2022 9th International Conference on Future Internet of Things and Cloud (FiCloud) | 978-1-6654-9350-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/FiCloud57274.2022.00042

A Graph-based Solution to Deal with Cyclic Dependencies in Microservices Architecture

Hassan Farsi, Driss Allaki, Abdeslam En-nouaary, Mohamed Dahchour
Institut National des Postes et Télécommunications, Rabat, Morocco
{farsi.hassan, d.allaki, abdeslam, dahchour}@inpt.ac.ma

①介绍微服务

***Abstract*— Microservices represents the most suitable architectural style to create cloud-native applications. Microservices is implemented by distributing the development to many small independent teams. Each team can make progress more independently of the others. That is what gives big organizations the ability to properly scale development in order to ensure the very demanding needs of their customers. However, it is actually quite difficult to achieve this desired "decoupled independence" of services since many anti-patterns can lead to opposite results. One of these classical anti-patterns is called "Cyclic Dependencies". In this paper, we provide a graph-based proposal that focuses on laying the foundations for solutions aiming to assist architects at design time to automatically detect the Cyclic Dependencies anti-pattern in microservices.**

②存在的困难

③提出的解决方案

Keywords—Cloud-Native; Microservices Architecture; Anti-Patterns; Cyclic Dependencies; Graph Theory.

I. Introduction

①现状

Microservices is one of today's most popular architectural approaches. It is basically a distributed system architecture that have already shown its effectiveness in large and engaging systems. This is the main reason why it is used by many of the most successful web companies in industry to build cloud-native applications.

②特点

The microservices architecture is composed of small services, focused on doing only one task (and doing it well). It is not about breaking a large system into arbitrary pieces, but about autonomous services that are aligned with the problem domain, allowing teams to make independent progress, without being constrained with dependencies on other teams or services. Microservices should be loosely coupled and independently deployable [1]. This enables rapid growth, the main value that organizations generally seek.

③困难

However, it is actually quite difficult to achieve this decoupled independence. In fact, microservices demand a high level of design sophistication to avoid a wrong decomposition that leads to tightly coupled and dependent services that are built, tested and deployed together. This way of doing things generally leads to a distorted architectural style that does not provide none of the benefits we are looking for when using microservices.

④明确任务

In this work, we choose to deal with a recurrent problem in microservices. It is one, among others, that can cause the loss of this "*decoupled independence*" of services. More precisely, we are pointing fingers at "*Cyclic Dependencies*", a classical microservices anti-pattern. *Cyclic dependencies* are services that depend on each other in order to function properly.

⑥目标

Our aim in this paper is to provide a strong solution that lays the theoretical foundations of industrial tools. The tools that can assist architects and developers automatically detect, at design time, the *Cyclic Dependencies* anti-pattern.

⑦文章结构

This paper is organized as follows. Section II presents the specific problem we are aiming to solve, namely the "*Cyclic Dependencies*" anti-pattern. Section III introduces our proposed graph-based solution that rely on transforming a microservices architecture into a graph in order to apply a set of graph algorithms that provide enough insights to detect the targeted anti-pattern. Section IV presents and discusses the results obtained when applying our solution to well selected case studies. Section V discusses related work. And the final section concludes the paper and announces the future work.

II. Problem Statement

①和引言、摘要过渡，提出概念

As stated before, finding the right set of services is very challenging since there is no concrete and well-defined way for decomposing a system into services. To make matters worse, incorrectly decomposing a system generates a lot of architectural anti-patterns that they will prevent getting the best of this architectural style. For instance, it is very likely to end up with a microservices application in which some services are depending on each other in a cyclic way as illustrated in Figure 1. This anti-pattern is commonly called in literature "*Cyclic Dependencies*" [2], [3], [4], [5] and [6].

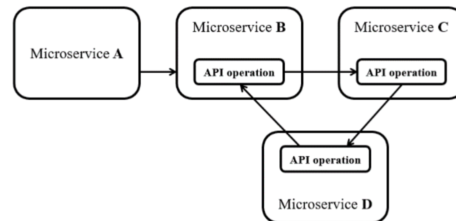


Fig. 1. Cyclic Dependencies

②问题

Cyclic dependencies occur when the output of one service should trigger an input of another. They are usually manifested through frequent communication and direct calls between services. In some cases, they can also take the form of HTTP requests in callbacks (forming a closed loop) [4].

The problem with this anti-pattern is that the system's services heavily lose their independence. More accurately, the deployment of a single microservice will depend on deploying its coupled microservices (*i.e. no independent deployments*).

试用水印

978-1-6654-9350-5/22/$31.00 ©2022 IEEE
DOI 10.1109/FiCloud57274.2022.00042

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.

Likewise, a failure in one of the cyclic-dependent microservices will cause a failure in the others in a cascading way (*i.e. the domino effect*).

Also, it is worth mentioning that the network complexity increases if there is *Cyclic Dependencies* in the system.

From another side, *Cyclic Dependencies* make reasoning about the business processes more difficult since making changes to the involved microservices becomes much harder (*i.e. complexity of maintenance*).

As a result, wasting all these keys defining characteristic of microservices means losing almost all the advantages promised by this architectural style.

③目标

In this paper, we intend to tackle this anti-pattern that have a high negative impact on the development team productivity.

III. The Proposed Graph-Based Solution

①步骤

Our detection approach is based on two major steps. The first one is about generating a graph that reflects the distributed architecture of our system. The graph data are collected at design time, where each vertex represents a single microservice and each edge acts as a labeled call between these microservices.

On the other hand, the main idea of the second step is to use the graph theory algorithms in order to analyze and get useful insights from the graph obtained in the first step to finally detect the anti-pattern we are interested in.

②优点

Using graphs as a formal representation for microservices or any distributed system is a very intuitive and logical idea; since a microservices architecture is consisting of a set of connected components (*i.e.* services) that could be easily translated into a graph.

Using a graph automatically opens the doors to several opportunities manifested in the application of graph algorithms. It is one of the major aspects of graph science that makes graph a strong mathematical tool for connecting data analysis.

A. Graph Algorithms

③介绍图算法

In graph science, we use graph algorithms to analyze, understand, and get insights about a graph, and eventually predict and decide changes to make. Algorithms are basically sets of basic mathematical techniques used to fulfil a certain purpose. They are mainly executed to parse a graph in order to manipulate or get hidden intuition about the structure of the system.

In general, graph algorithms are organized into multiple categories. Each category serves a common purpose. For instance, the algorithms that study link distances between nodes are grouped by the *Path Finding* cluster, and those that treat the importance of a node goes into the *Centrality* set of algorithms. The detection of nodes which tend to each other is handled by the *Community Detection* algorithms.

To mention that other classifications of graph algorithms exist. For example, *Searching*, *Sorting*, *Backtracking*, and other customized algorithms dedicated to solve different other problems.

④本文中的图算法

In this work, we will not use all the available graph algorithms, as we will focus only on those helping detecting *Cyclic Dependencies*.

Based on our research and experimentation, we can confirm that the *Strongly Connected Components (SCC)* [7] algorithm is one of the most important graph algorithms that enables effective *Cyclic Dependencies* detection. *SCC* is used with directed graphs in order to detect *(as the name indicates)* strongly connected groups of nodes. Components are considered strongly connected if each node can reach all the others directly or through another node.

To model the graph representing a microservices system as well as the *SCC* algorithm execution, we consider the following:

$$G = (V, E)$$

Where G is a graph having V as the set of vertices and E as the set of edges.

Concerning the *SCC* algorithm, we will use *Out(u)* to represent the set of outgoing paths from a node *(u)* and *In(u)* as the incoming paths to the node *(u)*. The *SCC* formula could be then presented as:

$$\forall\, u, v \in V, \exists\, p \in Out(v),\ p' \in Out(u):$$
$$p \in In(u),\ \ p' \in In(v)$$

SCC belongs to the community detection type of algorithms. Therefore, it is mainly used to detect tight clusters and groups. However, what is really interesting about *SCC* is its capability of detecting cycles or processes that can deteriorate or impasse a system. The algorithm capability of detecting cyclic relationships gives birth to the idea of detecting *Cyclic Dependencies* in the context of microservices.

B. The Followed Process

⑤承上启下

Considering all what was presented above, we follow up the process illustrated in Figure 2 to proof the existence of the *Cyclic Dependencies* anti-pattern in some specific case studies.

6流程

As a first step, we transform the microservices architectures of the chosen case studies into formal and well-constructed graph representations *(i.e.* a graph that represents all the necessary elements of the architecture such as the services, the APIs, and the connection between them*)*. When the graph is ready, the next step is the execution of the *Strongly Connected Components (SCC)* algorithm in order to extract metrics and get the desired results. After that, the final step comes to analyze the obtained results and come out with conclusions.

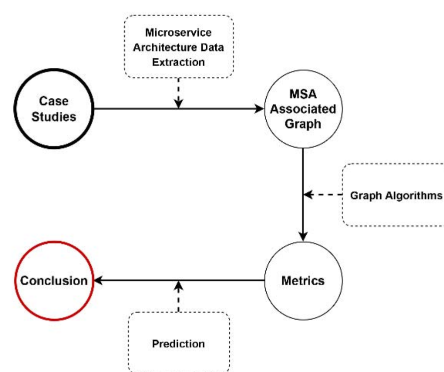


Fig. 2. The followed process

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.

IV. Proof of Concept

To illustrate the aforementioned graph-based solution, we propose to experiment this approach on two different case studies where we will try to show the efficiency of detecting the *Cyclic Dependencies* anti-pattern using this technique.

A. Case Studies Presentation

To assess our approach, we conducted a literature review in order to select some reliable case studies to work on. We were aware that the open-source projects are not the most appropriate ones to use for this experiment. Since they are usually not big enough to increase the chances of containing a *Cyclic Dependencies* anti-pattern. At the same time, this kind of projects is more accessible and it is generally what researchers use in academic and scientific experimentation.

1. Case Study 1: Lakeside Mutual project

Based on studying many papers and open source microservices projects, we choose to work on one of the case studies proposed by Genfer and Zdun [8] which is the *Lakeside Mutual* microservices application [9].

Lakeside Mutual is an insurance company microservices application decomposed into services using Domain Driven Design [10]. The latest version of the application consists of 4 backend services (11 in total). In this work, we used the Spring Term 2020 release of the *Lakeside Mutual* project (from March 2020) [11] as it is the last known version of the application that contains the *Cyclic Dependencies* anti-pattern between its APIs. According to Genfer and Zdun [8] there was a major refactoring of the application that had led to eliminate this anti-pattern in the recent versions.

Figure 3 illustrates the concerned part of the microservices architecture which contains a *Cyclic Dependencies* anti pattern emerging from cyclic calls between API operations.

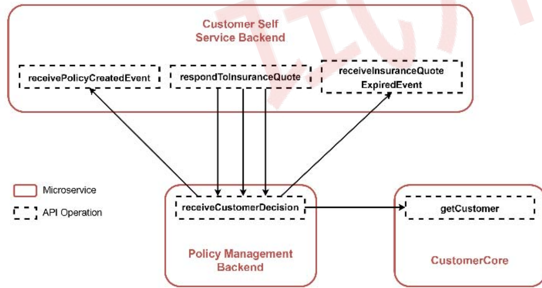


Fig. 3. Cycle dependency in the Lakeside Mutual Project, spring-term-2020 version [8].

The *Cyclic Dependencies* anti pattern can be clearly noticed between the *Customer Self Service Backend* microservice (called hereafter *MS1*) and the *Policy Management Backend* microservice (called hereafter *MS2*).

Actually, the "*respondToInsuranceQuote*" operation from *MS1* calls the "*receiveCustomerDecision*" operation from *MS2*. This later in turn calls "*receivePolicyCreatedEvent*" operation from *MS1* again.

Likewise, the "*respondToInsuranceQuote*" operation from *MS1* calls the "*receiveCustomerDecision*" operation from *MS2* to trigger a new call of the "*receiveInsuranceQuoteExpiredEvent*" *MS1* operation.

As a result, *MS1* and *MS2* suffer from a *Cyclic Dependencies* anti-pattern making them strongly dependent and then difficult to maintain, difficult to separately deploy and ensuring reliability and resiliency of the whole system is very hard.

2. Case Study 2: Typical ecommerce application ("Customized" example)

The second case study used to proof our concept is a customized architecture inspired by different ecommerce projects available in the literature. The application contains 5 microservices, namely "*Orders*", "*Account*", "*ShippingAPI*", "*Notification*" in addition to an *API Gateway*. These microservices generate *Cyclic Dependencies* when they are communicating.

Figure 4 presents the microservices mentioned above in addition to the different communication links between them. It is worth mentioning that the granularity level of this example is intentionally higher than the first one.

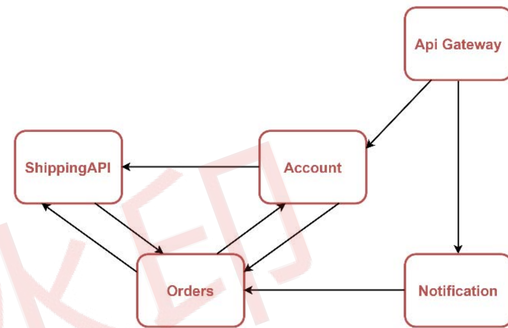


Fig. 4. Costumized part of a microservices architecture.

This case study exposes three main *Cyclic Dependencies* within its structure. According to Al-mutawa et al [12] two of them are *Tiny*, which means there is cycles between two pairs of nodes, and one of them is having a shape of *Circle* where there are more than two nodes included.

The first tiny cycle is between *Account* and *Orders*, when the user needs to make or get information about an order, and when the *Order* service needs information about the account that made the order.

The second one is about the relationships between *Order* service and *ShippingAPI*. The shipping needs to know about the order details and the state of the order is changed based on the shipping progress.

The third one is when the *Account* service track the shipping of an order. This relationship makes a circular dependency between *Accounts*, *ShippingAPI*, and *Order* services.

B. Graph Construction

To analyze a given graph, we must use a dedicated platform. Based on the comparative study carried out by Fernandes and Bernardino [13], we choose to work with the Neo4j database as it is an open-source, popular and easy tool to use for storing and analyzing connected data of a graph.

Neo4j has a lot of advantages including the smooth user experience, the reliability of the platform and the strong performance. Neo4j has a powerful platform named Neo4j

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.

Graph Data Science [14], which is a tool that implements most of data analytics, machine learning, and graph algorithms in order to help users understand and answer critical questions about the connections present within a graph.

Neo4j does not support the nested nodes (a node within a node). Thus, for the *Lakeside Mutual* case study, it is not possible to represent APIs operations inside each microservice as nodes. In contrast, it is possible to represent microservices as nodes and API operations as edges labelled with the operation name as illustrated in Figure 5. Doing that, we can visually identify two tiny *Cyclic Dependencies* between two microservices resulting from several API calls.

In turn, Figure 6 illustrates the obtained graph for the considered part of the customized typical ecommerce application.

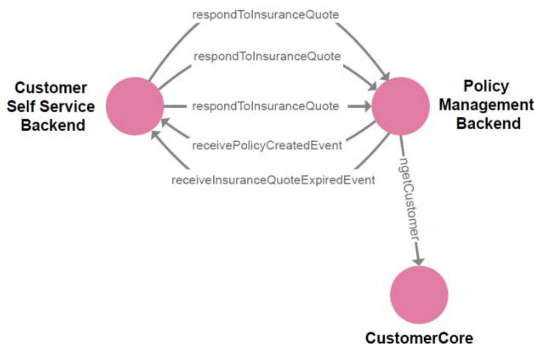


Fig. 5. Lakside Mutual project graph.

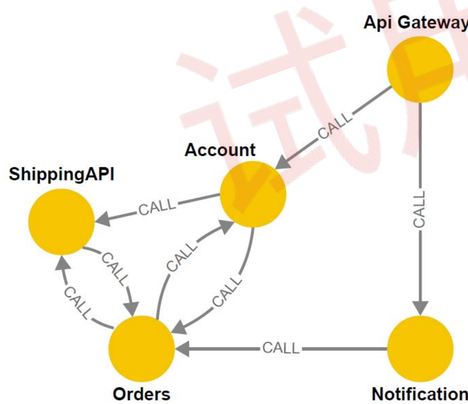


Fig. 6. Costumized ecommerce project graph.

In what follows, we intend to use Neo4j to store and generate the necessary graph for the two case studies and then use the Graph Data Science platform to execute the *Strongly Connected Components (SCC)* algorithm in order to detect the *Cyclic Dependencies*.

C. Obtained Results

The execution of the *Strongly Connected Components (SCC)* on the two projects provides the results indicated in Table 1.

⑤成果

These results show that each of the two case studies present some form of *Cyclic Dependencies*. The *SCC* algorithm divided the graph into multiple strongly components, each component with size higher that one, is by definition (Section III - A) contains nodes (microservices) that form a *Cyclic Dependencies* anti-pattern.

TABLE I. STRONGLY CONNECTED COMPONENTS, EXPERIMENTAL RESULTS.

Case Studies	Strongly Connected Community[a]	Size	Microservices
Lakeside Mutual	0	2	Customer Self Service Backend, Policy Management Backend
	2	1	CustomerCore
Costumized ecommerce App	2	3	Account, Orders, ShippingAPI
	0	1	Notification
	4	1	Api Gateway

[a] Each community has an id.

The other nodes remain in a single community as they do not achieve the strongly connected component criteria. The algorithm detects a tiny *Cyclic Dependencies* in the *Lakeside Mutual* project that includes two services. Additionally, it detects a crucial form of dependency between the three services in the customized microservices case study.

Figures 7 and 8 show the cycles location in the concerned graph using Neo4j Neuler [15].

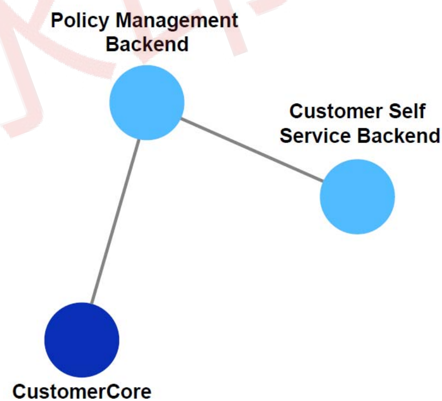
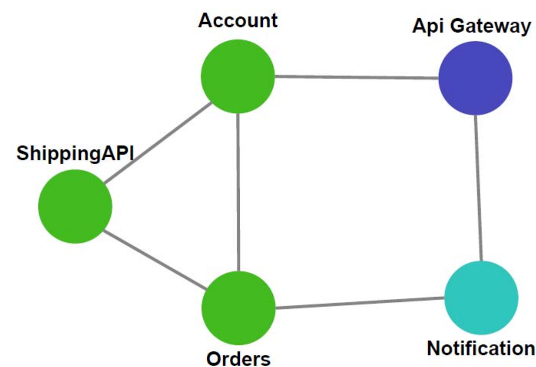


Fig. 7. Lakside Mutual project cyclic dependencies graph.



试用水印

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.

Fig. 8. Costumized ecommerce project Cyclic Dependencies graph.

The comparison between Figures 5, 6 and the results gathered in Table1 and Figures 7, 8, shows that regardless of the cyclic shape, our graph-based approach has successfully detected the *Cyclic Dependencies* anti-pattern within two microservices architecture case studies.

⑥缺陷

Although these results of the *SCC* algorithm are very promising, we can clearly see that it does not come up with enough details about the detected cycles. For instance, in the customized microservices application, we can visually recognize that there are three different *Cyclic Dependencies* (*i.e.* two tiny cycles and one having a shape of circle). However, the *SCC* algorithm only highlighted the nodes (microservices) concerned by all these dependencies; without explicating or mentioning that these dependencies are composed of two tiny cycles (*Account*, *Orders*) and (*Orders*, *Shipping* API) in addition to a circle cycle between *Orders*, *ShippingAPI* and *Account*.

V. Related Work

①称述研究状况

Over the last few years, dealing with anti-patterns in microservices architecture has been a priority research investigation to many researchers. As a consequence, several contributions have been proposed to deal with this issue. Hereafter, we present some of the recent works found in literature that superficially broach or deeply focus on "*Cyclic Dependencies*" anti-pattern.

②相关研究

Engel et al. [16] develop a tool that captures architecture data from traces of communication to evaluate the architectural design based on a set of identified principles and metrics that include *Cyclic Dependencies*.

Esparrachiari et al. [17] propose modeling infrastructure as a Directed Acyclic Graph (DAG) in addition to implementing dependency controls to track dependencies.

Shang-Pin Ma et al. [18] propose a new approach called GSMART that enables four main functionalities. The first one is the automatic generation of a service dependency graph to visualize and analyze dependency relationships between microservices as well as between scenarios and services. The second concerns detecting *Cyclic dependencies* references. The third and fourth ones are respectively, selecting regression test cases and retrieving existing microservices.

Walker et al. [19] develop a tool called *MSANose* to detect *Cyclic Dependencies* in addition to other microservices code smells using bytecode and/or source code analysis throughout the development process and before the application is deployed.

Gaidels et al. [20] apply graph algorithms on a specific microservices example (online banking) using the Neo4j set of tools to assess the quality of the underlying architecture.

Gamage et al. [21] provide an automated tool-based solution that adopt graph concepts to provide developers with adequate metrics along with dependency graphs for better visualization.

Ntentos et al. [22] and [23] aim to provide the foundations for an automated approach for assessing conformance to coupling related patterns and practices for microservices, in addition to detecting the possible violations.

Genfer and Zdun [8] suggest a new approach for identifying and evaluating domain-based *Cyclic Dependencies* based on modular and reusable source code detectors. These detectors, reconstruct an architecture model that serves to derive a set of architectural metrics for detecting and classifying *Cyclic Dependencies*.

③和本文方案对比

In this work we have studied the *Cyclic Dependencies* anti-pattern detection using a graph-based approach considered as a potential reference work for tools seeking that goal. The majority of the works cited above have mentioned *Cyclic Dependencies* as a small part of their experiment, without giving any details on how this anti-pattern was detected. As opposite to that, we focused in this work on the *Cyclic Dependencies* as the main anti-pattern to deal with.

On the other hand, it is true that some few papers in literature have also mentioned the *SCC* algorithm as a way of detecting *Cyclic Dependencies*. Unfortunately, these papers did not explain, analyze or study the problem, the approach (using the *SCC* algorithm) and the obtained results in details. In our case, we clearly showed the *SCC* algorithm capability of detecting the *Cyclic Dependencies* anti-pattern. In addition to that, we have also mentioned its main limitation as, when used alone, it does not provide much internal details about the detected anti-pattern in the case of a large and complex cycle of dependencies.

④在此背景下引出本文的案例研究

One last thing to mention is that the literature is not very rich of microservices projects that include *Cyclic Dependencies*. Nevertheless, we have based our demonstration on two different case studies that expose the anti-pattern in two levels of granularity. Dependencies from a service level, and the dependencies from the API operations level.

VI. Conclusion and Future Work

①总结

In this paper, we have presented a graph-based proposal that detect the *Cyclic Dependencies* anti-pattern in microservices architecture. This solution is dedicated to create helpful tools to assist architects achieving services loose coupling in microservices architecture. Then, it is up to these architects to take the appropriate actions to refactor their system in order to solve and avoid the anti-pattern fallouts and consequences.

②描述内容

We have applied the *Strongly Connected Components (SCC)* algorithm on two different case studies in two levels of abstraction; one on service-level and the other on API operations level. We have demonstrated the efficiency of using the *SCC* algorithm in detecting *Cyclic Dependencies*. At the same time, we have also highlighted the limitations of such algorithm as it does not provide enough details about the detected cycles.

③展望

To extend and improve this work, we intend to combine the *SCC* algorithm with other graph algorithms, tools and approaches that we will design and implement to provide further internal information concerning the details of the detected cycles.

References

[1] FOWLER, Martin et LEWIS, James: Microservices (2014). https://martinfowler.com/articles/microservices.html (Accessed 29 April 2022).

[2] AZADI, Umberto, FONTANA, Francesca Arcelli, et TAIBI, Davide. Architectural smells detected by tools: a catalogue proposal. In : 2019

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.

IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE, 2019. p. 88-97.

[3] TAIBI, Davide, LENARDUZZI, Valentina, et PAHL, Claus. Microservices anti-patterns: A taxonomy. In : Microservices. Springer, Cham, 2020. p. 111-128.

[4] TIGHILT, Rafik, ABDELLATIF, Manel, MOHA, Naouel, et al. On the study of microservices antipatterns: A catalog proposal. In : Proceedings of the European Conference on Pattern Languages of Programs 2020. 2020. p. 1-13.

[5] TAIBI, Davide et LENARDUZZI, Valentina. On the definition of microservice bad smells. IEEE software, 2018, vol. 35, no 3, p. 56-62.

[6] PIGAZZINI, Ilaria, FONTANA, Francesca Arcelli, LENARDUZZI, Valentina, et al. Towards microservice smells detection. In : Proceedings of the 3rd International Conference on Technical Debt. 2020. p. 92-97.

[7] NEEDHAM, Mark et HODLER, Amy E. Graph Algorithms: Practical Examples in Apache Spark and Neo4j. O'Reilly Media, 2019.

[8] GENFER, Patric et ZDUN, Uwe. Identifying Domain-Based Cyclic Dependencies in Microservice APIs Using Source Code Detectors. In : European Conference on Software Architecture. Springer, Cham, 2021. p. 207-222.

[9] Microservice-API-Patterns / LakesideMutual. url: https://github.com/Microservice-API-Patterns/LakesideMutual (Accessed 29 April 2022).

[10] EVANS, Eric et EVANS, Eric J. Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional, 2004.

[11] Microservice-API-Patterns / LakesideMutual. url: https://github.com/Microservice-API-Patterns/LakesideMutual/tree/spring-term-2020 (Accessed 29 April 2022).

[12] AL-MUTAWA, Hussain A., DIETRICH, Jens, MARSLAND, Stephen, et al. On the shape of circular dependencies in java programs. In : 2014 23rd Australian Software Engineering Conference. IEEE, 2014. p. 48-57.

[13] FERNANDES, Diogo et BERNARDINO, Jorge. Graph Databases Comparison: AllegroGraph, ArangoDB, InfiniteGraph, Neo4J, and OrientDB. In : Data. 2018. p. 373-380.

[14] Neo4j Graph Data Science. url: https://neo4j.com/product/graph-data-science/ (Accessed 29 April 2022).

[15] NEuler: No-code graph algorithms. url: https://neo4j.com/developer/graph-data-science/neuler-no-code-graph-algorithms/ (Accessed 29 April 2022).

[16] ENGEL, Thomas, LANGERMEIER, Melanie, BAUER, Bernhard, et al. Evaluation of microservice architectures: a metric and tool-based approach. In : International Conference on Advanced Information Systems Engineering. Springer, Cham, 2018. p. 74-89.

[17] ESPARRACHIARI, Silvia, REILLY, Tanya, et RENTZ, Ashleigh. Tracking and Controlling Microservice Dependencies: Dependency management is a crucial part of system and software design. Queue, 2018, vol. 16, no 4, p. 44-65.

[18] MA, Shang-Pin, FAN, Chen-Yuan, CHUANG, Yen, et al. Graph-based and scenario-driven microservice analysis, retrieval, and testing. Future Generation Computer Systems, 2019, vol. 100, p. 724-735.

[19] WALKER, Andrew, DAS, Dipta, et CERNY, Tomas. Automated code-smell detection in microservices through static analysis: a case study. Applied Sciences, 2020, vol. 10, no 21, p. 7800.

[20] GAIDELS, Edgars et KIRIKOVA, Marite. Service Dependency Graph Analysis in Microservice Architecture. In : International Conference on Business Informatics Research. Springer, Cham, 2020. p. 128-139.

[21] GAMAGE, Isuru Udara Piyadigama et PERERA, Indika. Using dependency graph and graph theory concepts to identify anti-patterns in a microservices system: A tool-based approach. In : 2021 Moratuwa Engineering Research Conference (MERCon). IEEE, 2021. p. 699-704.

[22] NTENTOS, Evangelos, ZDUN, Uwe, PLAKIDAS, Konstantinos, et al. Assessing architecture conformance to coupling-related patterns and practices in microservices. In : European Conference on Software Architecture. Springer, Cham, 2020. p. 3-20.

[23] NTENTOS, Evangelos, ZDUN, Uwe, PLAKIDAS, Konstantinos, et al. Semi-automatic feedback for improving architecture conformance to microservice patterns and practices. In : 2021 IEEE 18th International Conference on Software Architecture (ICSA). IEEE, 2021. p. 36-46.

试用水印

Authorized licensed use limited to: HANGZHOU DIANZI UNIVERSITY. Downloaded on November 16,2022 at 07:23:03 UTC from IEEE Xplore. Restrictions apply.