Apolinário and França *Journal of the Brazilian Computer Society* (2021) 27:17
https://doi.org/10.1186/s13173-021-00120-y

Journal of the
Brazilian Computer Society

RESEARCH Open Access

A method for monitoring the coupling evolution of microservice-based architectures

Daniel R.F. Apolinário[1,2*] and Breno B.N. de França[2]

*Correspondence:
drfapolinario@gmail.com
[1]Embrapa Digital Agriculture, Brazilian Agricultural Research Company, Campinas, Brazil
[2]Institute of Computing, State University of Campinas, Campinas, Brazil

Abstract

The microservice architecture is claimed to satisfy ongoing software development demands, such as resilience, flexibility, and velocity. However, developing applications based on microservices also brings some drawbacks, such as the increased software operational complexity. Recent studies have also pointed out the lack of methods to prevent problems related to the maintainability of these solutions. Disregarding established design principles during the software evolution may lead to the so-called architectural erosion, which can end up in a condition of unfeasible maintenance. As microservices can be considered a new architecture style, there are few initiatives to monitoring the evolution of software microservice-based architectures. In this paper, we introduce the SYMBIOTE method for monitoring the coupling evolution of microservice-based systems. More specifically, this method collects coupling metrics during runtime (staging or production environments) and monitors them throughout software evolution. The longitudinal analysis of the collected measures allows detecting an upward trend in coupling metrics that could represent signs of architectural degradation. To develop the proposed method, we performed an experimental analysis of the coupling metrics behavior using artificially generated data. The results of these experiment revealed the metrics behavior in different scenarios, providing insights to develop the analysis method for the identification of architectural degradation. We evaluated the SYMBIOTE method in a real-case open source project called Spinnaker. The results obtained in this evaluation show the relationship between architectural changes and upward trends in coupling metrics for most of the analyzed release intervals. Therefore, the first version of SYMBIOTE has shown potential to detect signs of architectural degradation during the evolution of microservice-based architectures.

Keywords: Microservices, Maintainability, Coupling metrics, Software evolution, Software architecture, Software engineering

Introduction

Companies have established the use of the microservice architectural style in software development. Scaling agile processes and continuous deployment are among the primary motivations for its use [1]. Microservices have as their main characteristic the breaking down of the software into small and independently deployable services that communicate through lightweight mechanisms [2].

Despite the expected benefits of adopting microservices, this architectural style poses new challenges for the evolution of systems. Bogner et al. [3] mention that some features of microservices such as technological heterogeneity and decentralized control may harm the maintainability of an application when not handled properly. For instance, when a team responsible for a microservice makes architectural decisions, the lack of a holistic view of the system may cause bad decisions w.r.t. the integration between microservices. Throughout the system evolution, it becomes more complex to understand its codebase and operational environment, which can lead developers to introduce modifications that damage the architectural integrity of the system [4]. The lack of knowledge related to code or tools in your environment can lead, for example, a developer to create an unnecessary dependency with another service. It is also necessary to consider the architectural technical debt that can occur even with experienced teams due to external pressures such as tight deadlines. Over time, the accumulation of architectural issues can cause architectural degradation [4].

Ideally, microservice-based applications (MSAs) should prevent architectural degradation based on its characteristics, such as the physical boundaries between services that inhibit development teams from making convenient changes causing system architecture violation. Other features of MSAs, such as flexibility, scalability, fewer dependency problems, and separation of concerns, also promote maintainability [3]. Those characteristics explain the claim that MSAs are resilient to architectural degradation [5]. However, Lehman's 7th software evolution law states that if no deliberate action is taken, there is a tendency for internal quality to degrade. As the number of microservices increases, the interactions among them tend to increase significantly, causing difficulties in monitoring the system as a whole [6], and its evolution. It calls for methods and tools to support monitoring the evolution of this type of architecture.

Due to the complexity involving distributed systems, maintaining an overview of these applications is challenging [7]. Existent monitoring technologies such as tracing, logs, operational metrics, and alerts [8] are useful for development teams, but there is a shortage of specific methods and tools to monitor architecture problems that can impact the maintainability of MSAs.

Therefore, we developed the SYMBIOTE method to monitor the architectural evolution of MSAs. It uses service coupling metrics to warn developers and architects when successive changes can negatively affect the system maintainability. We aim to identify microservices coupling issues throughout the software evolution, answering the research question: "Is the continuous monitoring of coupling metrics from microservice-based applications able to indicate architectural degradation?" To answer this, we understand it is crucial to distinguish regular increasing coupling from harmful degradation.

This paper is an extended version of the work that presents SYMBIOTE [9]. We additionally evaluated the SYMBIOTE method in a real application called Spinnaker (the "Method evaluation" section). We use the same set of metrics tested in the previous exper-

iment. We observed the trend in metrics of the real case is slightly different from the results of the experiment, which is a sign that the experiment did not cover all the variety of architectures and evolution that can happen. However, we identified a relationship between trends in metrics and the architectural changes of the application, which means the method is able to detect scenarios when architectural changes may cause architecture degradation.

The remainder of this paper is organized as follows. The "Background" section describes the background related to microservices and metrics maintainability. In the "Related work" section, we report the closest related works. The "Research method" section describes the research method. In the "Analysis of service coupling metrics" section, we report the experiment used to develop the analysis of proposed method. In the "The SYMBIOTE method" section, we present the proposed solution. The "Conclusions" section presents the conclusions and future work.

Background

Microservices have been a trend not only for modern new applications but also in the evolution (re-engineering) of monolithic systems. Jamshidi et al. [8] analyze several factors that have contributed to the emergence and subsequent growth of this architectural style. Among the main factors, the authors mention the growing demand for scalability and new practices for software development, such as continuous integration and continuous deployment, containerization, and cloud technologies. In this context, constant software changes may impact software architecture.

According to Lehman's 7th law, there is a tendency for software quality decay during its evolution. The loss of architectural quality is one of the most significant in the system's maintainability. The architectural decay of software is the phenomenon "1) when concrete (as-built) architecture of a software system deviates from its conceptual (as-planned) architecture where it no longer satisfies the key quality attributes that led to its construction or 2) *when the architecture of software system allows no more changes to it due to changes introduced in the system over time and renders it unmaintainable*" [10]. The second part of this definition is a motivation for our work since the coupling is a relevant aspect regarding well-established design principles and maintainability [11].

Lindvall et al. [12] argue that architectural degradation occurs when constant software changes can negatively impact its structural complexity. Architects should observe relevant maintenance aspects to avoid architectural degradation. Specifically for microservice architectures, a "systematic understanding of maintainability as well as metrics to automatically quantify its degrees" are essentials [13]. Systems built on this architecture are flexible, but managing constraints and dependencies between services is challenging [14].

Regarding metrics, Perepletchikov et al. [15] show that existing maintainability metrics for procedural and OO software are not well-suited for service-oriented systems. Recently, specific metrics and models have been introduced to measure the maintainability of service-oriented systems and microservices [16] [17]. Our work will focus on coupling metrics as they are directly related to system architecture and represent an important aspect when managing maintainability [11] [18]. Bogner et al. [17] list several maintainability metrics for service and microservice-based applications. We present a subset of the coupling metrics belonging to the maintainability metrics for Service-based Systems (SBSs) cataloged in [17].

The following metrics should be collected per individual service:

- *Absolute Importance of the Service (AIS)*: number of consumers invoking at least one operation from a service S1. The higher the AIS, the more important the service S1 is within the system. Average AIS can be useful for identifying and quantifying the most critical services.
- *Absolute Dependence of the Service (ADS)*: number of services on which the S1 service depends. In other words, ADS is the number of services that S1 calls for its operation to be complete. The higher the ADS, the more this service depends on other services, i.e., it is more vulnerable to the side effects of failures in the services invoked.

The following metrics work for the entire application:

- *Service Coupling Factor (SCF)*: measure of the density of a graph's connectivity. $SCF = SC/(N^2 - N)$, where *SC (service coupling)* is a sum of all dependencies between services. That is, each service that can invoke operations from another service adds one more to this value. N is the total number of services. If we represent dependencies as a graph, SC is the sum of all edges and $N^2 - N$ represents the maximum oriented edges the graph can have.
- *Average number of directly connected services (ADCS)*: the average of ADS metric of all services.

Moreover, metrics presenting the following criteria were not considered for the analysis of coupling evolution in our work:

1. Derived, repeated, or similar metrics tend to present similar variability and trends to their primary metrics. For instance, *Relative Coupling of Service (RCS)* and *Relative Importance of Service (RIS)* metrics are not considered as they will present a similar behavior to *ADS* and *AIS* metrics. The *System's Service Coupling (SSC)* metric measures edge density in the dependency graph (similar to the SCF metric). The SCF metric is repeated in two different research literature sources.
2. Metrics whose definitions are inconsistent were discarded. For instance, the *Absolute Criticality of the Service (ACS)* metric is defined as the product of two other metrics (ADS and AIS). In this case, if a service presents ADS equals zero (no outgoing edges), then ACS will be zero (no coupling) also regardless of the AIS value. We consider it offers a misleading regarding the coupling meaning. Thus, we decided not to use this metric to avoid misinterpretation in the analysis.
3. Metrics of cyclical dependency like *Services Interdependence in the System (SIY)* are not used because the mere existence of a single dependency of this type already constitutes an architectural smell. Therefore, it makes no sense to analyze the variability or trend of a metric for which we know that the only acceptable value would be zero.
4. Metrics based on internal elements of a service (classes, packages, operations, interfaces), as well as those that use weight to differentiate different types of connections between these elements will not be considered. Our interest in this initial work is to capture dependencies only at the logical service level and treat them with the same weight. This is the case with the metrics *WISCE*, *WESICE*,

WESOCE, *ESICSI*, *EESIOC*, *SIIEC*, *SPARF*, and *SPURF* (metrics originally defined in [16]).

Related work

Some related work focus on maintainability metrics for SOA-based systems, for instance, proposing coupling metrics [16], and reviewing the literature on SOA metrics and microservices [17]. Metrics presented in these works lack empirical validation.

De Toledo et al. [19] created a catalog of Architectural Technical Debt (ATD) related to microservice-based architectures, as well as impacts and solutions for each one of them. The authors conducted cross-company interviews with professionals to prepare this ATD map. This study aimed to contribute to the management of ATDs in software development. This study provides some evidence that multiple debts increase the probability of coupling problems between microservices.

To support the evolution of systems based on microservices, Sampaio [20] proposes a service evolution model. This model aggregates structural, deployment, and execution information of MSA, but there are no further details regarding the data extraction. Similarly, Mayer et al. [21] developed an approach to extract architectural information (static, infrastructure, and runtime) from MSA. In this last work, the authors propose both static and dynamic strategies to capture architecture-relevant information. In both works, authors created a small test scenario to verify their proposal feasibility.

Kitajima et al. [22] propose a method to extract information from calls between services to infer their relationship. The authors use a dynamic approach to gathering information from the running system. The evaluation of the method used a small test application (not a real system).

Other works regard the architecture conformance with patterns or standards. ArchCI [23] is an integrated CI tool to monitor architectural drift (deviation between concrete and planned) for each deployment in the application integration pipeline. The idea is to check architectural compliance through a Dependency Constraint Language. In a different approach, Ntentos et al. [24] propose an assessment of architecture compliance based on microservices with well-established coupling patterns. They propose metrics that represent adherence to patterns. The objective of this work was to evaluate the feasibility of building a method to measure compliance with standards.

Some works focus on coupling evolution. Sousa et al. [18] present an exploratory study observing the coupling behavior throughout the evolution of open-source, object-oriented software. This work is related to ours concerning the time series analysis of historic version and the use of coupling metrics. However, our work concerns the detection of architectural problems while the related is interested in establishing coupling evolution properties.

Jenkins and Kirk [25] used a software component instability metric to analyze the evolution of software architecture using complex network theory. The instability metric is based on coupling. This work is related to ours in some aspects, but it focuses on showing similarities between the behavior of software graphs with complex networks as well as predicting the software maintainability, which differs from our goal that is to evaluate the current architecture for detect potential issues.

The most related work is the GMAT [26], which proposes a tool to obtain the dependency graph of an MSA. This tool implements the static analysis approach to gather

information about dependencies. Its main concern is to offer a graphic visualization of the dependencies between the services to support the monitoring of the evolution of the software. This tool is not evaluated in a real case. Our work differs mainly in two aspects: (1) our goal is to detect indications of architectural deterioration whereas GMAT only generates the dependencies graphs; (2) as GMAT relies on static analysis, it is limited to several technological constraints such as the use of *Java* language, *Spring Boot Actuator*, *Spring Feign*, and *springfox-swagger2*. The capability to accurately identify calls between services using static declarations is also limited.

Our work uses coupling metrics to evaluate the structure of the architecture over the continuous software evolution. Therefore, it differs from these mainly because it proposes a dynamic analysis approach to collect the dependencies between services, analyze trends in the evolution data series, and inform architects when the trends indicate signs of architectural degradation; it is also evaluated in a real project.

Research method

Our research goal is to monitor the coupling evolution of MSA using metrics, assuming it can support software engineers on improving software maintainability.

For that, we concentrate on three objectives:

- Define a strategy for collecting the selected coupling metrics from MSAs based on its concrete (deployed) architecture
- Evaluate the behavior of existing selected metrics over time, considering the structural characteristics of MSAs
- Create a method to continuously analyze the evolution of the coupling metrics between services
- Evaluate the proposed method in a real-case application

To achieve them, we defined the following activities:

Literature review: we performed an *ad hoc* literature review on software maintenance metrics for service and microservice-based architectures. We identified two recent studies compiling metrics for services and microservices [16, 17].

Selecting maintainability metrics: Bogner et al. [17] present a literature review on maintainability metrics for service-oriented and microservice-based systems, but the identified ones lack empirical validation. From this review, we selected a subset of coupling metrics (see the "Background" section) to be used by the proposed method.

Defining a strategy for metrics collection: Firstly, we analyzed which software information we needed to collect. For the selected coupling metrics, we needed to identify service dependencies. Afterward, we decided on using the static or dynamic analysis to gather the dependencies between the services. Next, we defined the collection strategy and searched for existing tools to support it.

Develop the method for monitoring the coupling evolution of MSA: We designed an experiment (details in the "Analysis of service coupling metrics" section) that simulates the MSA evolution (represented as dependency graphs). The goal of this experiment is to characterize the behavior of the metrics in face of different changes over time. For that,

we use artificially generated data as it allows us to verify their behavior under different conditions, such as application size and structure.

Evaluate the proposed method in a real-case MSA: After developing the method, we design a study in retrospect based on a real case (open source software), so that we can characterize the method's effectiveness on identifying meaningful changes on the application coupling over time.

Although we recognize the existence of publicly available MSA repositories, some hindrances cannot be disregarded. Zhou et al. report the gap of benchmark systems reflecting the characteristics of real microservice systems [27]. In [28], the authors create a dataset of MSA open source projects. Most of the applications in this dataset are demo or toy projects. Furthermore, a suitable case for evaluating the proposed method demands (i) to be a real case; (ii) to have at least 10 *stable* releases to be able to detect trends and variations in metrics, besides, unstable releases do not allow to perform dynamic analysis; (iii) being an active project, as our experience (SiteWhere, Spinnaker, OpenEBS, Lelylan, Magda, and others) shows the deployment of this kind of system can become unfeasible without minimum support.

Analysis of service coupling metrics

Analyzing metrics individually per microservice could lead to misunderstandings about the evolution of the complete system. This way, we use the Gini coefficient to analyze both the ADS and AIS metrics (described in the "Background" section).

The Gini coefficient is currently widely used to measure the distribution of wealth in the field of Economics. This index has the advantage of working with a [0;1] interval regardless of the statistical distribution of the data. Its value reveals how unequal the values of the coupling metrics (ADS and AIS) are among microservices in the same application. Thus, it allows us to observe if few microservices concentrate coupling. For example, the *Single Responsibility Principle* is an important design principle for microservices, in which one service has one single responsibility. A Gini coefficient with a higher value may indicate a possible violation of the *Single Responsibility Principle*, as there must be a small number of services concentrating incoming or outgoing calls (logical coupling). There are several applications of the Gini coefficient in the literature for the software evolution analysis [29, 30]. However, none of these works applies it in the context of microservices. We calculated the Gini coefficient G as defined in [31] (1).

$$G = \frac{\sum_{i=1}^{n}(2i - n - 1)x_i}{n\sum_{i=1}^{n} x_i} \tag{1}$$

where the values $x_i, x_{i+1}, ...x_n$ are ordered, n is the number of values to be computed, and i represents the rank of the value x. G assumes values between 0 and 1, in which the value 0 indicates perfect equality, whereas values closer to 1 indicate more inequality among the observations. It is important to emphasize that the Gini index cannot quantify whether the observations have high or low mean values because it just measures how distributed the values are. That is, the Gini index will be equal to zero if all observations have maximum values and also if they have minimum values, since all services in each one have the same values. Therefore, the metrics based on the Gini index support the analysis of coupling distribution, but it can be inconclusive to assess whether the average level of coupling between services of an application is high or low. Finally, the derived metrics

presented in the following will be analyzed along with the *SCF* and *ADCS* metrics (both explained in the "Background" section), which are more suitable to assess the average level of coupling:

- *Gini coefficient for AIS*: calculated using the individual AIS measures for each microservice in a given release. That is, this coefficient indicates how the importance of services is distributed among themselves. Values close to zero mean an even distribution of importance among the microservices. Otherwise, values close to one mean the importance are very concentrated in a few services. To simplify, we call this metric *Service Importance Distribution (SID)*.
- *Gini coefficient for ADS*: calculated using the individual ADS measures for each microservice in a given release. That is, this coefficient indicates how balanced are dependencies among services. When close to zero, it represents evenly distributed dependencies among the microservices. Otherwise, values close to one mean that few services concentrate many dependencies. To simplify, we call this metric *Service Dependency Distribution (SDD)*.

We designed an experiment for testing the behavior of the metrics in different scenarios. We used artificially generated dependency graphs representing microservice architectures, as it is unfeasible to have real MSAs representing all the verified conditions. The generated data are directed graphs, in which nodes represent the microservices and edges represent the dependencies (incoming and outgoing). We developed a tool [32] to generate and evolve them.

Goal and hypotheses

This study aims *to analyze* four coupling metrics, *for the purpose of* characterizing *with respect to* their behavior over time, *in the context of* artificially generated dependency graphs representing MSA releases. For each metric, we test the following hypotheses:

H0: There is not a significant difference in trends in the evolution of the one SID/SDD/ADCS/SCF metric between introducing and removing an architecture smell throughout releases of an MSA.

H1: There is a significant difference in trends in the evolution of the one SID/SDD/ADCS/SCF metric between introducing and removing an architecture smell throughout releases of an MSA.

Experimental design

We adopted a full factorial design. Table 1 shows the scenarios resulting from the combinations between the two factors (*graph size* and *graph evolution scenario*) and their levels.

Table 1 Experimental design

Scenario	Graph size	Graph evolution
1	Small	Improvement
2	Small	Degradation
3	Medium	Improvement
4	Medium	Degradation
5	Large	Improvement
6	Large	Degradation

Graph structure

In the absence of a reference model for MSAs, we adopted the Barabasi-Albert model [33], which is an algorithm for generating random *scale-free networks* (SFN), whose degree distribution follows a power-law. Wheeldon et al. [34] and Potanin et al. [35] verified power-law distribution related to coupling in real Java programs, i.e., the vast majority classes have few dependencies while few classes have many dependencies. Wen et al. [36] observed that dependencies between Java packages also follow scale-free properties. Many other studies [37] [38] observed that software objects have characteristics of complex networks such as scale-free and power-law. We understand that, semantically, coupling metrics for (micro)services have the same meaning as OO coupling metrics. Therefore, we create all dependency graphs using the preferential attachment process following the power-law.

Graph size

A microservice-based system may vary in scale. Thus, we define three levels for the application size (amount of services) as follows: from five to ten services, it is considered a small application; from eleven to 25 services, it is medium; above 25 is large; however, due to computational limitations, we decided to limit it to 60 in this experiment, since we intend to evaluate the first results before scaling the number of services.

Architecture smells

For the improvement or degradation scenarios, we have chosen two coupling-related architecture problems: the concentration of incoming dependencies (problem 1) and outgoing dependencies (problem 2) around a single microservice. These problems reflect symptoms of known architecture smells with evidence of their existence in the field, such as God Component [39] or Megaservice [40][41], Hub-like Dependency [39], Bottleneck Service[41], Nanoservices[41], and The Knot[41].

The number of edges to characterize a microservice with high concentration of incoming or outgoing dependencies is defined by a percentage of the total number of services in the system, which is a parameter of this experiment. Further details on experiment parameters and how architecture smells are included and removed from dependency graphs are available in a GitHub repository available at Zenodo [42].

Graph evolution scenario

We established 21 releases (including the initial release 0) for the whole evolution of one application. The default changes during the evolution are limited to the inclusion of nodes. Additionally, we consider two levels for this factor: an improvement scenario and an degradation one. For the improvement scenario, we introduce one architecture smell in the first release, and, during the following releases, the main action is to remove the smell. For the level of degradation of the architecture, the first release is free of architecture smell and, during the following releases, we introduce an architecture smell incrementally.

In Fig. 1, we present an illustration of an architectural improvement scenario. The *I1* service is a node that concentrates four input edges which are considered a high concentration since there are few services in this application. In release 7, the only change between the 21 releases occurs when a new *MI1* service is created. Services *I2* and *I3* are no longer dependent on *I1* but are now dependent on *MI1*.

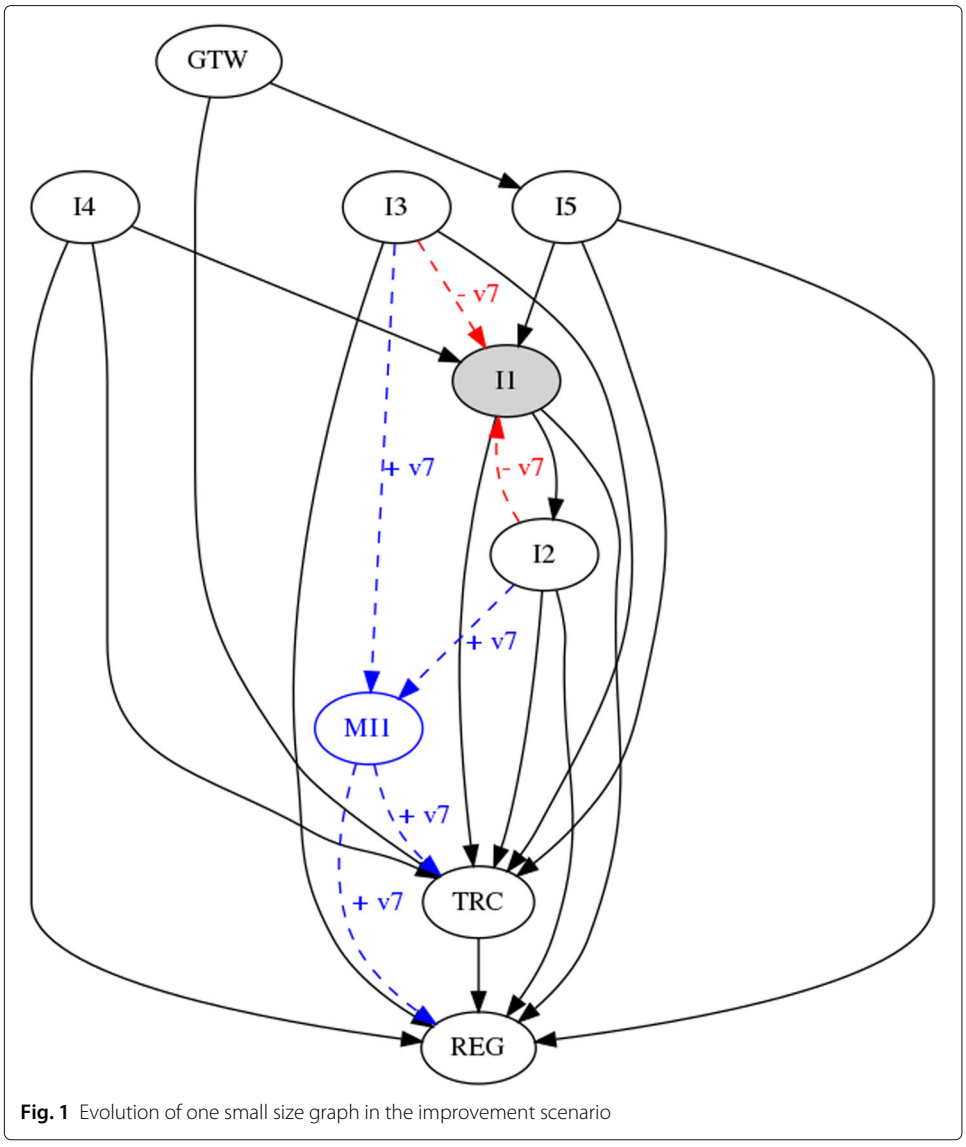

Fig. 1 Evolution of one small size graph in the improvement scenario

In Fig. 2, we see an example of an architectural degradation scenario. The *D3* service is chosen to concentrate outgoing edges. In releases 8, 16, and 20, nodes *MD1, MD2 and MD3* are included, respectively. The *D3* service then becomes dependent on these three new services, thus increasing its concentration of outgoing edges.

Microservice-related design patterns

Aiming at generating dependency graphs similar to real microservice-based systems, we apply six usual design patterns found in MSAs [43] and that can also be expressed in a dependency graph. The selected patterns are API Composition, Message Service Broker, Externalized Configuration, API Gateway, Service Registry, and Distributed Tracing. We know that some design patterns can increase coupling and also concentrate incoming or outgoing edges on a few nodes. Therefore, the method is applied to the evolution of metrics values, performing the comparison with itself, thus avoiding the creation of

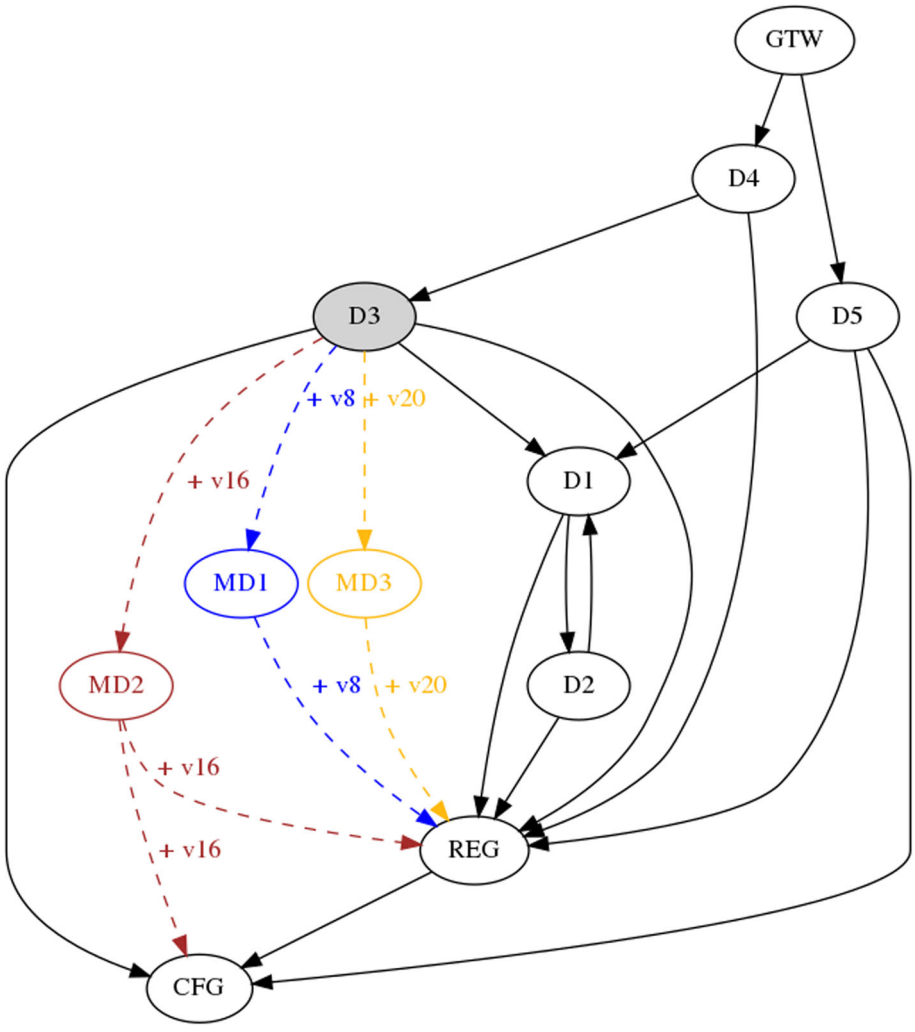

Fig. 2 Evolution of one small size graph in the degradation scenario

generic thresholds that fail to consider the different architectural decisions of each software. The configurations related to the inclusion of these design patterns are available in the GitHub[42] repository.

Metric trends

We analyze metric trends (up or down) through releases. When there is an upward trend in ADCS and SCF metric values, we assume the coupling is increasing, and we assume the coupling decreases when these metrics are trending downward. For the metrics based on the Gini index (SID and SDD), we acknowledge there may be specific situations in which this relationship may represent the opposite. For example, if we have an application in which, in an initial release, all services are fully coupled, we will have SID and SDD equal to zero. If we are removing one dependency from the same service at each release, these metrics based on Gini will show an upward trend. However, in this case, we have a decrease in coupling. Because of that, we chose a suite of metrics and perform the final

analysis together, as each one of them is more or less sensitive depending on the coupling aspect under analysis.

Replications and procedure

We need multiple trials as we have stochastic components to generate graphs so that we can quantify variation in the results. To determine the minimum number of replications, we adopted a sequential procedure proposed for simulation modeling [44]. After executing this procedure, we reached the amount of 210 replications. Based on this, we followed the experimental procedure:

- *Graph generation:* based on the factors and levels, the tool generates the dependency graph corresponding to the first release of each MSA. As we have 6 scenarios and 210 replications each, we generated 1260 experimental units. For all the units in the improvement scenario, we introduced one architecture smell in the first release.
- *Application evolution:* according to the evolution scenario, the tool generates a dependency graph for each of the next 20 releases.
- *Metrics calculation:* the metrics are calculated for each release of each MSA.
- *Metrics analysis:* we use the Cox-Stuart test to detect trends for each metric in each experimental unit, being ten statistical tests per unit considering a range of 12 observations per test. Additionally, we explore trends for the scenarios visually using the mean values for the coupling metrics evolution.
- *Hypothesis testing:* we used the *chi-square* test of independence to evaluate the experimental hypotheses for each individual replication. Also, we used the Cramér's V statistic as the *chi-square* test is sensitive to large sample sizes.

Experimental results

Firstly, we analyze the general behavior of the metrics. For each scenario, we grouped the metric values of all replications, and we calculated the mean values for each release as they are independent. Figure 3 shows the plot of the four metrics for the graphs of medium size as an example of how we can get a visual sense of the metrics with the most evident upward or downward trends.

In this analysis, the SCF metric presents a downward trend in all scenarios, even when purposefully introducing smells. Therefore, it is not able to detect the architecture changes introduced during the experiment. The SCF metric seems to be more sensitive to the number of nodes in the graph than to the coupling between them since the average number of edges does not vary significantly during evolution. An increase in SCF would reveal a chaotic architecture, where the deterioration of the architecture should be evident. However, in our simulation, there is always growth in the number of microservices, and in cases there is no such growth, the SCF metric should behave differently.

We also realized the vast majority of cases in architectural degradation scenarios present an upward trend. In contrast, for ten improvement scenarios, trends are difficult to be identified only by visual analysis. In general, the SID and SDD metrics seem to be good indicators for the architecture smells, as their trends are easier to detect when there is a concentration of input or output edges in few nodes. The ADCS metric shows more considerable differences in behavior according to the size of the application, performing better for small applications.

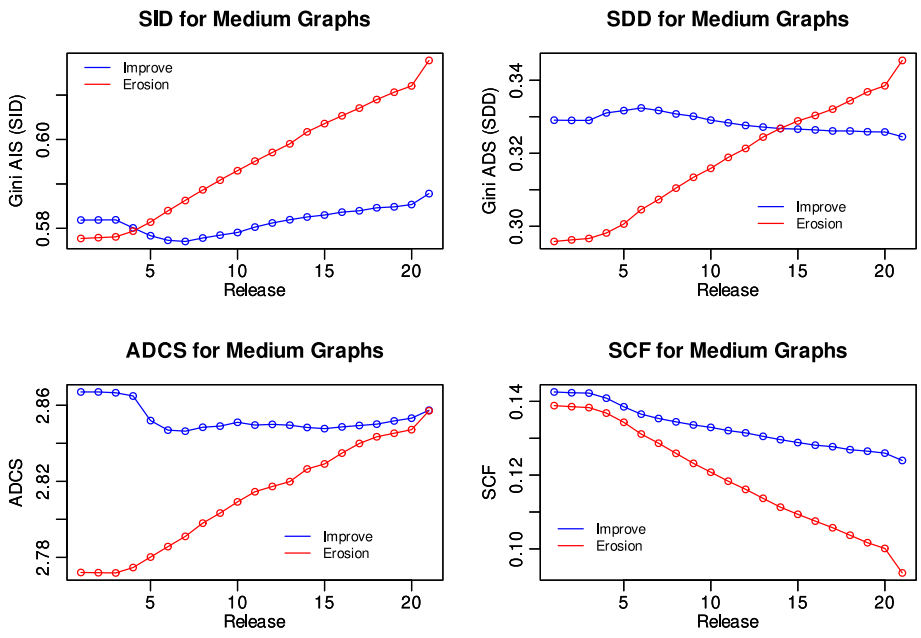

Fig. 3 Mean trends for the medium-size graphs

We use the Cox-Stuart test to characterize statistically a trend (upward or downward) in the series metrics values through the releases. In this experiment, we performed ten trend tests for each experimental unit (MSA), considering all possible intervals of 12 releases length (from release n to $n + 11$ successively). For each scenario, we determined a contingency (see Table 2).

For each MSA, we count as *Improvement Scenario* and *Decreasing Trend* when at least one of the ten tests resulted in Decreasing Trend. For instance (in Table 2 for the Medium Scenarios), the evolution of 60 applications reveals a significant decreasing trend for the SID metric when we remove the architecture smell. The same is valid for counting as *Degradation Scenario* and *Increasing Trend*; that is when at least one of the ten tests resulted in Increasing Trend. Similarly, from 210 unities (MSA) in the degradation scenario, SID revealed a significant increase for 198 MSAs.

We justify this rationale as just one single intervention is made to improve or deteriorate the application, so it must affect the series in a unique change-point. Conversely, *Improvement Scenario* and *Increasing Trend* will be computed when there is at least one test resulting in Increasing Trend and none resulting in Decreasing Trend. The opposite case (*Degradation Scenario* and *Decreasing Trend*) occurs when there is at least one test resulting in Decreasing Trend and none Increasing Trend. Finally, we count as *No Trend* only when all ten tests result in No Trend, i.e., it has no statistical significance.

Based on the contingency table for each scenario, we used the chi-square test of independence to verify how correlated are the intended evolution scenarios (Improvement or Degradation) and the result of the Cox-Stuart test for trend analysis (results in Table 3). We do not consider the SCF metric for testing the experiment's hypotheses due to

Table 2 Contingency tables for all scenarios

Metrics		Scenarios					
		Small		Medium		Large	
		Impr.[1]	Degr.[2]	Impr.[1]	Degr.[2]	Impr.[1]	Degr.[2]
SID	Decreasing trend	73	5	60	12	25	40
	No trend	16	0	80	0	113	3
	Increasing trend	121	205	70	198	72	167
SDD	Decreasing trend	64	11	73	27	79	28
	No trend	14	0	85	2	112	3
	Increasing trend	132	199	52	181	19	179
ADCS	Decreasing trend	132	18	82	67	75	100
	No trend	16	9	89	9	109	6
	Increasing trend	62	183	39	134	26	104
SCF	Decreasing trend	174	210	117	210	97	210
	No trend	29	0	85	0	111	0
	Increasing trend	7	0	8	0	2	0

[1] **Impr.** = improvement evolution scenario
[2] **Degr.** = degradation evolution scenario

its anomalous behavior (monotonic-decreasing no matter the scenario), which is also reflected in Table 2.

Table 3 also presents the Cramer's V measure. We use it in association with the chi-square test as the latter is sensitive to large sample sizes. The Cramer's V measures the correlation between two nominal variables (architectural evolution scenario and detected trends) for each coupling metric as an interval between zero (no association) and one (strong association). We consider rejecting the null hypothesis when the chi-square test (p-value <0.05) and the Cramer's V statistic ($\varphi_c > 0.5$) result in a significant association.

Therefore, we could not reject *H0* in the scenarios: SID metric with small MSAs and SDD metric with small MSAs. Except for these two combinations of metrics and scenarios, we can reject *H0* and accept the *H1* for the other 10 combinations (metrics x MSA size). The SDD metric for large graphs shows great results since Cramér's V points to a strong correlation (0.78). The SID metric also has good results, mainly for large and medium MSAs. The ADCS metric seems to work appropriately for all MSA sizes.

The presented results warn us regarding the use of the SCF metric and to validate the use of statistical trend calculations. Our method will replicate the same analysis procedure in the last step of SYMBIOTE, which corresponds to the analysis of coupling metrics.

Table 3 Experiment results per metric

Metrics		Graph size		
		Small	Medium	Large
SID	Chi-square test	96.92	173.13	145.53
	Chi-square p-value	$9.0e^{-22}$	$2.5e^{-38}$	$2.5e^{-32}$
	Cramér's V	0.48	0.64	0.59
SDD	Chi-square test	65.01	171.76	256.91
	Chi-square p-value	$7.6e^{-15}$	$5.0e^{-38}$	$1.6e^{-56}$
	Cramér's V	0.39	0.64	0.78
ADCS	Chi-square test	148.36	118.98	142.62
	Chi-square p-value	$6.1e^{-33}$	$1.4e^{-26}$	$1.1e^{-31}$
	Cramér's V	0.59	0.53	0.58

Threats to validity

We have no empirical evidence whether the model we used to create and evolve the graph structures used in the experiment resembles the graph structures of real MSAs. However, we do have evidence on this for other types of software. In architectural terms, the main difference is that MSAs have an extra level of abstraction (services).

In the wild scenario, several problems can occur together, and there may be problems that can cancel each other's effects. However, the controlled use of architecture smells in this experiment gives us the advantage of isolating the causes of metrics deterioration.

The trend analysis is effective but does not take into account *level changes* in a time series, and can cause misinterpretations when this occurs. We mitigate this by using several intervals for a single evolution so that we could detect multiple change points.

Finally, the *chi-square* test is sensitive to large sample sizes like the one we have in the experiment design considering the number of replications. Thus, it may impose a threat to conclusion validity. However, we associated the Cramér's V statistic to support the effect size analysis. Besides, the chi-square test statistic represents the independence magnitude, from which we can highlight the difference across the three metrics, corroborating the results in Table 3 and discussion.

The SYMBIOTE method

In this section, we present the SYMBIOTE method for monitoring the evolution of microservice-based systems using coupling metrics between services (overview in Fig. 4). This method was proposed by Apolinário and de França [9], but we have made some modifications to improve the method and apply it in a real case. In the following subsections, we explain the whole method, including the chosen approaches to collect and analyze the metrics and we will highlight small changes proposed in this paper.

Approach for collecting metrics

The proposed method is based on the coupling metrics SID, SDD, SCF, and ADCS. To calculate all of them, we need to build a directed graph, in which nodes represent microservices and edges represent the dependencies between them. Nonetheless, discov-

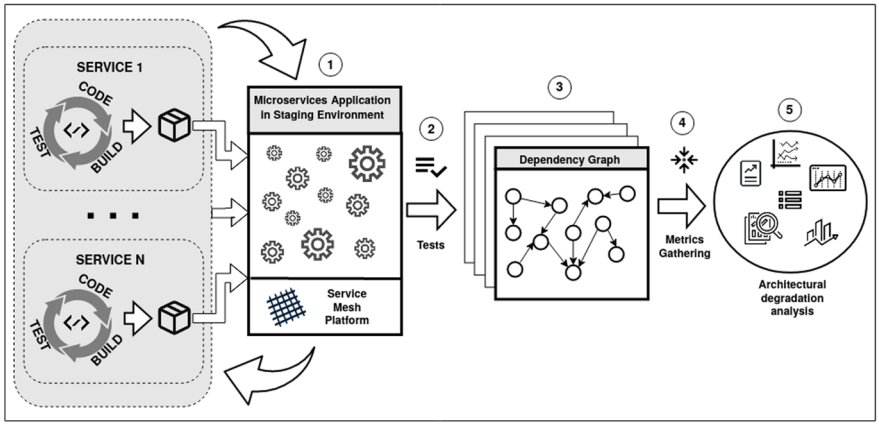

Fig. 4 Overview of the proposed method

ering every dependency between services in an MSA is a complicated task. Two potential approaches for generating this graph are static and dynamic analysis.

The static analysis uses only the source code as input and a tool to transform this code into an abstraction to be manipulated. However, in the case of service coupling metrics, it is not easy to identify all remote calls. There are several ways to implement a service call (invocation) in different programming languages. For instance, in Java using RESTful[1] services, it is possible to call services using low-level native APIs of the language to establish connections, sending HTTP requests, and handling HTTP responses. However, we can also use high-level APIs implementing the Java API for RESTful Web Services, or we can use a framework like *Spring Boot* that has more than one different abstraction for consuming services. Remote calls may be through Inversion of Control and Dependency Injection mechanisms, annotations, configuration files, polymorphism, dynamic binding, and other alternatives. In other words, extracting coupling metrics only via static analysis means restricting the analysis to particularities of a given programming language or platform. This way, it would weaken the accuracy of the dependencies between identified services and the capacity of developing a platform-independent approach.

Alternatively, dynamic analysis potentially discovers dependencies more accurately. In dynamic analysis, if all services of an application are known, the monitoring of the requests between services allows us to extract their dependencies. This type of analysis consumes more resources because it depends on the entire application running. Also, running a test suite that covers all dependencies between services is vital to success. It becomes critical since there are no guarantees that existing test suites have enough coverage in most of the current applications.

Given the advantages and disadvantages of these approaches, we selected the dynamic analysis, since it is a more comprehensive solution, potentially achieving higher accuracy for the construction of the dependency graph. It does not depend on particular programming languages or frameworks. Considering this decision, we understand this technique runs without source code instrumentation to facilitate its practical application.

The next sections detail the five steps of the SYMBIOTE. The first three steps are performed for each application release in the case of executing it in retrospect.

Step 1: Distributed tracing

Usually, each microservice in an MSA has its source code repository. As each microservice may also have its continuous deployment pipeline, we need to deploy the entire application first, run the integration tests, so that we can gather the data required to calculate the coupling metrics. That is the reason for assuming staging as the target environment for the metric collection step.

In Fig. 4, the integration pipeline (code-build-test cycle) of each microservice builds this component as a deployment unit for the staging environment. In this environment, we monitor all service requests to capture calls between services and build their dependency graph. For this, we use a *service mesh* platform. Also, we successfully tested another approach based on sniffing HTTP packets inside a Kubernetes cluster network, capable of detecting asynchronous communication using message queues in a publish-subscribe

[1] Services following the architectural style REST (Representational State Transfer)

style. Other methods could be used as long as it can recognize services and discover their dependencies.

Microservices can be deployed as containers directly in Virtual Machines, or in a container orchestrator. For each service deployed in the staging environment, we transparently inject an interceptor (also called *sidecar proxy*) that is part of the *service mesh* framework architecture, with no instrumentation required in the microservice source code. The *sidecar proxy* intercepts all incoming and outgoing requests for a service. Figure 5 shows the microservices communicate exclusively with their interceptor that, in turn, interacts with each other, logging all request-response information to a *distributed tracing system*. Tracing information is fundamental to identify dependencies between microservices dynamically.

Validation of the approach for collecting metrics

Due to the complexity of collecting metrics dynamically, we describe the scenario we developed for a feasibility study of this solution. The intention is to test the extraction of the information needed to build the dependency graph and to identify the limitations of the adopted approach. We use a strategy in which no instrumentation is required in the source code to verify the feasibility of tracking all interactions between services in an environment in which a microservice-based application runs.

The test scenario (Fig. 5) uses the following technologies:

- Docker: in general, microservices are deployed into a containerized environment. Docker is one of the most popular container technology used in software development practice. Figure 5 presents an example (sample application) where each service (transaction-generator, compute-interest-api, account-database, and account-summary) is encapsulated into a Docker container, which is deployed into a Kubernetes POD (a group of one or more containers) managed by a Kubernetes Cluster.

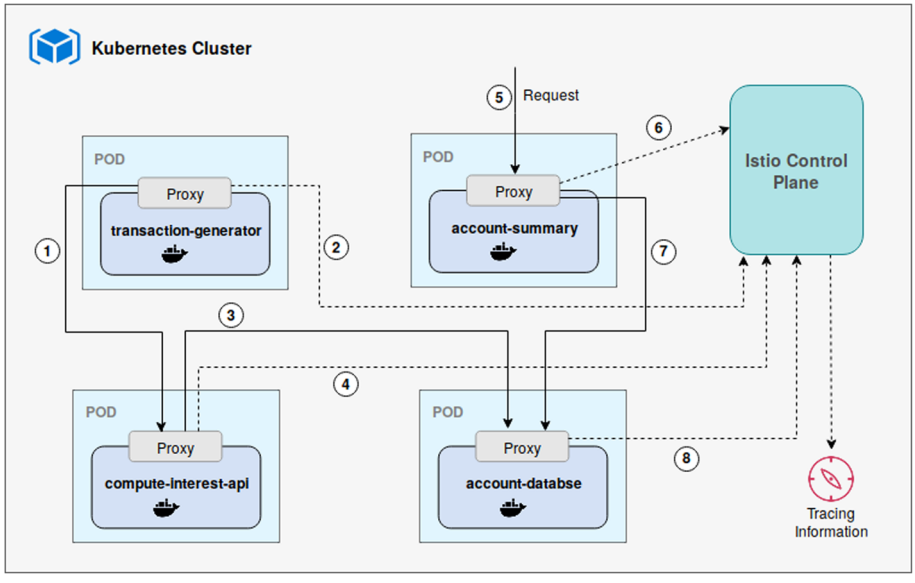

Fig. 5 Scenario for collecting service metrics

- Kubernetes: it is a tool to orchestrate containers and to manage applications and deployments. Kubernetes have also been widely used in the software industry. It is not mandatory to use Kubernetes for the method proposed in this work, but it is part of our testing scenario for integration with Istio.
- Istio: it is a service mesh platform used to manage distributed applications. We have chosen Istio because it requires no instrumentation in the source code to work, but as already mentioned in the "Step 1: Distributed tracing" section, it is possible to use another solution based on network sniffing. *Istio Control Plane* is the component responsible for collecting all service requests sent by interceptors (sidecar proxy services comprising the *Istio Data Plane* component).
- Jaeger: it is an open-source distributed tracing system used to consume the calls between services (tracing information) stored on Istio and made available through a REST API.

We set up an environment with these tools, and we used a sample application available at GitHub [45]. In Fig. 5, we can see this application composed of their microservices deployed in this environment. We deployed this application in a Kubernetes cluster. For each microservice in the staging environment, we inject the sidecar proxy transparently. The solid arrows (1, 3, and 7) represent communication between them. The dashed arrows represent the sidecar proxies' communication with the Istio Control Plane that feeds the distributed tracing. This way, they communicate exclusively with their sidecar proxy that, in turn, interacts with other sidecars, sending all request-response information to the Istio Control Plane.

Therefore, Istio can collect the information we need to assemble a dependency graph across the microservices from this example application. The dashed arrows (2, 4, 6, 8) in Fig. 5 represent the dependency information sent from proxies to Istio Control Plane. This sample application is straightforward enough to evaluate the pathway to obtain the required information to calculate the metrics. The Jaeger plugin installed in Kubernetes provides this information.

Step 2: Application test execution

After deploying into a staging environment, integration test cases are executed. The test suite needs to ensure coverage for all dependencies between services. During the execution of the tests, the service mesh framework captures all communications between services and sends them to a distributed tracing system. Therefore, this is an intermediate step in the SYMBIOTE, whose sole purpose is to capture the application's behavior in terms of dependencies between services.

Applications should have a test suite with adequate coverage to ensure their quality. For SYMBIOTE, we are assuming this suite exercises all remote calls among the services. If the application has no such test or tests have partial coverage, SYMBIOTE can only work by collecting data from production.

Step 3: Dependency graph generation

After executing integration tests, the distributed tracing system will have stored all calls between services. This information can be represented as a dependency graph for each deployed version. Figure 6 shows the dependency graph of the microservice applica-

tion used to test the environment described in Step 1: Distributed tracing section and illustraded in Fig. 5.

This step can be automated by querying the requests performed during the tests. For instance, Jaeger exposes an API available returning this request data in JSON format. If using a network sniffer, it requires a parser to extract the information from a log file.

Each dependency graph instance should be stored to keep track of its evolution over time. For describing graphs, we suggest using the DOT language [46]. The dependency graphs from the past releases are input for the collection of metrics.

Step 4: Calculating metrics

From the dependency information, basic AIS and ADS metrics are calculated for each microservice. Based on this, we calculate the SID, SDD, and ADCS metrics. Although the results of the experiment alert us to the usefulness of the SCF metric, we decided to include it again in our suite of metrics (unlike what was proposed in the original paper) since the experimental units may not represent all possible architecture configurations and evolution paths. For the SCF calculation, we only need the total number of dependencies and the total number of services.

As an example, we calculate the metrics of the sample application depicted in Fig. 5. Table 4 presents the microservice metrics AIS and ADS, and Table 5 shows the metrics for the whole application calculated for this sample scenario.

Step 5: Analysis of collected metrics

The main modification from the original method to the one in this article is the analysis of the collected metrics. We realize that the interpretation of Gini-based metrics (SID and SDD) must be different from ADCS and SCF metrics. Therefore, we decided to clarify the rationale for this analysis and changed the architectural degradation warning mechanism.

This step is based on the result of the experimental analysis of the coupling metrics (see the "Analysis of service coupling metrics" section). Based on the metrics collected over time, we can analyze how they evolve. We aim to identify variation in these metrics

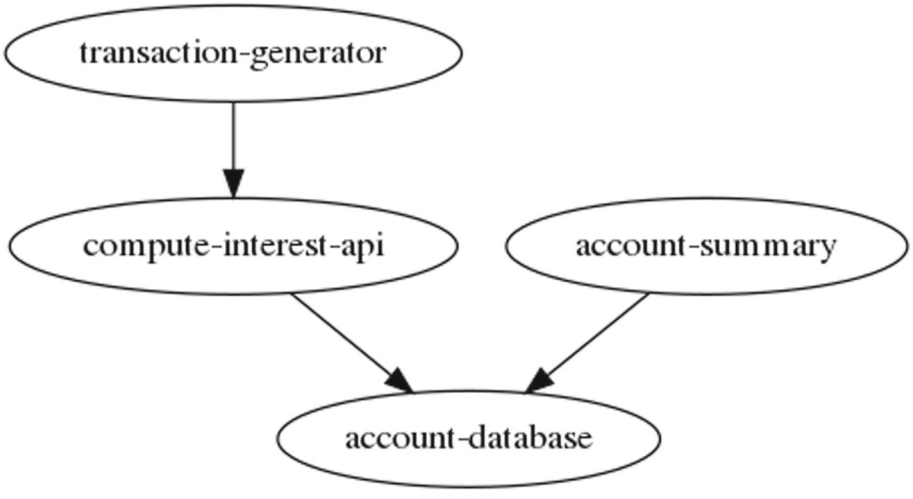

Fig. 6 Example of a generated dependency graph

Table 4 ADS and AIS coupling metrics per microservices

Microservice	AIS	ADS
Transaction-generator	0	1
Compute-interest-api	1	1
Account-database	2	0
Account-summary	0	1

that typify violations of architectural principles, such as low coupling. The initial focus of this work was to identify changes that can have a negative effect, that is, when coupling increases. However, positive variations (decreasing coupling) can also occur and engineers usually plan for them. Therefore, SYMBIOTE supports software engineers to track the (positive or negative) effects of architectural changes on the coupling between services when analyzing the behavior of the metrics and their trends.

For the two Gini-based metrics (SID and SDD), the interpretation is the same: values closer to zero mean an even coupling distribution among microservices. In contrast, values closer to one say the opposite, i.e., the coupling distribution among microservices varies a lot. However, it is not possible to qualify an upward or downward trend for these metrics w.r.t. improvement or degradation. The main reason for that regards the level of coupling at the beginning of the trend. For instance, if an evenly distributed and high coupling is observed in the beginning of an upward trend, it may be the case the coupling is improving since few services are reducing coupling in the next releases; on the other hand, if a evenly distributed and low coupling is observed in the beginning of an upward trend, it may be the case the coupling is degrading since few services are now concentrating more dependencies in the next releases.

For the ADCS and SCF metrics, an upward trend represents a bad indicator, and a downward trend could be good for the architecture. ADCS metric measures the density of edges in the graph, so a downward trend also means improved coupling and an upward trend coupling deterioration, as fewer edges represent lower coupling between services. The SCF metric measures the ratio between existing and possible dependencies between microservices; therefore, it is similar to ADCS in meaning but different in its numeric values. SCF is also more sensitive to the inclusion and exclusion of services. Therefore, the interpretation of the SCF variation is the same as for ADCS.

We use the Cox-Stuart test to analyze trends for the collected metrics. This test is simple and useful for trending detection. We set $\alpha = 0.5$ and $p - value < 0.05$ (confidence interval of 95%). For each metric of an application, we calculate the trend and classify it into *decreasing*, *increasing*, or *no trend*.

Individual analysis of the values of a metric can lead to misunderstandings about the actual behavior. Therefore, the SYMBIOTE takes into account the joint analysis of the metrics. The method generates a monitoring alert when three out of the four metrics indicate a significant trend, considering any (upward or downward) trend for SID and SDD, and only upward trends for ADCS and SCF. Notice that the development team still

Table 5 Coupling metrics for the sample application

SID	SDD	ADCS	SCF
0.58	0.25	0.75	0.25

can observe isolated trends for each metric, but no alerts will be emitted, due to possible increase in false positives for architectural degradation.

Architectural analysis monitoring

The proposed method is entirely automated. A repository stores information about dependencies and calculated metrics. Whenever new information reaches the repository, it triggers an event that a dashboard tool consumes through APIs. Also, we propose a dashboard containing:

- *Metrics evolution charts:* this element contains three Run Charts (one for each metric: SID, SDD, ADCS, and SCF). The *x*-axis represents the releases of the application, and the *y*-axis shows the values of the calculated metrics. By the charts, it is possible to visualize the behavior of each metric throughout the software evolution. When the Cox-Stuart test detects a trend, the data series is highlighted (for instance, with a different color). If there is evidence of architectural degradation, it displays an alert message on this element.
- *Dependency graph visualization:* graphic of the dependency graph representing the application's latest release. Service names can filter the graph, and it is possible to zoom in and out for better visualization. It is also possible to browse the nodes (services). When selecting a service, it presents a table with individual values for ADS and AIS in each release.
- *Individual ADS and AIS values:* this element consists of two summary tables for the ADS and AIS metrics, respectively. Each table contains the top five services with the highest metric values in the last releases in decreasing order.

This way, software architects and developers can monitor the information to make informed architectural decisions when demanded.

Method evaluation

The experiment with artificially generated data allowed the evaluation of the coupling metrics and gaining knowledge for the development of the SYMBIOTE analysis method. However, real applications can be different from the simulations produced in the experiment. Therefore, in this section, we report the retrospective application of the method in a real case. This evaluation has the purpose of verifying the effectiveness of the method under a real scenario. Besides, it has the potential to generate future improvements to the proposed method.

Selecting a real case

Although we recognize the existence of public available MSA repositories, some hindrances cannot be disregarded. Recent work has shown some barriers. Aderaldo et al. [47] identified an existing gap of repeatable empirical research in microservices due to the low availability of microservice applications to the Software Engineering research community. Zhou et al. [27] and Jamishidi et al. [8] report the gap of benchmark systems reflecting the characteristics of real microservice systems. In [28], the authors create a dataset of MSA open source projects. Most of the applications in this dataset are demo or toy projects. Furthermore, a suitable case for evaluating the proposed method demands:

1. To be a real case
2. To have at least 10 *stable* releases to be able to detect trends and variations in metrics, besides, unstable releases do not allow to perform dynamic analysis
3. Being an active project, as our experience with some projects (SiteWhere, Spinnaker, OpenEBS, Lelylan, Magda, and others) shows the deployment of this kind of system is almost unfeasible without minimum support, which also hinders dynamic analysis

When choosing a real case, we take into account a search for the top projects with the keyword "microservices" on the GitHub platform and also lists already compiled in research papers [28] [47] [48]. Based on the above criteria and also considering factors such as its complete documentation and use by large companies, we chose the *Spinnaker* [49] application.

Spinnaker

Spinnaker is an open-source tool for managing continuous software deliveries. It has features for application-level and deployment-level management. Further details of the Spinnaker can be found at [49]. Further details on the Spinnaker environment and configurations used in this validation can be consulted at [50].

Architecture

Figure 7 presents all nine services that integrate Spinnaker in the latest version. Green and gold boxes are components considered external to the application. This diagram was useful to verify that our integration tests (the "Integration tests" section) were able to perform all possible dependencies between services.

The *deck* component represents the system UI and is not considered a microservice. The *gate* service implements the API Gateway design pattern on Spinnaker. Note it has outgoing dependencies for all other services. *Orca* is the service responsible for the orchestration within the application, i.e., it coordinates the execution of the other services. The *clouddriver* service is responsible for the integration with cloud provider services. *Front50* is responsible for the persistence of the entities representing Spinnaker's key concepts. *Rosco* is responsible for producing VM images for the various supported cloud providers. The *igor* service integrates Spinnaker with Continuous Integration tools such as Jenkins and Travis. *Echo* is an event router and scheduler to trigger continuous integration pipelines. The *fiat* service implements the Spinnaker authorization mechanism, and *Kayenta* is responsible for automating the canary analysis feature.

Environment

Setting up an environment to deploy Spinnaker requires performing a series of tasks. Due to the hardware requirements (18 GB of RAM and a 4 core CPU) to run the Spinnaker, we deployed it on the Google Cloud Platform (GCP), as in Fig. 8. We installed Spinnaker in its distributed mode, which is the recommended mode for deployments in production. This environment followed the model proposed in the Spinnaker documentation[2].

Although there are several ways to install Spinnaker, such as using Helm charts, the Spinnaker documentation strongly recommends using the Halyard tool for any type of deployment. Halyard is a command-line tool developed in Java that is responsible

[2] https://spinnaker.io/docs/setup/install/providers/kubernetes-v2/gke/

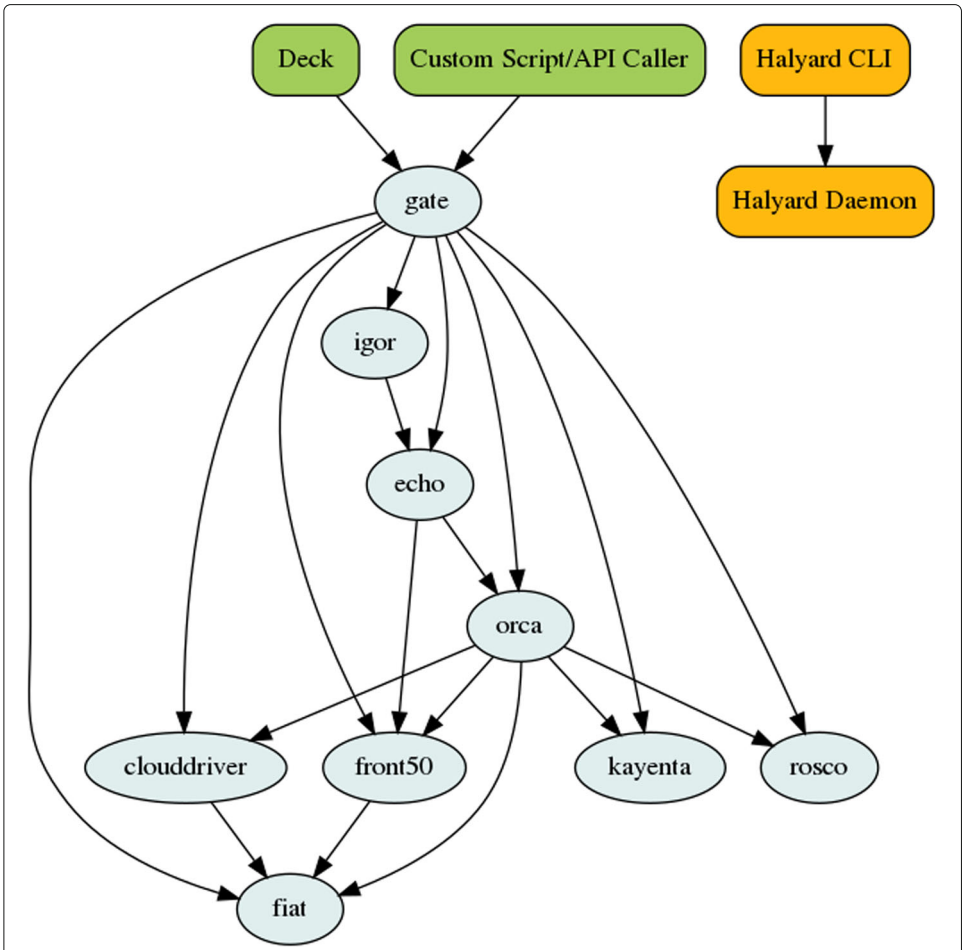

Fig. 7 Spinnaker components

for Spinnaker's configuration and deployment activities. When deploying to apply the SYMBIOTE method, we chose to install Halyard on a VM (Fig. 8) created on the GCP.

In addition to the VM in Google Compute Engine (GCE), we are using Google Identity and Access Management (IAM) users, roles and permissions, and Google Compute Storage (GCS) for Spinnaker to create and manage buckets. We also used the Google Kubernetes Engine (GKE) to create the cluster on which we installed Spinnaker, Istio, and Kiali (these 2 latest are requirements for the SYMBIOTE method).

Configuration

To configure the basic deployment of a version of Spinnaker, several steps were necessary. We have followed all the configuration steps described in the Spinnaker documentation. Much of the additional configuration was required so that we could run the integration tests that exercise dependencies between services. The most significant complexity of Spinnaker validation was installing and configuring Spinnaker itself. Specifically for SYMBIOTE, we installed Istio and Kiali which have simple installation procedures for the Kubernetes environment. We did not perform performance tests to check if there is any impact on Spinnaker execution when Istio and Kiali are deployed, but we did not detect

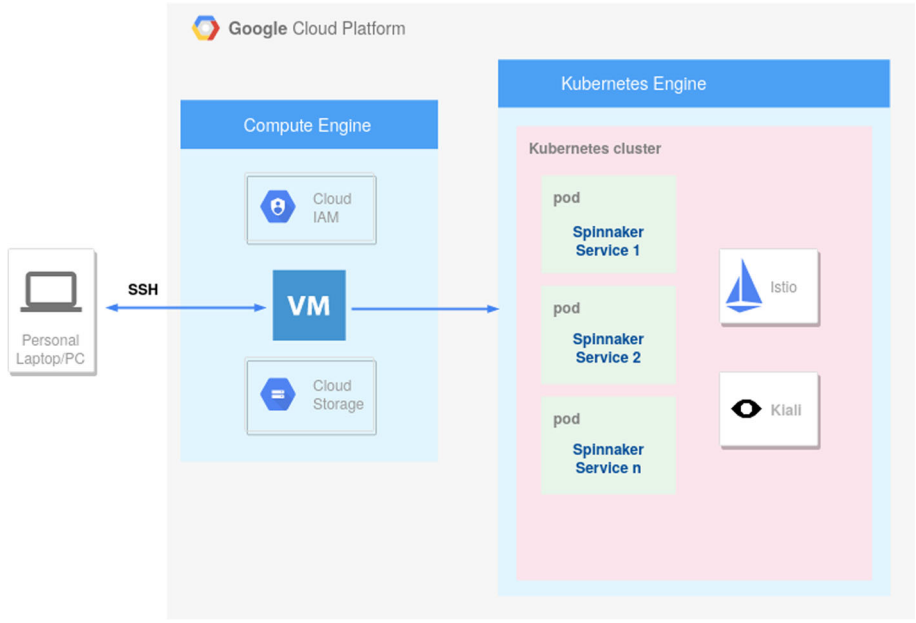

Fig. 8 Spinnaker deployment environment overview

any problems of this type during this validation. Ideally, SYMBIOTE should be run in a test or staging environment and therefore an eventual decrease in performance should not be a serious problem.

Integration tests

The integration tests in Spinnaker aims to make the services carry out all possible communications with each other. We aim not testing the system's functionalities, but only a minimum set that allows us to generate a complete dependency graph. We defined and implemented some manual testing scenarios to achieve our goal because we did not find any integration tests in the public Spinnaker repositories.

Releases

We selected Spinnaker versions from major release 1 (first stable release) to deploy and extract dependency graphs. From it, we selected minor releases starting from 6 to 22 (the latest version when the evaluation was performed). An important aspect to justify not going back further is a practical reason: older releases have limited documentation, including their deployment steps, since the Spinnaker team only supports the last three versions. This way, we avoid working with those past versions to not introduce additional confounding factors. Initially, we intended to get the full number of versions (considering major, minor, and bug fix releases). However, at a first glance, we noticed that bug fix releases did not make structural changes (no coupling-related changes); this explains why we moved to minor releases only. Then, we intended to analyze all minor releases. So, the decision to start with minor release 6 was based on the fact that we identified in the release changelog that minor release 7 was the only one that had a change in the number of services (addition of 1 new service) and we needed to analyze if the method was capable of identifying this in our analysis. We always took the latest patch within a minor release. Table 6 lists all the releases used.

Table 6 Spinnaker releases used

#	Version number	Release date
1	1.6.2	2018-07-26
2	1.7.8	2018-08-29
3	1.8.7	2018-09-28
4	1.9.5	2018-10-01
5	1.10.14	2019-03-01
6	1.11.13	2019-05-01
7	1.12.14	2019-07-08
8	1.13.12	2019-07-29
9	1.14.15	2019-09-16
10	1.15.7	2019-12-03
11	1.16.7	2020-03-09
12	1.17.10	2020-04-03
13	1.18.12	2020-05-26
14	1.19.14	2020-08-13
15	1.20.7	2020-07-22
16	1.21.4	2020-08-13
17	1.22.1	2020-08-31

Collected metrics

For each release listed in Table 6, we configure, deploy, and execute the integration tests. Istio telemetry captures the dependencies, and the Kiali tool provides this information visually (all graphics are available at GiHub repository [50]) and also through an API RESTful. We transformed the JSON output from the Kiali API to the DOT language. Based on this DOT files, we reuse the tool that extracts and calculate metrics created for the experiment reported in the "Analysis of service coupling metrics" section.

After that, we created an R script to perform the trend analysis and generate graphs to visualize the evolution of each of the metrics.

All artifacts used for applying the SYMBIOTE in Spinnaker are available at a public repository [50].

We compared the graphs from the last release (1.22) with the Spinnaker architecture available in the online documentation to verify the coverage of our integration tests. We can see the difference found in Fig. 9 (blue dashed edges are the difference), which shows that our method could find more dependencies than those illustrated in Fig. 7. We believe the documentation may be out of date or the illustration refers to a more logical view of architecture, not highlighting dependencies that are not part of this view. Anyway, this shows that our strategy of collecting dependencies is adequate.

Results

Table 7 contains information about the 17 releases of Spinnaker analyzed using the SYMBIOTE method. The number of services only changes once (from 8 to 9), when in version 1.7 the team added the *kayenta* service. Dependencies vary from 21 (version 1.6) to 27 (version 1.22), representing an increase of 28.6%. Regarding measures, the values of the last release are almost always higher than those of the initial release. The SCF metric values present a slightly different behavior in this regard since its initial and final values are the same, with a initial downward variation and then upwards.

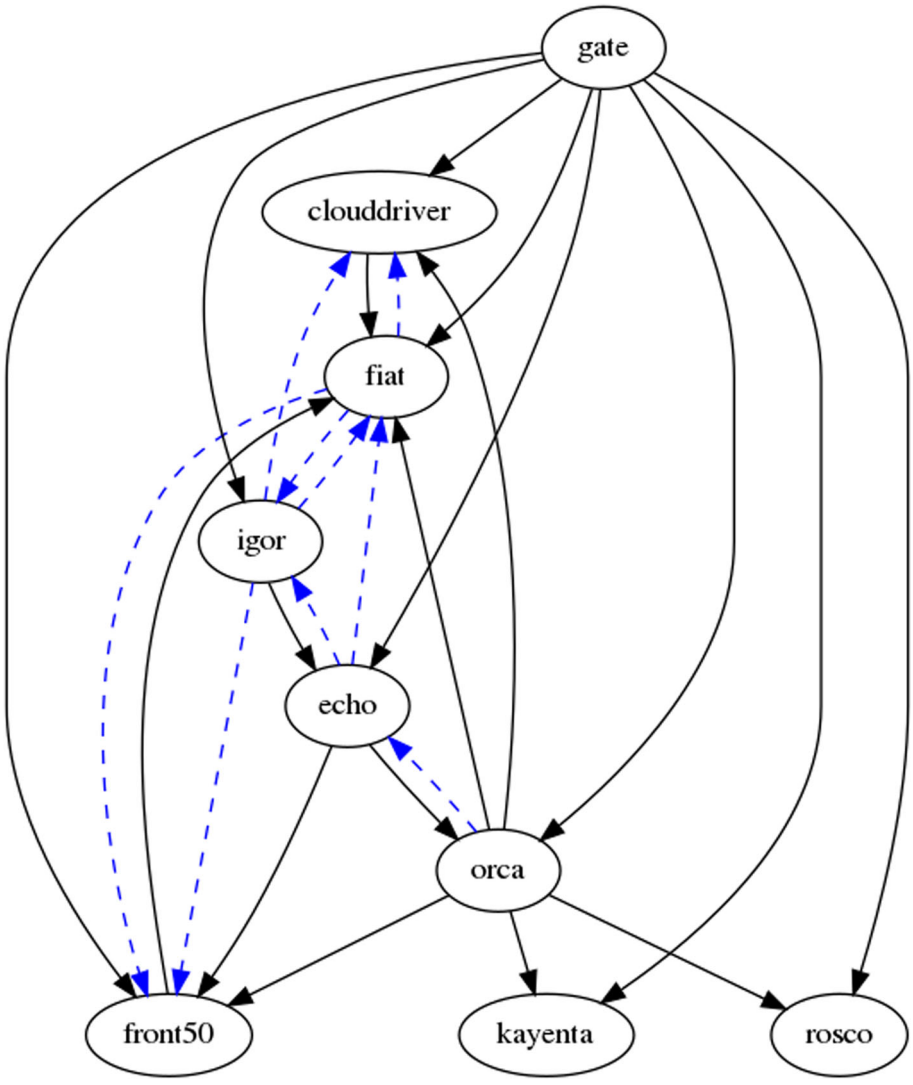

Fig. 9 Dependencies of release 1.22 emphasizing the difference (dashed blue edges) to the architecture specification

In order to evaluate and present some ways of using the method, we performed two types of analysis for Spinnaker. First, we used a single interval that contains all 17 releases (from version 1.6 to 1.22), and in the second, we performed an analysis for eight release intervals of length 10. Figure 10 presents the trend analysis results for one single interval containing a total of 17 releases. Therefore, in this first analysis, the SYMBIOTE method issued an alert for indications of architectural degradation, as the metrics SID, ADCS, and SCF (three out of four) indicate an upward trend in the metrics values throughout all 17 releases.

Second, Table 8 presents a summary of the trend analysis for each release interval, as well as the output of the SYMBIOTE method. For the first two intervals, SYMBIOTE did not detect any signs of architectural degradation (NO in column Alert), because metrics SID, SDD, and SCF did not detect any type of trend for interval 1, and, in interval 2,

Table 7 Spinnaker coupling metrics

Version	Services	Dependencies	SID	SDD	ADCS	SCF
1.6.2	8	21	0.2678571	0.4583333	2.625000	0.3750000
1.7.8	9	23	0.2608696	0.5410628	2.555556	0.3194444
1.8.7	9	22	0.2525253	0.5252525	2.444444	0.3055556
1.9.5	9	23	0.2608696	0.5410628	2.555556	0.3194444
1.10.14	9	23	0.2608696	0.5410628	2.555556	0.3194444
1.11.13	9	24	0.2500000	0.5370370	2.666667	0.3333333
1.12.14	9	24	0.2500000	0.5370370	2.666667	0.3333333
1.13.12	9	23	0.2608696	0.5314010	2.555556	0.3194444
1.14.15	9	25	0.2755556	0.4977778	2.777778	0.3472222
1.15.7	9	25	0.2755556	0.4977778	2.777778	0.3472222
1.16.7	9	26	0.2991453	0.4957265	2.888889	0.3611111
1.17.10	9	26	0.2991453	0.4957265	2.888889	0.3611111
1.18.12	9	26	0.2991453	0.4957265	2.888889	0.3611111
1.19.14	9	26	0.2991453	0.4957265	2.888889	0.3611111
1.20.7	9	26	0.2991453	0.4957265	2.888889	0.3611111
1.21.4	9	27	0.3127572	0.4855967	3.000000	0.3750000
1.22.1	9	27	0.3127572	0.4855967	3.000000	0.3750000

despite the SCF metric indicating an upward trend, SID and SDD metrics have indicated no trend. For intervals 3 to 8, the method issued an alert for the indication of architectural degradation, as the metrics SID, ADCS, and SCF showed an upward trend.

For each of the metrics, we also analyzed the intervals that had the most notable changes (either upward or downward). For these intervals, we investigated what were the main dependency changes that caused the fluctuations in the metrics values. For all of them,

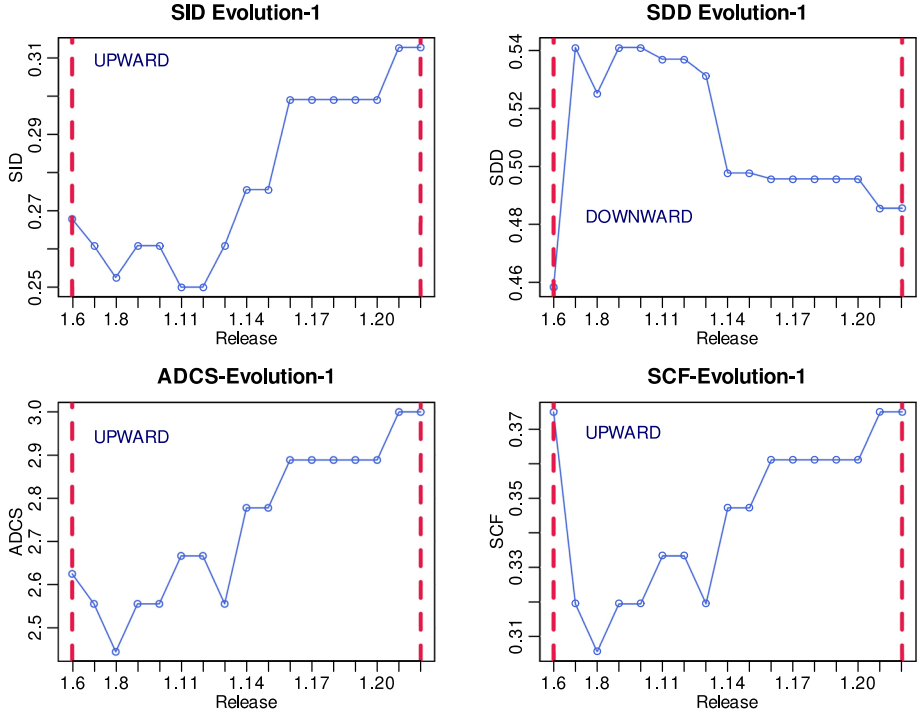

Fig. 10 Coupling metrics evolution for Spinnaker's 17-releases interval

Table 8 Spinnaker coupling metrics analysis for release intervals

Interval	Begin	End	SID*	SDD*	ADCS*	SCF*	Alert
1	1.6.2	1.15.7	N	N	U	N	No
2	1.7.8	1.16.7	N	N	U	U	No
3	1.8.7	1.17.10	U	N	U	U	Yes
4	1.9.5	1.18.12	U	D	U	U	Yes
5	1.10.14	1.19.14	U	D	U	U	Yes
6	1.11.13	1.20.7	U	D	U	U	Yes
7	1.12.14	1.21.4	U	D	U	U	Yes
8	1.13.12	1.22.1	U	D	U	U	Yes

**D* downward trend, *N* no trend, *U* upward trend

we were able to find the relationship between architectural changes and variations in the values of the metrics. For example, we will detail this investigation for the SID metric below.

Figure 11 shows the SID evolution for the 8 intervals of 10 releases each. We can see an upward trend for this metric, except for the first 2 intervals (SID Evolution-1 and 2). We can also notice that, until version 1.12, the SID metric did not present great variation. However, in the interval starting at 1.13 and ending at 1.16, there was a greater variation of this metric. Undoubtedly, the changes made between 1.13 and 1.16 are decisive for the upward trend in the metric. Analyzing the changes between the dependency graph in this release interval (see Fig. 12), the *fiat* service received two additional incoming edges (*fiat* AIS went from 2 to 3) and *igor* service received one additional incoming edge (*fiat* AIS went from 4 to 6). Considering the AIS average went from 2.5 to 2.8 in the same interval, *fiat* and *igor* services concentrated more input edges, thus explaining the increase in the value of the SID metric, i.e., the service importance is more unbalanced.

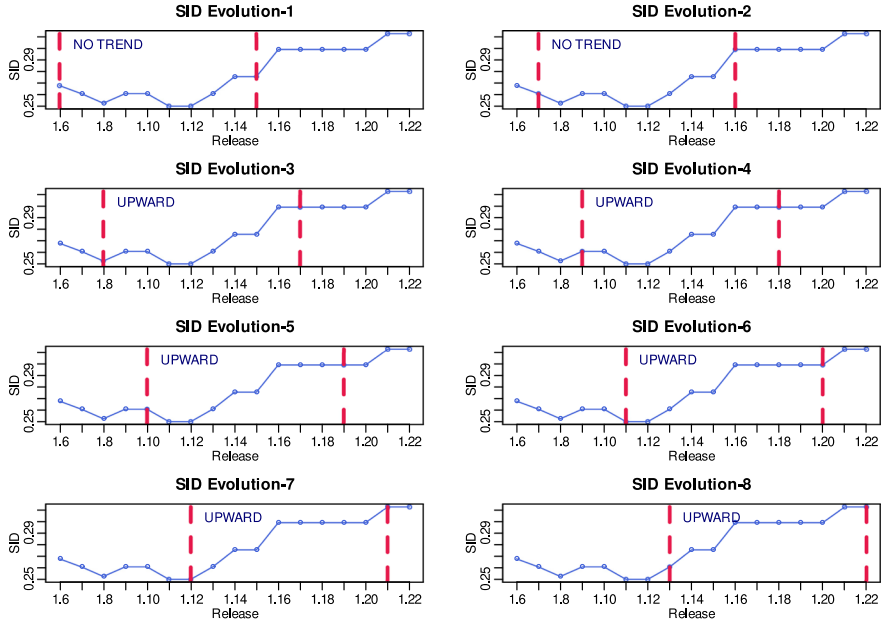

Fig. 11 SID metric evolution for Spinnaker

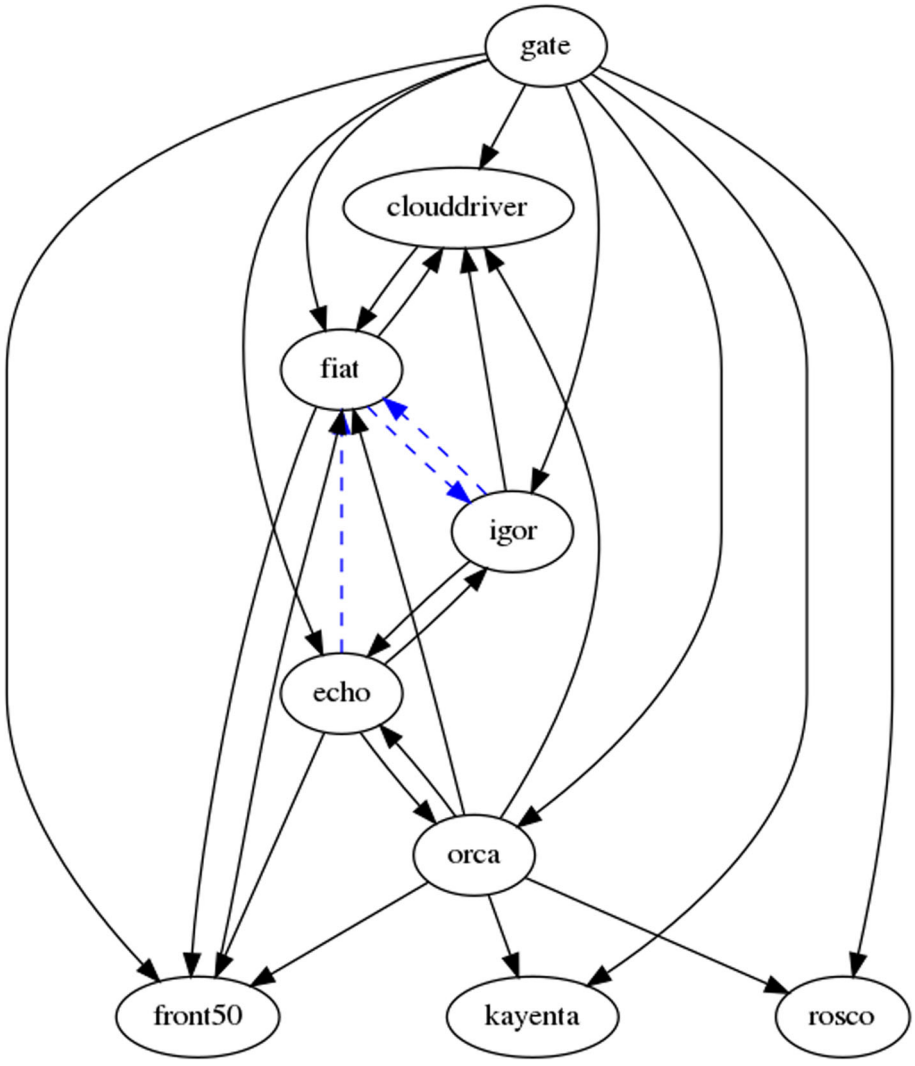

Fig. 12 Differences between releases 1.13 and 1.16

Discussion

The results of the application of SYMBIOTE in a real scenario shows the selected metrics can reveal the targeted coupling aspects. This way, when investigating the most significant variations of the metrics, we always explain them with the changes that occurred in the application architecture. For instance, metrics such as ADCS and SCF, which measure edge density in the graph, support the observation of the low coupling architectural principle. Furthermore, the SID and SDD metrics can be useful for identifying known architecture smells as MegaService, Hub-like Dependency, etc. (the "Architecture smells" section), in which there is a concentration of responsibilities on few nodes, disregarding principles such as Single Responsibility.

The trend analysis strategy by release interval allows software architects or engineers to take a snapshot of different moments in the software evolution. As we could observe in the Spinnaker case, the method did not detect any signs of architectural degradation in the first eleven releases analyzed when considering at least three metrics. However, the

increase in edge density in the graph from the twelfth release analyzed made the method alert for signs of architectural degradation. That is, in the first releases, even though ADCS and SCF metrics presents an increasing coupling, the method has not yet considered it as an indication of architectural degradation. This corroborates with our statement that not every increase in coupling can be considered an architectural degradation, but we get more confident when various coupling symptoms accumulate over time.

In Spinnaker, ten releases is the minimum interval length for which the SYMBIOTE method is able to detect trends. Below that, the method always indicates "No Trend." That is, very small release intervals are not enough for the method to detect trends. Furthermore, it is important to highlight this trend analysis is performed in a project practicing Continuous Delivery, where releases are shorter in time and amount of changes. So, the minimum interval may be different depending on the frequency of releases. In any case, a minimum number of releases shall be required. In the future, the method could adopt another statistical method for analyzing trends and this minimum number of releases could be different. However, in the context of CSE, in which releases are frequent, this minimum number should not be a problem, since frequent releases contain fewer changes.

SYMBIOTE aims to be used as an instrument for decision-making, since in Software Engineering practice there may be trade-offs between architectural principles and software requirements, mainly those related to scalability and following certain design patterns. For instance, certain patterns increase coupling between services. In Spinnaker, we can mention the API Gateway pattern implemented by the *gate* service, which depends on all the other eight Spinnaker services. In other words, the ADS measure for this service is very large when compared to the other services, which means the SDD measure has also a high value because it reveals an irregular distribution of the ADS metric among all services.

The SCF metric has a different behavior in Spinnaker when compared to what we observed in the experiment with artificially generated data, in which the SCF metric would always decrease. The reason for this is the evolution model used in the previous experiment always assumes an increase in the number of nodes, while in Spinnaker the number of nodes increases only once (add 1 service in version 1.7).

The explanation of the SDD behavior needs further consideration. Though all other metrics are trending upwards, it trends downwards. The first consideration is the SDD value for the last release (0.4855967) is greater than for the first one (0.4583333). Another consideration is that this metric presents a significant increase from release 1.6 to 1.7 because a new service is introduced (with ADS = 0) and the *gate* and *orca* services (which already have the largest ADS in the application) increase their ADS by 1, thus unbalancing the distribution of the ADS metric. Throughout evolution, the dependencies added mostly vary in source nodes than in target nodes, i.e., a few increase their AIS (worsening the SID) while the ADS is better distributed (improving the SDD).

One of the possible causes for the increase in dependencies between Spinnaker services may be the responsibilities of the services are increasing, and the team may be avoiding the creation of other services. Another possibility is that architects have postponed some dependencies. For example, the authorization service (*fiat*) received several incoming edges throughout evolution, which may indicate that some services only started to have an authorization mechanism after some releases.

In Spinnaker, most of the analyzed coupling metrics worsen throughout the evolution, revealing the joint analysis of the metrics made by SYMBIOTE is useful and has a high potential to diagnose common architectural issues related to tight coupling.

As an answer to our research question, we conclude that, based on the experiment results and the evaluation in a real case, the continuous monitoring of the four adopted coupling metrics and the trend analysis embedded in the SYMBIOTE method provide an effective approach to identify meaningful architectural changes in microservice-based application coupling, especially considering architectural degradation. Finally, this conclusion is limited to the scope of our observations, i.e., the scenarios represented by the evaluation settings.

The worsening of the metrics may also confirm Lehman's law, which states there is a tendency for software quality to decline throughout its evolution. However, due to generalization limitations of this work, it is not possible to map the conditions in which the method is more useful or more appropriate yet. Previous knowledge of architecture is also necessary to be able to state categorically whether the evidence captured by SYMBIOTE reflects a real situation of architectural degradation or a known technical debt. One future work is to validate these results with the Spinnaker team and also to execute the method in other real cases.

Limitations

The dependency between the *orca* service and *kayenta* in older releases could not be reproduced by our tests, since we had problems related to the supported version of Kubernetes and GCP. However, we verified in the source code this dependency exists and therefore we manually include it in the dependency graph before calculating the metrics.

Some scenarios of our integration tests do not run successfully due to quota limitations in GCP or limitations related to version incompatibility between Spinnaker and cloud providers, Kubernetes API. However, in all these cases, the execution was enough to exercise calls between the services and capture the dependencies, even if errors occur.

Architectural degradation is not caused only by high coupling. However, the SYMBIOTE is unable to detect architectural degradation by factors other than this. For example, source code smells internal to microservices may lead to other types of architectural degradation that cannot be identified by the SYMBIOTE method.

Software engineers who want to track the values of the SID and SDD metrics per release should be careful with interpretations based only on the values. They are calculated using the Gini coefficient and, therefore, values closer to zero not necessarily mean the architecture is good. For instance, if the application is represented by a complete digraph (every pair of nodes is connected by a pair of edges), the SID and SDD metrics will be 0, but logically the architecture would be exactly what we want to avoid considering the cyclical dependencies.

We have no guarantee that in the older versions the captured dependency graphs are complete. However, when comparing versions, we realize this risk is reduced, as we do not detect dependencies that disappear over the evolution without a valid reason.

Although the coupling metrics were evaluated in an experiment and SYMBIOTE was successfully applied in a real case, we cannot ignore that we have generalization problems in both studies. This implies that, in the wild, we can find microservice architectures with different structures and evolution from the scenarios faced in this work.

One improvement opportunity to the SYMBIOTE regards the selection of trends detection methods robust enough to deal with fewer data points. Currently, we adopt the Cox-Stuart, but we could not find any statistical limitation regarding the number of observations for this test identifying trends. However, for very small samples, the test has less power. In the metrics experiment, the length of 12 data points was the minimum number of observations for which the Cox-Stuart method significantly identified a trend. For any shorter data series, it was not able to detect trends. In the Spinnaker evaluation, we started with the same length (12). However, as we had less releases, we decided to test with a lower number of releases. And the minimum we found for this data set was 10 releases. The test could not detect trends for data series with nine or less data points, no matter the setting. Our goal was to select a test that could point out possible trends in the smallest possible release window, so that there is not a very large accumulation of coupling issues. Considering these limitations, the paper points out both the root cause analysis of what is causing the minimum number of data point variation and the possible review of the trend test method as future work.

Being a tool to support decision-making, we believe that the impact of the occurrence of false positives (alerting architectural degradation when there is none) is reduced since the responsible development team can assess before acting. In the case of false negatives, the problem is the method ignoring the possible existence of an architectural problem and the team not being alerted in time. Therefore, due to the generalization problem, we cannot guarantee a lack of false negatives in the application of the method in real cases.

Considering we are currently not aware of methods performing this type of analysis, the use of this first version of SYMBIOTE should have more advantages than disadvantages since its role is to alert about common coupling problems and that can be detected in cases similar to those used in this work. If such coupling problems remain hidden in software evolution, they can increase maintenance costs in the long term.

Conclusions

Our research goal was to investigate if we could propose an effective method to analyze a coupling metrics suite over time to support the identification of signs of architectural degradation. To achieve this, we developed the SYMBIOTE method to monitor coupling in microservice-based architectures. First, we selected metrics from the literature and defined a set of 4 metrics. After, we decided to use dynamic analysis to extract the metrics from the applications. Subsequently, we conducted an experiment to verify the behavior of the metrics and the analysis procedure used in this experiment fostered the analysis (last step) in the SYMBIOTE method. Finally, we applied the method in a real case: a CD tool named Spinnaker.

We have shown the strategy of extracting metrics dynamically is feasibly through a proof-of-concept in a demo application and then again in a real application. Besides, dynamic analysis is also more accurate and does not require code instrumentation. The real-case evaluation has shown that trend analysis by release interval can be a powerful tool for software development teams to monitor how software coupling evolves in certain periods. Also, we realized there is a strong relationship between the changes impacting coupling (which can be common causes of architectural degradation) and the changes in

trends of the metrics suite. The use of the four metrics together has more chance to discover coupling problems since three different aspects of coupling (afferent, efferent, and density of connections) are covered.

Our work contributed to the empirical validation of metrics proposed in the literature (ADCS and SCF); the proposition and validation of new coupling metrics for microservices (SID and SDD based on the AIS and ADS metrics in the literature); the proposal for a dynamic approach to collect metrics at runtime; the experimental results with artificially generated data indicate the potential of at least three out of four metrics (SID, SDD, and ADCS showed good results); the application of the method in a real case revealed that SYMBIOTE can be very useful for decision-making, thus assisting Software Engineering professionals in quality monitoring tasks. Therefore, we understand this work presented some contribution to research in the field of microservice architecture, which is still in its initial stages.

Future work includes developing a tool to support the SYMBIOTE, to evolve the method in terms of selected metrics, to differentiate weak and strong dependencies through weights according to the number of requests between services, to include change-point analysis in the evolution series, to capture other types of dependencies (for example indirect dependencies), to evolve the method to allow customizing architectural degradation alerts, and to validate the method's usefulness with experienced professionals in microservice architecture.

Abbreviations

MSA: Microservice-based applications; SBS: Service-Based Systems; AIS: Absolute Importance of the Service; ADS: Absolute Dependency of the Service; SCF: Service Coupling Factor; ADCS: Average number of directly connected service; RCS: Relative Coupling of Service; RIS: Relative Importance of Service; SSC: System's Service Coupling; ACS: Absolute Criticality of the Service; SIY :Services Interdependence in the System; SID: Service Importance Distribution; SDD: Service Dependency Distribution; SFN: Scale-Free Networks; REST: Representational State Transfer

Acknowledgements

Not applicable

Authors' contributions

Both authors had an equal input in this paper. Both authors read and approved the final manuscript.

Funding

The work is supported by the Brazilian Agricultural Research Corporation (Embrapa) and the National Council for Scientific and Technological Development (CNPq, Process Number 407478/2018-3).

Availability of data and materials

The source code, experimental results, and all generated data supporting the conclusions of this article are at [32].

Declarations

Competing interests

The authors declare that they have no competing interests.

Received: 8 March 2021 Accepted: 29 October 2021
Published online: 11 December 2021

References

1. Shahin M, Zahedi M, Babar MA, Zhu L (2018) An empirical study of architecting for continuous delivery and deployment. Empir Softw Eng 24:1–48
2. Lewis J, Fowle M (2014) Microservices - a definition of this new architectural term. http://martinfowler.com/articles/microservices.html. Accessed 01 Nov 2018
3. Bogner J, Fritzsch J, Wagner S, Zimmermann A (2018) Limiting technical debt with maintainability assurance – an industry survey on used techniques and differences with service- and microservice-based systems. In: 2018 IEEE/ACM International Conference on Technical Debt (TechDebt). Association for Computing Machinery, New York. pp 125–133
4. de Silva L, Balasubramaniam D (2012) Controlling software architecture erosion: a survey. J Syst Softw 85(1):132–151

5. Chen L (2018) Microservices: architecting for continuous delivery and devops. In: 2018 IEEE International Conference on Software Architecture (ICSA). IEEE Computer Society, Los Alamitos. pp 39–397
6. Dragoni N, Giallorenzo S, Lafuente, AL, Mazzara M, Montesi F, Mustafin R, Safina L (2017) Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering. Springer International Publishing, Cham. pp 195–216
7. Engel T, Langermeier M, Bauer B, Hofmann A (2018) Evaluation of microservice architectures: a metric and tool-based approach. In: International Conference on Advanced Information Systems Engineering. Springer International Publishing, Cham. pp 74–89
8. Jamshidi P, Pahl C, Mendonca NC, Lewis J, Tilkov S (2018) Microservices: the journey so far and challenges ahead. IEEE Softw 35(3):24–35
9. Apolinário DRDF, de França BBN (2020) Towards a method for monitoring the coupling evolution of microservice-based architectures. In: Proceedings of the 14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '20). ACM. https://doi.org/10.1145/3425269.3425273
10. Riaz M, Sulayman M, Naqvi H (2009) Architectural decay during continuous software evolution and impact of 'design for change' on software architecture. In: International Conference on Advanced Software Engineering and Its Applications. Springer, Berlin. pp 119–126
11. Binkley AB, Schach SR (1998) Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures. In: Proceedings of the 20th International Conference on Software Engineering. IEEE Computer Society, Los Alamitos. pp 452–455
12. Lindvall M, Tesoriero R, Costa P (2002) Avoiding architectural degeneration: an evaluation process for software architecture. In: Proceedings Eighth IEEE Symposium on Software Metrics. IEEE Computer Society, Los Alamitos. pp 77–86
13. Bogner J, Wagner S, Zimmermann A (2017) Towards a practical maintainability quality model for service-and microservice-based systems. In: Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings. ACM, New York. pp 195–198
14. Alshuqayran N, Ali N, Evans R (2018) Towards micro service architecture recovery: an empirical study. In: 2018 IEEE International Conference on Software Architecture (ICSA). IEEE Computer Society, Los Alamitos. pp 47–4709
15. Perepletchikov M, Ryan C, Frampton K (2005) Comparing the impact of service-oriented and object-oriented paradigms on the structural properties of software. In: OTM Confederated International Conferences "On the Move to Meaningful Internet Systems". Springer, Berlin. pp 431–441
16. Perepletchikov M, Ryan C, Frampton K, Tari Z (2007) Coupling metrics for predicting maintainability in service-oriented designs. In: 2007 Australian Software Engineering Conference (ASWEC'07). IEEE Computer Society, Los Alamitos. pp 329–340
17. Bogner J, Wagner S, Zimmermann A (2017) Automatically measuring the maintainability of service- and microservice-based systems: a literature review. In: Proc. of the 27th Int. Workshop on Software Measurement. ACM, New York. pp 107–115
18. Sousa BL, Bigonha MAS, Ferreira KAM (2019) Analysis of coupling evolution on open source systems. In: Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS '19). Association for Computing Machinery, New York. pp 23–32
19. de Toledo SS, Martini A, Sjøberg DIK (2021) Identifying architectural technical debt, principal, and interest in microservices: a multiple-case study. J Syst Softw 177:110968. https://doi.org/10.1016/j.jss.2021.110968
20. Sampaio AR (2017) Supporting microservice evolution," proc. - 2017. IEEE Int Conf Softw Maint Evol ICSME 2017:539–543
21. Mayer B, Weinreich R (2018) An approach to extract the architecture of microservice-based software systems. In: 2018 IEEE Symposium on Service-Oriented System Engineering (SOSE). IEEE Computer Society, Los Alamitos. pp 21–30
22. Kitajima S, Matsuoka N (2017) Inferring calling relationship based on external observation for microservice architecture. In: Intl. Conf. on Service-Oriented Computing. Springer International Publishing, Cham. pp 229–237
23. Pinto AF, Terra R, Guerra E, São Sabbas F (2017) Introducing an architectural conformance process in continuous integration. J Univ Comput Sci 23(8):769–805
24. Ntentos E, Zdun U, Plakidas K, Meixner S, Geiger S (2020) Assessing architecture conformance to coupling-related patterns and practices in microservices. In: European Conference on Software Architecture. Springer International Publishing, Cham. pp 3–20
25. Jenkins S, Kirk SR (2007) Software architecture graphs as complex networks: a novel partitioning scheme to measure stability and evolution. Inf Sci 177(12):2587–2601
26. Ma S-P, Fan C-Y, Chuang Y, Lee W-T, Lee S-J, Hsueh N-L (2018) Using service dependency graph to analyze and test microservices. In: 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), vol. 2. IEEE Computer Society, Los Alamitos. pp 81–86
27. Zhou X, Peng X, Xie T, Sun J, Xu C, Ji C, Zhao W (2018) Benchmarking microservice systems for software engineering research. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE '18). Association for Computing Machinery, New York. pp 323–324
28. Rahman DMI (2019) Taibi: a curated dataset of microservices-based systems. In: Joint Proceedings of the Summer School on Software Maintenance and Evolution. CEUR-WS, Tampere
29. Vasa R, Lumpe M, Branch P, Nierstrasz O (2009) Comparative analysis of evolving software systems using the gini coefficient. In: 2009 IEEE International Conference on Software Maintenance. IEEE Computer Society, Los Alamitos. pp 179–188
30. Adnan SD (2019) Software evolution on azureus bit torrent software: a study on growth and change analysis. J Eng Sci Technol 14:430–447
31. Xu K (2003) How has the literature on gini's index evolved in the past 80 years?. Dalhousie University, Economics Working Paper, Halifax

32. Microservices Graph Generation tool and experiment results. https://github.com/daniel-apolinario/microservices-graph. Accessed 02 Mar 2021
33. Barabási A-L, Albert R (1999) Emergence of scaling in random networks. Science 286(5439):509–512
34. Wheeldon R, Counsell S (2003) Power law distributions in class relationships. In: Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation. IEEE Computer Society, Los Alamitos. pp 45–54
35. Potanin A, Noble J, Frean M, Biddle R (2005) Scale-free geometry in OO programs. Commun ACM 48(5):99–103
36. Wen L, Dromey RG, Kirk D (2009) IEEE Trans Syst Man Cybern B (Cyberne) 39(4):845–854
37. Šubelj L, Bajec M (2012) Software systems through complex networks science: review, analysis and applications. In: Proceedings of the First International Workshop on Software Mining. Association for Computing Machinery, New York. pp 9–16
38. Jing L, Keqing H, Yutao M, Rong P (2006) Scale free in software metrics. In: 30th Annual International Computer Software and Applications Conference (COMPSAC'06), vol. 1. IEEE. pp 229–235
39. Azadi U, Arcelli Fontana F, Taibi D (2019) Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE Computer Society, Los Alamitos. pp 88–97
40. Taibi D, Lenarduzzi V, Pahl C (2020) Microservices anti-patterns: a taxonomy. In: Microservices. Springer International Publishing, Cham. pp 111–128
41. Bogner J, Boceck T, Popp M, Tschechlov D, Wagner S, Zimmermann A (2019) Towards a collaborative repository for the documentation of service-based antipatterns and bad smells. In: 2019 IEEE International Conference on Software Architecture Companion (ICSA-C). IEEE Computer Society, Los Alamitos. pp 95–101
42. de Freitas Apolinário DR, de França BBN Microservices dependencies graph generation tool. https://doi.org/10.5281/zenodo.5101943
43. Richardson C Microservices.io website. https://microservices.io/. Accessed 15 Feb 2020
44. Law AM, Kelton WD, Kelton WD (2013) Simulation modeling and analysis, vol. 5. McGraw-Hill Education, New York
45. Build and deploy Java Spring Boot microservices on Kubernetes. https://github.com/IBM/spring-boot-microservices-on-kubernetes. Accessed 02 Mar 2021
46. Graphviz - dot language site. https://www.graphviz.org/doc/info/lang.html. Accessed 19 Mar 2020
47. Aderaldo CM, Mendonça NC, Pahl C, Jamshidi P (2017) Benchmark requirements for microservices architecture research. In: 2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE). pp 8–13. https://doi.org/10.1109/ECASE.2017.4
48. Márquez G, Astudillo H (2018) Actual use of architectural patterns in microservices-based open source projects. In: 2018 25th Asia-Pacific Software Engineering Conference (APSEC). pp 31–40. https://doi.org/10.1109/APSEC.2018.00017
49. Spinnaker website. https://spinnaker.io/. Accessed 25 Oct 2020
50. Symbiote public repository. https://github.com/daniel-apolinario/symbiote. Accessed 02 Mar 2021

Publisher's Note