M_A} *cannot* communicate with S_B microservices through I. Formally:
  $$\exists m \; \exists B \; [\; m \in S_{M_A} \;\wedge\; B \in S_B \;\wedge\; comm\_using(m, B, I) \;]$$

Constraints *only-can* and *can-only* are defined based on constraint *cannot*.

- *only* S_{M_A} *can-communicate* S_B: only modules belonging to the subset S_{M_A} *can* communicate with S_B microservices. Stated differently, any other module of the system except for S_{M_A} is forbidden to communicate with S_B microservices. Formally:
  $$only\ S_{M_A}\ can\text{–}communicate\ S_B \implies \overline{S_{M_A}}\ \ cannot\text{–}communicate\ S_B$$
- *only* S_{M_A} *can-communicate* S_B **using** I: only modules belonging to the subset S_{M_A} *can* communicate with S_B microservices through I. Formally:
  $$only\ S_{M_A}\ can\text{–}communicate\ S_B\ using\ I \rightarrow \overline{S_{M_A}}\ cannot\text{–}communicate\ S_B\ using\ I$$
- S_{M_A} *can-communicate-only* S_B: modules belonging to the subset S_{M_A} *can* communicate *only* with S_B microservices. Stated differently, S_{M_A} is forbidden to communicate with any other microservices of the system except for microservices defined in S_B. Formally:
  $$S_{M_A}\ can\text{–}communicate\text{–}only\ S_B \implies S_{M_A}\ \ cannot\text{–}communicate\ \overline{S_B}$$
- S_{M_A} *can-communicate-only* S_B **using** I: modules belonging to the subset S_{M_A} *can* communicate *only* with S_B microservices through I. Formally:
  $$only\ S_{M_A}\ can\text{–}communicate\text{–}only\ S_B\ using\ I \rightarrow S_{M_A}\ cannot\text{–}communicate\ \overline{S_B}\ using\ I$$

Absences: To detect missing communication, *DCL*$^+$ supports the following type of restriction between microservices:

- S_{M_A} *must-communicate* S_B: modules belonging to the subset S_{M_A} *must* (required) communicate with S_B microservices. Formally:
  $$\forall m \; !\exists B \; [\; m \in S_{M_A} \;\wedge\; B \in S_B \;\wedge\; communicate(m, B) \;]$$

- S_{M_A} *must-communicate* S_B **using** I: modules belonging to the subset S_{M_A} *must* (required) communicate with S_B microservices through I. Formally:

$$\forall m \, !\exists B \, [m \in S_{M_A} \wedge B \in S_B \wedge comm\_using(m, B, I)\,]$$

Alerts: It occurs when modules of a properly specified microservice communicates with a microservice unknown by the microservice architecture. Therefore, we raise an alert whenever an arbitrary element m establishes a *communication* with an element $E \notin S$.

4 THE PROPOSED ARCHITECTURAL CONFORMANCE SOLUTION

This paper proposes an architectural conformance process for the microservice architecture, as illustrated in Fig. 4. This process receives as input an *Application* composed of microservices (Fig. 4a) and its *Architectural Specification* (Fig. 4b). The execution of the proposed process is supported by the *Communication Extractor and Checker* (Fig. 4c), and the *Structural Design Extractor and Checker* (Fig. 4d), automated by the DCL$^+$*check*[7] tool. The process outputs a set of *communication* and *structural design* violations (Figs. 4e and f).

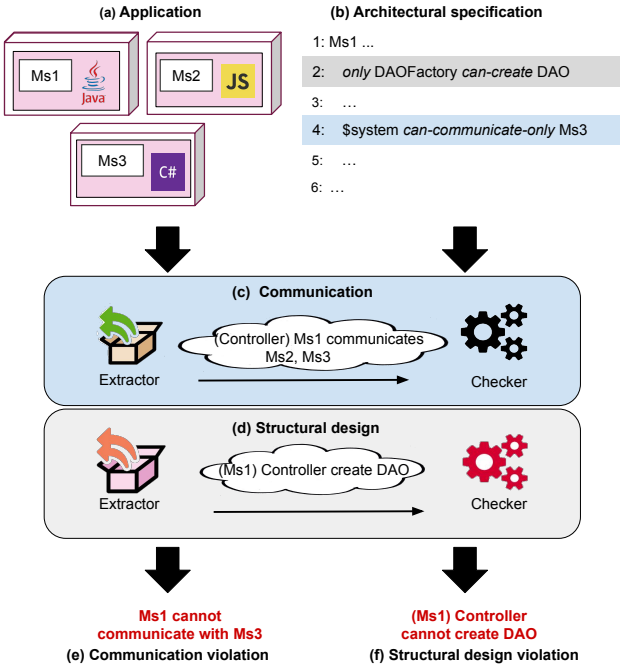


Figure 4: Microservice architecture conformance process.

4.1 Application

As illustrated in Fig. 4a, the process requires access to the source code repository of each microservice.

[7] <hide for blind purposes>

```
1  [id_ms1]: [url_ms1] ; [repository_ms1] ; [pl_ms1]                (a)
2      Constraint DCL #1 from ms1                                   (b)
3      ...                                                          (b)
4      Constraint DCL #N from ms1                                   (b)
5      Constraint DCL+ communication #1 from ms1                    (c)
6      ...                                                          (c)
7      Constraint DCL+ communication #N from ms1                    (c)
8  ...
9  [id_msn]: [url_msn] ; [repository_msn] ; [pl_msn]                (a)
10     Constraint DCL #1 from msn                                   (b)
11     ...                                                          (b)
12     Constraint DCL #N from msn                                   (b)
13     Constraint DCL+ communication #1 from msn                    (c)
14     ...                                                          (c)
15     Constraint DCL+ communication #N from msn                    (c)
```

Listing 1: Architectural Specification our solution.

4.2 Architectural Specification

The process also requires the architectural specification in the DCL$^+$ language (Fig. 4b). As ilustrated in Listing 1, it is a textual file that specifies (a) each microservice, (b) the design constraints of each microservice, and (c) the allowed communications between microservices.

Microservice: Microservices are specified by an *identifier*, its URL, its source code *repository*, and its underlying programming *language*, as shown in lines 1 and 9.

Design constraints: The design constraints (in the DCL language, refer to Section 2.2) of each microservice come indented immediately after the specification of each microservice, as presented in lines 2 to 4 and 10 to 12.

Communication constraints: The communication constraints among microservices (in the DCL$^+$ language, refer to Section 3) come indented immediately after the design constraints, as illustrated in lines 5 to 7 and 13 to 15.

4.3 Communication Conformance

The communication conformance supported by the Communication Extractor and Checker reports a set of *communication violations.*

Communication Extractor: It extracts every communication between microservices. Our current implementation detects communications through the static source code analysis technique by using the Abstract Syntax Tree (AST) for microservices implemented in Java and JavaScript languages (JavaCommExtractor and JsCommExtractor, respectively).

Technically, this module extracts from the source-code of each microservice the invocation of other microservices and stores such information in a HashMap structure. In the context of Java-implemented services, this work is restricted to the identification of the communication between microservices by means of the *@FeignCleint* annotation, specified in the *Spring Cloud* project [19]. With the application of this kind of client, it is possible to identify services' composition by means of coreography and orquestration when all the services, including the orchestrator, are implemented in the Java language. On the other hand, in the

context of microservices created in JavaScript, we developed an extractor module aimed at the *Node.js* framework, considering its vast application in web practices. As an example of the application of the Communication extractor module, Fig. 4c, shows the detection of a communication between the `Controller` module of the `Ms1` microservice with microservices `Ms2` and `Ms3`.

Communication Checker: It verifies whether the extracted communications are in conformance with the architectural specification. In the example from Fig.4b, a constraint defines that `Ms1` cannot communicate with `Ms3`. Since the extractor has detected such communication, this module reports a *communication violation* (see Fig.4e). This module also reports *warnings* when microservices communicate with unknown microservice.

4.4 Structural Design Conformance

The architectural conformance is composed of the Structural Design Extractor and Checker.

Structural Design Extractor: For each microservice, it extracts the structural dependencies in triples `[source-class, dependency-type, target-class]`. Since our process is multi-platform, we have so far implemented extractors for C#[8], Java[9], and JavaScript[10]. In the example from Fig. 4d, it extracts the triple `[CustomerController, create, CustomerDAO]`, which indicates that the `Controller` module from `Ms1` is instantiating DAO objects.

Structural Design Checker: It verifies whether internal dependencies from each microservice—regardless of its language—are in conformance to its planned architecture (i.e., its DCL constraints) using a platform-independent checker[11]. In the example from Fig. 4b, a constraint defines that only `DAOFactory` can create DAO objects. Since the extractor has detected an instantiation at the Controller layer, this module reports a *structural design violation*.

4.5 Limitation

The main limitations of the proposed solution are centered in:

Communication Extractor. They are tied to the capacity of the tool to extract communications between microservices. In its current implementation, the tool is capable of extracting communications through the static analysis of source-code in the context of the libraries `FeignClient` (Java), and `Express` and `Request` (JavaScript). Thus, communications established by methods that are detected only during execution time or using external configurations, or through the usage of other libraries are not be detected, producing false-negatives. On the other hand, there is no occurrence of false-positives, i.e., all the communications detected by the tool are in fact established between microservices.

Structural Design Extractor. There is also the possibility of false-negatives, given that our implementation is not capable of capturing and processing dynamic data, such as methods execute through reflexion. However, we do not report false-positives, i.e., all reported violations are indeed violations.

[8] *CsDepExtractor*, available at <hide for blind purposes>
[9] *JavaDepExtractor*, available at <hide for blind purposes>
[10] JSDepExtractor, available at <hide for blind purposes>
[11] *piDCLcheck*, available at <hide for blind purposes>

We claim that our limitations are acceptable since are related to implementation issues. Since there are several programming languages and frameworks, it is not feasible to implement complete communication and structural design extractors for all of them.

5 METHODOLOGY

This section describe the methodology of seven steps we follow during our evaluation. Our goal is to demonstrate the applicability and usefulness of our proposed solution in a real-world scenario.

Target system choice (Step #1). We sought a proprietary application composed by microservices implemented in different environments, specially programming languages. In other words, a real scenario where our solution must show itself useful. We opt for proprietary systems since, although harder to obtain the system, it is easy to schedule talks and discussion with the chief architect. After the choice of the system, two researchers (the first and second authors of this paper) will conduct the other six steps inside the company during a whole week as follows:

Day 1: *System overview (Step #2).* The chief architect of the subject system will provide an overview of how the system works with focus on its functionalities. Our goal is to obtain an overview of the communication among the underlying microservices. Thus, the researchers will extract the source code architecture as an initial view and will inspect the source code for a better understanding to have a more solid discussion with the chief architect.

Day 2: *Microservices understanding and Constraint specification (Steps #3 and #4).* The chief architect will go deeper in each microservice, its architecture and allowed, forbidden, and required communications. Thus, the researchers themselves could specify the DCL^+ constraints for each microservice.

Day 3: *Constraints validation (Step #5).* The chief architect will revise the DCL^+ constraints created by us based on his/her expertise on the system. Our goal is to obtain a solid and validated set of DCL^+ constraints.

Day 4: *Conformance checking (Step #6).* The researchers will run DCL^+check on the subject system. First, the extractors will obtain every single communication and structural dependency. Next, the verifiers will check the extracted dependencies against the DCL^+ constraints. Our goal is to obtain a report of communication and structural design violations (i.e., divergences and absences).

Day 5: *Feedback (Step #7).* The researchers will present the communication and structural design violations to the chief architect to collect his/her feedback.

6 EVALUATION

During the whole last week of July 2018, the first two authors of this paper evaluated a real-world medium-size complex financial system focused on sales management and conciliation with credit cards, debit cards, tickets, and other means of online payments. Due to a nondisclosure agreement, we omit the company's name in this paper and refer to this system just as Bank Conciliation.

6.1 Application

Bank Conciliation is composed by one orchestrator system (*Node-Middle*) and ten microservices (*Audit*, *Authentication*, *Authorization*, *Conciliation*, *Dashboard*, *Entries*, *FileLoad*, *FileProcess*, *Reports,* and *Summary*). *Node-Middle* is implemented in JavaScript through Node.js and is composed by 27 modules. Other microservices are developed in Java using the Spring Boot framework[12]. It is worth noting that our approach also works on choreography and a controlled evaluation can be found in our previous paper [17].

6.2 Architectural Specification

Listing 2 illustrates a subset of the architectural specification of Bank Conciliation. For a better visualization, this specification shows only constraints that have violations. The complete architectural specification of all microservices and their respective violations are publicly available in our companion website[13].

Microservice: It illustrates the DCL$^+$ specification of all microservices. The first is identified by *Node-Middleware*, URL at `http://localhost:8099`, repository at `/home/microservices/node-middleware/`, and is implemented in JavaScript (line 1). The second is identified by *Audit*, URL at `http://localhost:8085`, repository at `/home/microservices/audit`, and is implemented in Java (line 24). The others microservices are specified alike.

Communication constraints: The communication architecture from this application is based in orchestration. Therefore, only *Node-Middle* communicates with the other microservices. The communication constraints with violations (NM-CC's) are illustrated in lines 15 to 22. For example, **NM-CC3** restricts modules *Assignment, BankStatement, OccurrenceReport, ConciliationReport,* and *FinancialMovements* to communicate only with the *Reports* microservice (line 15); **NM-CC9** limits module *ConciliationSummary* as the only one allowed to communicate with the *Summary* microservice (line 18); **NM-CC10** requires *ConciliationSummary* to communicate with *Summary* (line 19); and **NM-CC22** forbids *ProcessedFiles* to communicate with any microservice (line 22).

Structural Design Constraints: It also illustrates the specification of violated structural design constraints (SC's) in the DCL language for all microservices. For instance, for the *Node-Middle* system, its structural design through **NM-SC8** restricts files from *AuditLog* to depend only on files of its own module and modules *Node_module*, *Config*, and *TransfData* (line 3); For the *Audit* microservice, for example, a constraint **Audit-SC5** restricts classes of the *Main* module to access only classes from *Controller*, *MainAnnotations*, *Logger*, *Apache*, and Java API (line 26).

6.3 Communication Conformance

Communication Extractor: Our solution extracted communications between 27 modules of the *Node-Middle* system and the application's microservices. These modules accomplishes 101 communications with the microservices, ranging from two communications with the *Summary* microservice and 45 with the *Entries* one.

[12]https://projects.spring.io/spring-boot/

[13]https://github.com/anonymoussoftwareengineering/2020_sbcars_microservices/releases

Communication Checker: After extracting the communications, our solution analyzes each communication constraint defined in the architectural specification (lines 15 to 22 of Listing 2) and verifies whether the communications established are allowed or not. Fig. 5 presents the graph of the communication violations detected by our solution. Microservices are represented by circles and modules of *Node-Middle* are represented by squares, where an arrow from a module A to a microservice B indicates that A communicates with B, and its label indicates the number of detected violations (if none, the label is omitted). The squares—which represent modules of the orchestrator system—has an ID inside instead of the name for visualization purposes.

The mapping ID → name we use is reported in the table on the right of the figure. As an example, module `ConcilliationSummary` (identified by number 23) communicates with microservices `Entries` and `Reports`, which both communications are considered violations (divergence), and it also communicates with microservice `Summary`, which is considered a violation (absence). As another example, module `Assignment` (identified by number 13) communicates with microservices `Reports` and `Entries`, which the latter is counted as two violations (divergences).

In the entire application, our solution detected 16 communication violations, 15 caused by divergent and one by an absent communications.

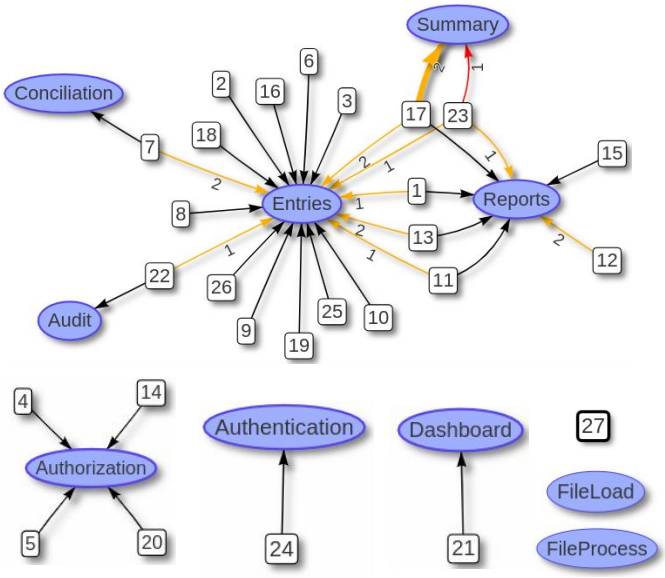
When our solution verified whether the extracted communications are in conformance to the architectural specification, it detected the following *communication* issues:

- *Divergences*:
  (1) Modules *OccurrenceReport* (1), *FinancialMovements* (11), *Assignment* (13), *ConciliationReport* (17), and *AuditLog* (22) established forbidden communications with the *Entries* microservice, violating constraints `NM-CC3` and `NM-CC5`;
  (2) *ConcilationSummary* (23) communicates with *Reports*, violating constraint `NM-CC8`;
  (3) *ConciliationReport* (17) also communicates with the *Summary* microservice, violating constraint `NM-CC9`;
  (4) *ManualConciliation* (7) performs communications that are not provided in the architectural specification with the *Entries* service, which violates constraint `NM-CC18`; and
  (5) Module *ProcessedFiles* (12) communicates with the *Reports* microservice, violating `NM-CC22` constraint.
- *Absence*:
  (1) Module *ConciliationSummary* (23) did not establish expected communications with the *Summary* microservice, violating constraint `NM-CC10`.

According to the architect, the violations occurred due to the lack of knowledge on the part of the developers and also due to the evolution of the application. Although we have not sought for any evidence using surveys or experiments, we believe in the architect' statement since he has been also the project team leader since its beginning. As usual in IT companies, there was no architectural

```
Node–Middleware: http://localhost:8099; /home/microservices/node–middleware/; JavaScript
    #Structural Design Constraints SC's
    AuditLog can-access-only AuditLog, Node_module, Config, TransfData                          "#NM-SC8"[4]
    BankStatement can-access-only BankStatement, Node_module, Config, TransfData               "#NM-SC10"[4]
    ConciliationReport can-access-only ConciliationReport, Node_module, Config, TransfData     "#NM-SC15"[4]
    ConciliationSummary can-access-only ConciliationSummary, Node_module, Config, TransfData   "#NM-SC16"[2]
    Dashboard can-access-only Dashboard, Node_module, Config, TransfData                       "#NM-SC17"[4]
    FinancialMovements can-access-only FinancialMovements, Node_module, Config, TransfData    "#NM-SC21"[4]
    ManualConciliation can-access-only ManualConciliation, Node_module, Config, TransfData     "#NM-SC23"[8]
    OccurrenceReport can-access-only OccurrenceReport, Node_module, Config, TransfData         "#NM-SC25"[6]
    Operator can-access-only Operator, Node_module, Config, TransfData                         "#NM-SC26"[2]

    #Communications Constraints NM-CC's
    Assignment, BankStatement, OccurrenceReport, ConciliationReport, FinancialMovements
       can-communicate-only Reports                                                            "#NM-CC3"[6]
    AuditLog can-communicate-only Audit                                                        "#NM-CC5"[1]
    ConciliationSummary can-communicate-only Summary                                           "#NM-CC8"[3]
    only ConciliationSummary can-communicate Summary                                           "#NM-CC9"[2]
    ConciliationSummary must-communicate Summary                                               "#NM-CC10"[2]
    ManualConciliation can-communicate-only Conciliation                                       "#NM-CC18"[2]
    ProcessedFiles cannot-communicate Audit, Authentication, Authorization, Conciliation,
       Entries, Reports, Dashboard, Summary, FileProcess, FileLoad                             "#NM-CC22"[2]

Audit: http://localhost:8085; /home/microservices/audit/; Java
    only Service can-depend Jndi                                                               "#Audit-SC3"[3]
    Main can-access-only Controller, MainAnnotations, Logger, Apache, API                      "#Audit-SC5"[4]

Authentication: http://localhost:8082; /home/microservices/authentication/; Java
    only Service can-depend MyBatis                                                            "#Authentication-SC5"[2]
    Service must-useannotation ServiceAnnotation                                               "#Authentication-SC9"[2]

Authorization: http://localhost:8083; /home/microservices/authorization/; Java
    only Service can-depend Jndi                                                               "#Authorization-SC3"[6]
    Main can-depend-only API, MainAnnotations, Apache                                          "#Authorization-SC5"[5]

Conciliation: http://localhost:8093; /home/microservices/conciliation/; Java
    Util can-depend-only Util, API, Logger                                                     "#Conciliation-SC7"[24]

Dashboard: http://localhost:8095; /home/microservices/dashboard/; Java

Entries: http://localhost:8092; /home/microservices/entries/; Java
    only Main can-depend BCMultitenancy                                                        "#Entries-SC4"[3]
    Util can-depend-only Util, API                                                             "#Entries-SC5"[2]
    Controller cannot-access DAO                                                               "#Entries-SC7"[6]
    Model, Serializer must-implement Serializable                                              "#Entries-SC11"[11]
    Service must-useannotation ServiceAnnotation                                               "#Entries-SC9"[7]
    Controller must-useannotation CtlrAnnotations                                              "#Entries-SC8"[2]

FileLoad: http://localhost:8075; /home/microservices/fileLoad/; Java
    Main cannot-depend DAO, Model, BCDAO, JPA                                                  "#FileLoad-SC6"[4]
    Model must-implement Serializable                                                          "#FileLoad-SC10"[6]

FileProcess: http://localhost:8076; /home/microservices/fileProcess/; Java
    only DAO, Model can-depend JPA                                                             "#FileProcess-SC4"[8]
    Util can-depend-only Util, BCUtil, API, Logger                                             "#FileProcess-SC6"[22]
    Model must-implement Serializable                                                          "#FileProcess-SC11"[1]

Reports: http://localhost:8094; /home/microservices/reports/; Java
    Controller must-useannotation CtlrAnnotations                                              "#Reports-SC7"[1]
    Controller must-extend BCController                                                        "#Reports-SC8"[1]
    Serializer must-depend JSONSerializer                                                      "#Reports-SC10"[4]

Summary: http://localhost:8096; /home/microservices/summary/; Java
    Service must-useannotation ServiceAnnotation                                               "#Summary-SC7"[1]
    Controller must-useannotation CtlrAnnotations                                              "#Summary-SC8"[1]
```

Listing 2: Subset of the architectural specification of Bank Conciliation.



ID	Name	ID	Name	ID	Name
1	OccurrenceReport	10	FileUpload	19	BankAccounts
2	Categories	11	FinancialMovements	20	Client
3	EstablishmentGroups	12	ProccedFiles	21	Dashboad
4	Operator	13	Assignment	22	AuditLog
5	AccesProfile	14	Home	23	ConciliationSummary
6	Task	15	BankStatement	24	User
7	ManualConciliation	16	Establishment	25	AttributesSet
8	ConciliationPlan	17	ConciliationReport	26	Cities
9	States	18	OccurrenceReason	27	Tregrid

Figure 5: Communication violations of Bank Conciliation.

artifacts for developers, but only transferal of knowledge from another project team member. Divergences 1 and 4 occurred due to the lack of knowledge of the developers regarding the restriction that each module in the *Node-Middle* system could only communicate with microservices. In other words, modules *OccurrenceReport, FinancialMovements, Assignment, ConciliationReport*, and *ManualConciliation* can establish communications with the *Reports* and *Conciliation* microservices, but not with the *Entries* service. Similarly, divergence 5 refers again to the lack of knowledge of the developers since module *ProcessedFile*—which should keep a history of the processed files and it cannot communicate with any microservice—communicates with the *Reports* service.

Divergences 2 and 3 and the absence detected occurred due to the evolution of the application from the *Reports* to the *Summary* microservice. Consequently, the *ConciliationReport* module has being moved to *ConcilationSummary*. This evolution justifies the detected violations since the complete evolution of the system is taking place in the next releases.

It is worth noting that the *ConciliationReport* module has the highest number of detected violations, being responsible for 26,6% of the divergences (four out of 15). Next, modules *ConciliationSummary, ProcessedFiles, Assignment*, and *ManualConciliation* represent each 13,3% of the divergences. Modules *AuditLog*, *FinancialMovements*, and *OccurrenceReport* represent each 6,66% of the divergences (only one each). With regards to the absence of communication, module *ConciliationSummary* is fully responsible for this violation type. In total, among the 27 specified modules, violations occurred in only eight, corresponding to 29,6% of the modules. These results show that, although there was no previous specification of the application, the team members showed themselves committed to conducting the development following the standards defined by the architect.

6.4 Structural Design Conformance

Structural Design Extractor: Our solution successfully ran JSDepExtractor and JavaDepExtractor to extract the structural dependencies of the *Node-Middle* system and ten microservices, respectively.

Structural Design Checker: When our solution applied piDCLcheck to verify whether internal dependencies from each microservice were in conformance to their planned architecture (in DCL constraints) or not, it detected structural design violations, which the number *n* of respective violations is indicated in front of the constraint in Listing 2 as `[n]`. As it can be observed, our solution detected 162 structural design violations of the application's microservices, more specifically 125 divergences and 37 absences. In general, these violations were caused due to the lack of knowledge on the part of the developers regarding the concepts of software architecture, specifically in the correct application of the *Spring MVC* framework and the expected dependencies of the *Util* modules.

Regarding the unknown dependencies to the *Spring MVC* framework, our solution detected 45 divergences and five absences which represent 36% and 10.8% of the violations, respectively. For violations detected through dependency implementations for the *Util* module, our solution reported 48 divergences, corresponding to 38.4% of those violations. Listing 2 reports that the *Entries* and *FileProcess* services are responsible for the largest number of violations (31) respectively, representing 38.2% of the total number of violations. According to the architect, this is possibly due to the fact that this service is the largest—in number of lines of code—among the others. In contrast, the *Dashboard* microservice have not had any structural design violation.

6.5 Threats to Validity

At least three threats may affect the validity of our findings:

First, one could argue that the two researchers specified the DCL$^+$ constraints for the evaluated system. Nevertheless, they worked together to specify what the chief architect described in natural language. Moreover, the chief architect validated not only the constraints but the violations detected by them.

Second, one could argue that there could be false negatives due to limitations of our current implementation. Although it could indeed occur for structural design checking, we ensure—by manual inspection—that all communications between microservices have been considered. During such inspection, we detected some missing communications and implemented support for Request (JavaScript).

Third, although we cannot extrapolate our results to other systems (external validity), we argue that our evaluation was conducted in a real development scenario.

7 RELATED WORK

The microservices architecture is a new and underexplored research area, being most proprietary applications and therefore not easily accessible to the academic community [1, 27]. Since currently there is a small amount of work that seeks solutions for conformance verification in the microservices architecture, this section also discusses studies on the concepts related to microservices usage and benchmarks for application evaluation.

Microservices usage: Alshuqayran et al. [3] point out the main research interests in microservices are establishment of communication, integration and composition of microservices, deployment, architectural discovery, performance, security, APIs, and containers technologies.

Francesco et al. [9] argue that most recurring problems in this architectural style are relate to complexity, flexibility, management, and composition. In a nutshell, the good level of flexibility this style brings (with low service coupling and greater maintainability), come together with greater complexity mainly because it implies in a high number of distributed services to operate.

Granchelli et al. [10, 11] propose an architectural discovery approach to address complexity problems of such architectural style. Basically, using the MicroART tool, the authors come up with a logical architecture model from the physical one, which is reviewed manually by the architects. Our work could complement this one by enforcing architect reviews through DCL$^+$ contraints.

Mayer and Weinreich [13] also propose a discovery approach to identify microservices using static and dynamic analysis, which is very similar to the communication extractor we use as part of our solution. Although our extractor does not rely on dynamic analysis, our proposed solution is multiplatform (theirs is Java only) and our evaluation is stronger (theirs is in a controlled environment).

In another research line, Engel et al. [6] propose an approach to evaluate the architecture of microservices-based systems. Again, they developed a communication extractor based on dynamic analysis only called MAAT (Microservice Architecture Analysis Tool). In short, the authors calculate metrics on the extracted communications to assess several design principles microservices development should follow. Our solution has a more flexible scope since the constraints can be used in a generic way to assess any design principles, not only pre-defined ones.

Benchmarks: Although the solution proposed in this paper was evaluated in a real scenario, academic research in microservice architecture is still limited, in part due to the lack of reference applications that reflect the characteristics of microservice applications developed in Industry [27]. There are at least three studies that aim to solve this gap. Aderaldo et al. [1] evaluated four benchmarks and—based on twelve requirements for microservices architectures—recommending the Socks Shop benchmark because it satisfies 10 out of twelve requirements.

Zhou et al. [27] evaluated seven benchmarks on microservices and identified four main issues: (i) misuse of different modes of interaction as mechanisms of synchronous and asynchronous communication, (ii) the small number of microservices present in applications, (iii) the inadequacy of benchmarks to the principles of this architectural style, not being structured around their respective business capacity, and (iv) the insufficient generation of unit test cases or cases of integration test. Thereupon, the authors developed TrainTicket, a benchmark for the context of rail ticket sales, which contains 24 Spring Boot or Node.js-based microservices.

Architecture conformance: So far, only one work has been found that provides a conformance check on microservices architectures. Although it does not have a broader scope like the proposed in this paper, it focuses on the conformance to microservices patterns [20]. Zdun et al. [26] suggested a set of constraints to check and metrics to assess architecture conformance to those microservice patterns. The authors argue that conformance to those patterns is hard to ensure and assess automatically, leading to problems such as architectural drift and erosion, especially in the context of continued software evolution or large-scale microservice systems. Thus, they suggest a set of constraints to check and metrics to assess architecture conformance to microservice patterns. In comparison to expert judgment derived from the patterns, a subset of these constraints and metrics shows a good relative performance and potential for automation.

8 FINAL REMARKS

We proposed and evaluated an architectural conformance process for the microservice architecture. Given a microservice-based system and its architectural specification, the process extracts and checks the communications between microservices and the structural design of each microservice. As a result, the process points out *communication* violations between the microservices and *structural design* violations of each microservice. As our *main contribution for the state-of-the-art* is our solution formally restricts communications (e.g., over HTTP) between different systems, which none of the existing architectural conformance solutions does.

We applied the proposed process in a real-world system composed of eleven microservices developed in two different programming languages (JavaScript and Java). This evaluation consists of our *main contribution for the state-of-the-practice* since so far there are no studies conducting a case study from the communication

point of view. As the result, our process reported a set of 16 communication violations and 162 structural design violations. In general, such violations occurred due to the lack of knowledge by the developers about communication restrictions between the orchestrator system modules and other microservices, as well as the evolution of two microservices in the analyzed version of the application. These violations were caused by only eight modules out of 27 specified modules (29.6%). Regarding the project's architectural specifications of each microservice, our solution detected 162 structural design violations of the application's microservices with 125 divergences and 37 absences. In general, these violations were caused due to the lack of knowledge on the part of the developers regarding the concepts of software architecture, specifically in the correct application of the `Spring MVC` and expected dependencies of module *Util*. These results show that although there was no prior specification of the application, team members were mostly committed to conduct development following the standards defined by the architect.

Last but not least, we conclude that the our proposed solution was effectively able to detect deviations in the communication architecture and the architectural design of a real-world microservice-based system written in two different programming languages, which until then were unknown by its architects. Besides restricting communication, our solution was designed it as multi-platform and also included local structural design conformance to promote better applicability in real-world scenarios. This is one of our *contributions aiming to reduce the gap between Industry and Academy*.

Further work includes (i) the development of communication extractors and structural design extractors to other commonly-used languages in microservices' development, such as C#, PHP, and Python; (ii) the evaluation of the our process in other real-world ecosystems; and (iii) the investigation of the cost and effort to maintain DCL constraints during system evolution.

ACKNOWLEDGMENTS

This work is partially supported by INES (www.ines.org.br), CNPq grants 465614/2014-0 and 305829/2018-1, FAPEMIG grant APQ-03513-18, FACEPE grants APQ-0399-1.03/17 and APQ/0388-1.03/14, and CAPES grant 88887.136410/2017-00.

REFERENCES

[1] Carlos M. Aderaldo, Nabor C. Mendonça, Claus Pahl, and Pooyan Jamshidi. 2017. Benchmark requirements for microservices architecture research. In *1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*. 8–13.

[2] Sascha Alpers, Christoph Becker, Andreas Oberweis, and Thomas Schuster. 2015. Microservice based tool support for business process modelling. In *19th International Enterprise Distributed Object Computing Workshop (EDOCW)*. 71–78.

[3] Nuha Alshuqayran, Nour Ali, and Roger Evans. 2016. A systematic mapping study in microservice architecture. In *9th International Conference on Service-Oriented Computing and Applications (SOCA)*. 44–51.

[4] Jens Borchers. 2011. Invited Talk: Reengineering from a Practitioner's View – A Personal Lesson's Learned Assessment. In *15th European Conference on Software Maintenance and Reengineering (CSMR)*. 1–2. https://doi.org/10.1109/CSMR.2011.63

[5] Jan Bosch. 2004. Software Architecture: The Next Step. In *First European Workshop (EWSA)*. 194–199. https://doi.org/10.1007/978-3-540-24769-2_14

[6] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. 2018. Evaluation of Microservice Architectures: A Metric and Tool-Based Approach. In *30th International Conference on Advanced Information Systems Engineering (CAiSE)*. 74–89.

[7] Martin Fowler. 2014. Microservices Resource Guide. https://martinfowler.com/microservices/.

[8] Martin Fowler and James Lewis. 2014. Microservices: a definition of this new architectural term. Disponível em: https://martinfowler.com/articles/microservices.html.

[9] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. 2017. Research on architecting microservices: trends, focus, and potential for industrial adoption. In *4rd International Conference on Software Architecture (ICSA)*. 21–30.

[10] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. 2017. MicroART: A Software Architecture Recovery Tool for Maintaining Microservice-based Systems. In *1st International Conference on Software Architecture (ICSA)*. 298–302.

[11] Giona Granchelli, Mario Cardarelli, Paolo Di Francesco, Ivano Malavolta, Ludovico Iovino, and Amleto Di Salle. 2017. Towards Recovering the Software Architecture of Microservice-Based Systems. In *1st International Conference on Software Architecture Workshops (ICSAW)*. 46–53.

[12] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. 2008. Architecture Compliance Checking - Experiences from Successful Technology Transfer to Industry. In *12th European Conference on Software Maintenance and Reengineering (CSMR)*. 43–52. https://doi.org/10.1109/CSMR.2008.4493299

[13] Benjamin Mayer and Rainer Weinreich. 2018. An Approach to Extract the Architecture of Microservice-Based Software Systems. In *12th Symposium on Service-Oriented System Engineering (SOSE)*. 21–30.

[14] Gail Murphy, David Notkin, and Kevin Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. In *3rd Symposium on Foundations of Software Engineering (FSE)*. 18–28. https://doi.org/10.1145/222124.222136

[15] Gail C. Murphy, David Notkin, and Kevin Sullivan. 1995. Software reflexion models: Bridging the gap between source and high-level models. In *3rd International Symposium on Foundations of Software Engineering (FSE)*. 18–28.

[16] Sam Newman. 2015. *Building microservices: designing fine-grained systems* (1 ed.). O Reilly Media.

[17] <omitted>. <omitted>. <omitted>. In *Workshop on Software Visualization, Evolution and Maintenance (VEM)*. <omitted>.

[18] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. 2010. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software* 27, 5 (2010), 82–89.

[19] Pivotal. 2018. Spring Cloud Netflix. https://cloud.spring.io/spring-cloud-netflix.

[20] Chris Richardson. 2017. A pattern language for microservices. http://microservices.io/patterns/index.html.

[21] Santonu Sarkar, Shubha Ramachandran, G. Sathish Kumar, Madhu K. Iyengar, K. Rangarajan, and Saravanan Sivagnanam. 2009. Modularization of a Large-Scale Business Application: A Case Study. *IEEE Software* 26 (2009), 28–35. https://doi.org/10.1109/MS.2009.42

[22] Ricardo Terra and Marco Tulio Valente. 2008. Verificação Estática de Arquiteturas de Software utilizando Restrições de Dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*. 24–37.

[23] Ricardo Terra and Marco Tulio Valente. 2009. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience* 39, 12 (2009), 1073–1094.

[24] Ricardo Terra, Marco Tulio Valente, Krzysztof Czarnecki, and Roberto S. Bigonha. 2015. A Recommendation System for Repairing Violations Detected by Static Architecture Conformance Checking. *Software: Practice and Experience* 45, 3 (2015), 315–342. https://doi.org/10.1002/spe.2228

[25] Johannes Thönes. 2015. Microservices. *IEEE Software* 32, 1 (2015), 116–126.

[26] Uwe Zdun, Elena Navarro, and Frank Leymann. 2017. Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns. In *15th International Conference on Service-Oriented Computing (ICSOC)*. 411–429.

[27] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Poster: Benchmarking Microservice Systems for Software Engineering Research. In *40th International Conference on Software Engineering (ICSE)*. 323–324.