Automated Microservice Code-Smell Detection

Andrew Walker, Dipta Das, and Tomas Cerny

Baylor University, One Bear Place, Waco, Texas, 76706
tomas_cerny@baylor.edu

Abstract. Microservice Architecture (MSA) is rapidly taking over modern software engineering and becoming the predominant architecture of new cloud-based applications (apps). There are many advantages to using MSA, but there are many downsides to using a more complex architecture than a typical monolithic enterprise app. Beyond the normal bad coding practices and code-smells of a typical app, MSA specific code-smells are difficult to discover within a distributed app. There are many static code analysis tools for monolithic apps, but no tool exists to offer code-smell detection for MSA-based apps. This paper proposes a new approach to detect code smells in distributed apps based on MSA. We develop an open-source tool, MSANose, which can accurately detect up to eleven different types of MSA specific code smells. We demonstrate our tool through a case study on a benchmark MSA app and verify its accuracy. Our results show that it is possible to detect code-smells within MSA apps using bytecode and or source code analysis throughout the development or before deployment to production.

Keywords: Microservice · Cloud-computing · Code Smells · Code-Analysis

1 Introduction

Microservice Architecture (MSA) has become the preeminent architecture in modern enterprise applications (apps) [16]. MSA brings many advantages, which have led to its rise in popularity [4]. The distributed nature of an MSA-based app allows for greater autonomy of developer units. While this provides greater flexibility with respect to faster delivery, improved scalability, and benefits in existing problem domains, it also presents the opportunity for code smells to be more readily created within the app.

Code smells [9] are anomalies within codebases that do not necessarily impact the performance or correct functionality of an app. They are patterns of bad programming practice that can affect a wide range of areas in a program, including reusability, testability, and maintainability. If code smells go unchecked in an MSA-based app, the benefits of using a distributed development process can be mitigated. It is, therefore, important that the code-smells in an app are properly detected and managed [9].

MSA presents a unique situation when it comes to code-smells due to its distributed nature. MSA-specific code smells often focus on inter-module issues rather than an intra-module issue. Traditional code-smell detecting tools cannot

detect code smells between discrete modules, so these issues go unchecked during the development process. This paper shows that when we augment static code analysis to recognize enterprise development constructs, we can effectively detect code smells in MSA distributed apps. A case study demonstrates our approach targeting 11 recently identified code smells for this architecture. Furthermore, we share our open-source prototype code analyzer with the community that can recognize Enterprise Java platform constructs and standards, along with the MSA code smell detector recognizing the 11 code smells targeted in this paper.

The rest of the paper is as follows: Section 2 assesses related work. Section 3 introduces the MSA code smells. Section 4 describes the code analysis of enterprise systems. Section 5 introduces our solution for automatic MSA code smell detection, and Section 6 shares a case study evaluation. We conclude in Section 7.

2 Related Work

Code smells can be defined as "characteristics of the software that may indicate a code or design problem that can make software hard to evolve and maintain" [8]. Code smells do not necessarily impact the performance or correct functionality; they are patterns of bad programming practice that can affect a wide range of areas in a program, including reusability, testability, and maintainability. They can be seen as code structures that indicate a violation of fundamental design principles and negatively impact design quality [21].

Gupta et al. [10] underlines the need to identify and control code smells during the design and development stages to achieve higher code maintainability and quality. If developers are not invested in fixing them, code smells do matter to the overall maintainability of the software. If left unchecked, they can impact the overall system architecture [11]. Code smells can be deceptive and hide the true extent of their 'smelliness' and even carry into further refactorings of the code [6, 11]. Frequently code smells are also related to anti-patterns [18] in an app.

Code-smell correction is clearly a necessary process for developers [20], but it is often pushed aside. It was found that the most prevalent factor towards developers addressing code smells is the importance and relevance of the issue to the task worked on. Peters et al. [17] found that while developers are oftentimes aware of the code smells in their app, they do not care about actively fixing them. Code smells are often fixed accidentally through unrelated code refactoring [9].

Tahir et al. [22] studied how developers discussed code smells in stack exchange sites and found out that these sites work as an informal crowd-based code smell detector. Peers discuss the identification of smells and how to get rid of them in a specific given context. Thus, the question is not only how to detect them but also how to eliminate them in a given context. They found that the most popular smells discussed between developers are also shown to be most frequently covered by available code analysis tools. It is also noted that while Java support is the broadest, other platforms, including C#, JavaScript, C++, Python, Ruby, and PHP, are lacking in support. Concerns were also raised that there is a missing classification for how harmful smells are on a given app.

Some researchers would argue that developers do not have the time to fix all smells. For instance, Gupta et al. [10] identified 18 common code smells and identified the driving power of these code smells to improve the overall code maintainability. The effect is that developers could refactor one of the smells with higher driving power, rather than address all smells in an app, and still significantly improve code maintainability.

An attempt at automatic code smell detection [25], defined an automated code smell detection tool for Java. Since then, the field of code smell detection has continued to grow. Code smell tools have been developed for *high level design* [2], *architectural smells* [15], as well as for *language-specific code smells* [14], measuring not just code smells but also the quality [10] of the app. The field of automatic code smell detection continues to evolve with an ever-changing list of code smells and languages to cover.

It is common to identify code smells in monolithic systems using code-analysis. Anil et al. [13] recently analyzed 24 code smells detection tools (e.g, SpotBugs, PMD, etc.). While the tools correctly mapped the code smells in an app, they are limited to a single codebase, and so they become antiquated as modern software development tends towards MSA.

While extensive research has been done to define and detect code smells in a monolithic app, little has been done for distributed systems [3]. It would be possible for a developer to run code smell detection on each individual module, but this does not address any code smells specific to MSA.

In a distributed environment, in particular MSA, there have been multiple code smells identified. In one study [23], these smells include improper module interaction, modules with too many responsibilities, or a misunderstanding of the MSA. Code smells can be specific to a certain app perspective, including the *communication perspective*, or in the *development and design process* of the app. These smells can be detected manually, which usually requires assessing the app and a basic understanding of the system, but this demands considerable effort from the developers. With code analysis instruments, smells can be discovered almost instantly and automatically with no previous knowledge of the system required. However, we are aware that no tool at present can detect the code anomalies that can exist between discrete modules of an MSA app.

3 Microservice Code Smell Catalogue

In this paper, we reuse the definition of eleven MSA specific code smells from a recent exploratory study by Taibi et al. [23], which used existing literature and interviews with industry leaders to distill and rank these eleven code smells for MSA. The code smells are briefly summarized as follows:

ESB Usage (EU) An Enterprise Service Bus (ESB) [4] is a way of message passing between modules of a distributed app in which one module acts as a service bus for all of the other modules to pass messages on. There are pros and cons to this approach. However, it can become an issue of creating a single point of failure and increasing coupling, so it should be avoided in MSA.

Too Many Standards (TMS) Given the distributed nature of the MSA app, multiple discrete teams of developers often work on a given module, separate from the other teams. This can create a situation where multiple frameworks are used when a standard should be established for consistency across the modules.
Wrong Cuts (WC) WC is when modules are split into their technical layers (presentation, business, and data layers). MSA modules must be split by features, and each fully contains their domain's presentation, business, and data layers.
Not Having an API Gateway (NAG) The API gateway design pattern is for managing the connections between MSA modules. In large, complex systems, this should be used to reduce the potential issues of direct communication.
Hard-Coded Endpoints (HCE) Hardcoded IP addresses and ports to communicate between services. By hardcoding the endpoints, the app becomes more brittle to change and reduces the app's scalability.
API Versioning (AV) All Application Programming Interfaces (API) should be versioned to keep track of changes properly.
Microservice Greedy (MG) This occurs when modules are created for every new feature, and oftentimes, these new modules are too small and do not serve many purposes. This increases complexity and the overhead of the system. Smaller features should be wrapped into a larger MSA module, if possible.
Shared Persistency (SP) When two modules of the MSA app access the same database. This breaks the definition of an MSA where each module should have autonomy and control over its data and database.
Inappropriate Service Intimacy (ISI) One module requesting private data from a separate module. This likewise breaks the MSA definition, where each module should have control over its private data.
Shared Libraries (SL) If modules are coupled with a common library, that library should be refactored into a separate module to reduce the app's fragility by migrating the shared functionality behind a common, unchanging interface. This will make the app resistant ripples from changes within the library.
Cyclic Dependency (CD): Cyclic connection between calls to different modules. This can cause repetitive calls and also increase the complexity of understanding call traces for developers. This is a bad architectural practice for MSA.

To highlight the gap in MSA code smells, we assessed existing state-of-the-art architecture-specific code smell detection tools, compiled in a previous study [3], AI Reviewer, ARCADE, Arcan, Designite, Hotspot Detector, Massey Architecture Explorer, Sonargraph, STAN, and Structure 101 and verified that all are capable of detecting CD and in the case of Arcan, also HCE and SP MSA code smells. We chose these tools as they were compiled to study the existing state of the art of architecture smell detection tools and were shown to meet a minimum threshold of documentation and information about the tool.

4 Code Analysis in Enterprise Systems

The two static code analysis processes, source code analysis and bytecode analysis, ultimately create a representation of the app. This is done through several

processes, including recognizing components, classes, methods, fields or annotations, tokenization, and parsing, which produce graph representations of the code. These include Abstract Syntax Trees (AST), Control-Flow Graphs (CFG), or Program Dependency Graphs (PDG) [19].

Bytecode analysis uses the compiled code of an app. This is useful in uncovering endpoints, components, authorization policy enforcements, classes, and methods. It can also be used to augment or build CFG or AST. However, the disadvantage is that not all languages have a bytecode.

We can also turn to source code analysis [5], which parses through the source code of the app, without having to compile it into an immediate representation. Many approaches exist to do this; however, most tools tokenize the code and construct trees, including AST, CFG, or PDG.

However, limits exist with these representations in encapsulating the complexity of enterprise systems. To mitigate the shortcomings of existing static code analysis techniques on enterprise systems, we must augment existing techniques with an understanding of enterprise standards [7]. A more realistic representation of the enterprise app can be constructed with aid from either source code or bytecode analysis. This primarily includes a tree representation and the detection of the system's endpoints, and the construction of a communication map. These augmented representations and metadata have been successful in other problem domains, including security, networking, and semantic clone detection.

5 Proposed Solution To Detect Code Smells

Our approach is integrable to the software development life-cycle. It uses static-code analysis for fast and easily-integrated reports on the code-smells in an system. To cover the wide variety of possible issues within an MSA app, as well as the different concerns (app, business, and data) issues that the identified smells cover, we must statically analyze a couple of different areas of an app. Our approach specifically involved the Java Enterprise Edition platform because of its rich standards for enterprise development, which include Spring Boot (`https://spring.io/projects/spring-boot`) and Java EE (`https://docs.oracle.com/javaee`). However, alternative standard adoptions exist also for other platforms. Extending for another language would be trivial since we utilize an intermediate representation for analysis, as explained below.

The core of our solution is the creation of a centralized view of the app. To begin with, we individually analyze each MSA module in the app. Once each module is fully analyzed, it can be aggregated into a larger service mesh. Then the full detection can be done on the aggregated mesh.

Our analysis process aims to generate a graph of interaction between the different MSA modules. This involves exploring each module for a connection to another module, usually through a REST API call. The inter-module communications are realized using a two-phase analysis: scanning and matching. In the first phase, we scan each module to list all the REST endpoints and their specification metadata. This metadata contains the HTTP type, path, parameter,

and return type of the REST endpoint. Additionally, the server IP addresses (or their placeholders) are resolved by analyzing app configuration files that accompany system modules. These IP addresses, together with the paths, define the fully-qualified URLs for each REST endpoint. We further analyze each module to enumerate all REST calls along with request URLs and similar metadata. We list these REST endpoints and REST calls based solely on static code analysis, where we leveraged the annotation-based REST API configuration commonly used in enterprise frameworks. We match each endpoint with each REST call across different MSA modules based on the URL and metadata in the second phase. During matching, URLs are generalized to address different naming of path variables across different MSA modules. Each resultant matching pair indicates an inter-module REST communication.

Afterward, the underlying dependency management tool's configuration file, e.g., pom.xml file for maven, is analyzed for each of the different MSA modules. This allows us to find the dependencies and libraries used by each of the apps. Lastly, the app configuration, where developers define information such as the port for the module, the databases it connects to, and other relevant environment variables for the app, is analyzed. Once the processing of each module is done, we begin the process of code-smell detection. In the following text, we provide details relevant to each particular smell and its detection.

ESB Usage is detected by tallying up all of the incoming and outgoing connections within each module. We see an ESB as a module with a high, almost outlier, number of connections, and a relatively equal number of incoming and outgoing connections. Additionally, an ESB should connect to nearly all the modules.

Too Many Standards is tricky to detect since it is entirely subjective on how many standards are "too many." Additionally, there are very good reasons developers would choose different standards for different system modules, including speed, available features, and security. We tally the standards used for each of the layers of the app (presentation, business, and data). The user can configure how many standards are too many for each of the respective sections.

Wrong Cuts depends on the business logic and, therefore, nearly impossible to automatically detect without extrapolating a deep understanding of the business domain. However, we would expect to see an unbalanced distribution of artifacts within the MSA modules along with the different layers of the app (presentation, business, and data). To detect an unbalance presentation MSA module, we look for an abnormally high number of front-end artifacts (such as HTML/XML documents for JSP). For the potentially WC business MSA modules, we look for an unbalanced number of service objects, and lastly, for WC data modules, we look for an unbalanced number of entity objects. To find unbalanced MSA modules, we look for outliers in the number of the specified artifacts within each module and report the possibility of MSA module WC. We defined an outlier count of greater than two times the standard deviation away from the average count of the artifacts in each module, which is seen in Eq. 1.

$$2 * \sqrt{\frac{\sum_{i=0}^{n}(x_i - \bar{X})^2}{n-1}} \quad (1)$$

Not Having an API Gateway is determined from code analysis alone, especially as cloud apps increasingly rely on routing frameworks such as AWS API Gateway (`https://aws.amazon.com/api-gateway/`). This uses an online configuration console and is not discoverable from code analysis, to handle routing API calls. In a study by Taibi [23], it was found that developers could adequately manage up to 50 distinct modules without needing to rely on an API gateway. For this reason, if the scanned app has more than 50 distinct modules, we include a warning message in the report to use an API gateway. This is not classified as an error, but rather a suggestion for best practice.

Shared Persistency is detected by parsing the app's configuration files and finding the persistence settings location for each of the submodules. For example, in a Spring Boot apps, the YAML file is parsed for the datasource URL. Then the persistence of each module is compared to the others to find shared datasources.

Inappropriate Service Intimacy can appear in a couple of different ways. First, we detect this as a variant of the shared persistency problem. Instead of sharing a datasource between two or more modules, a module is directly accessing another's datasource in addition to its own; however, once a duplicate datasource is found, if the module also has its own private datasource, then it is an instance of inappropriate service intimacy. Next, we look for two modules with the same entities. If one of those modules is only modifying/requesting the other's data, we defined it as inappropriate service intimacy.

Shared Libraries is found by scanning the dependency management files for each app module to locate all shared libraries. Clearly, some shared outside libraries will be shared among the MSA modules; however, the focus should be on any in-house libraries. Developers can then decide to extract into a separate module if necessary to bolster the app against the libraries' changes.

Cyclic Dependency is found using a modified depth-first search [24].

Hard-Coded Endpoints is found during the bytecode analysis phase of the app. Using the bytecode instructions, we can peek at the variable stack and see what parameters are passed into the function calls used to connect to other MSA modules. E.g., in Spring Boot, we took calls from RestTemplate. We link the passed address back to any parameters passed to the function or any class fields to find the path parameters used. We test for hardcoded port numbers and IP addresses as both should be avoided.

API Versioning is found in the app by first finding all the fully qualified paths for the app. To locate the unversioned paths, each API path is matched against a regular expression pattern `.*/v[0-9]+(.?[0-9]*).*`, matching the app convention. All unversioned APIs are reported back to the user.

Microservice Greedy is found by calculating a couple of different metrics for each module. This includes the counts of front-end files, service and entity objects. Then we find outliers, if any exist, as potential MG. We define outliers similarly as when finding MSA module WC using the Eq. 1. However, we focus only on those that are outliers due to being undersized, as opposed to too large.

6 Case Study

We developed a prototype open-source MSANose tool (`https://github.com/cloudhubs/msa-nose`) using our approach. It accepts Java-based MSA apps and performs static analysis of MSA modules. From the individual modules, it extracts the interaction patterns, combines the partial results and derives a holistic view on the distributed system. Next, it performs the smell detection and reports a list of MSA code smells with references to the offending modules and code.

Smell	Manual	MSANose
EU	No	No
TMS	No	No
WC	0	2
NAG	No	No
HCE	28	28
AV	76	76
MG	0	0
SP	0	0
ISI	1	1
SL	4	4
CD	No	No

Table 1. Code Smells in the TrainTicket benchmark [26]

Recent efforts [12] to catalog MSA testbed apps have found a lack of apps that adhere to the guidelines for testbeds outlined by Aderaldo et al. [1]. To test our app, we chose to run it on an existing MSA benchmark, the Train Ticket Benchmark [26], it is a reasonable size for an MSA app and provides a good test of all of the conditions in our app. Furthermore, it was designed as a real-world interaction model between MSA modules in an industrial environment and is one of the largest MSA benchmarks available. This benchmark consists of 41 modules and contains over 60,000 lines of code. It uses Docker or Kubernetes for deployment and relies on NGINX or Ingress for routing.

We manually analyzed the testbed for each of the eleven MSA code smells, by manual tracing of REST calls, the cataloging of entities, and endpoints within the app. We show the results of our manual assessment in column two of Table 1. Next, we ran our app on the testbed system. The app took just ten seconds to run on a system with an Intel i7-4770k and 8 Gb of RAM. This includes the average time (taken over ten runs) it took to analyze the source code fully and compiled the bytecode of the testbed app. In the third column of Table 1 is a quick overview of the results from running our app on the testbed. Our tool correctly analyzed the testbed and successfully identified the MSA code smells.

Code smells do not always break the system, but they are indicators of poor programming practice. As the testbed app has done over the past couple of years, these smells can easily work their way into the system as a system grows organically. Our tool can help developers locate code smells in enterprise MSA apps and provide a catalog of the smells and their common fix solutions.

7 Conclusions

In this paper, we have discussed the nature of code smells in software apps. Code smells, which may not break the app in the immediate time-frame, can cause long-lasting problems for maintainability and efficiency later on. Many tools have been developed which automatically detect code smells in apps, including ones designed for architecture and overall design of a system. However, none of these tools adequately address a distributed app's needs, specifically an MSA-based

app. To address these issues, we draw upon previous research into defining MSA specific code smells to build an app capable of detecting eleven unique MSA-based code smells. We then run our app on an established MSA benchmark app and compare our results to manually gathered ones. We show that it is possible, through static code analysis, to analyze an MSA-based app and derive MSA-specific code smells accurately.

For future work, we plan to assess more app testbeds. Moreover, we plan to continue our work on integrating the python platform to our approach since there are no platform-specific details, and most of the enterprise standards apply to across platforms.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research `https://research.redhat.com`.

References

1. Aderaldo, C.M., Mendonça, N.C., Pahl, C., Jamshidi, P.: Benchmark requirements for microservices architecture research. In: Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering. p. 8–13. ECASE '17, IEEE Press (2017)
2. Alikacem, E.H., Sahraoui, H.A.: A metric extraction framework based on a high-level description language. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 159–167 (2009)
3. Azadi, U., Arcelli Fontana, F., Taibi, D.: Architectural smells detected by tools: a catalogue proposal. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt). pp. 88–97 (2019)
4. Cerny, T., Donahoo, M.J., Trnka, M.: Contextual understanding of microservice architecture: Current and future directions. SIGAPP Appl. Comput. Rev. **17**(4), 29–45 (Jan 2018). https://doi.org/10.1145/3183628.3183631
5. Chatley, G., Kaur, S., Sohal, B.: Software clone detection: A review. International Journal of Control Theory and Applications **9**, 555–563 (01 2016)
6. Counsell, S., Hamza, H., Hierons, R.M.: The 'deception' of code smells: An empirical investigation. In: Proceedings of the ITI 2010, 32nd International Conference on Information Technology Interfaces. pp. 683–688 (2010)
7. DeMichiel, L., Shannon, W.: JSR 366: Java Platform, Enterprise Edition 8 Spec (2016), `https://jcp.org/en/jsr/detail?id=342`, Accessed on: March 27, 2020
8. Fontana, F.A., Zanoni, M.: On investigating code smells correlations. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 474–475 (2011)
9. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc., USA (2018)
10. Gupta, V., Kapur, P., Kumar, D.: Modelling and measuring code smells in enterprise applications using tism and two-way assessment. International Journal of System Assurance Engineering and Management **7** (04 2016). https://doi.org/10.1007/s13198-016-0460-0

11. Macia, I., Garcia, J., Daniel, P., Garcia, A., Medvidovic, N., Staa, A.: Are automatically-detected code anomalies relevant to architectural modularity? an exploratory analysis of evolving systems. AOSD'12 - Proceedings of the 11th Annual International Conference on Aspect Oriented Software Development (03 2012). https://doi.org/10.1145/2162049.2162069
12. Márquez, G., Astudillo, H.: Identifying availability tactics to support security architectural design of microservice-based systems. In: Proceedings of the 13th European Conference on Software Architecture - Volume 2. p. 123–129. ECSA '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3344948.3344996
13. Mathew, A.P., Capela, F.A.: An analysis on code smell detection tools. 17th SC@ RUG 2019-2020 p. 57
14. Moha, N., Gueheneuc, Y., Duchien, L., Le Meur, A.: Decor: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering **36**(1), 20–36 (2010)
15. Moha, N., Guéhéneuc, Y.G., Meur, A.F., Duchien, L., Tiberghien, A.: From a domain analysis to the specification and detection of code and design smells. Formal Aspects of Computing **22** (05 2010). https://doi.org/10.1007/s00165-009-0115-x
16. NGINX, Inc.: The Future of Application Development and Delivery Is Now Containers and Microservices Are Hitting the Mainstream (2015), `https://www.nginx.com/resources/library/app-dev-survey/`, Accessed on: March 27, 2020
17. Peters, R., Zaidman, A.: Evaluating the lifespan of code smells using software repository mining. In: 2012 16th European Conference on Software Maintenance and Reengineering. pp. 411–416 (2012)
18. Reeshti, Sehgal, R., Nagpal, R., Mehrotra, D.: Measuring code smells and antipatterns. In: 2019 4th International Conference on Information Systems and Computer Networks (ISCON). pp. 311–314 (2019)
19. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7), 470–495 (May 2009). https://doi.org/10.1016/j.scico.2009.02.007
20. Sae-Lim, N., Hayashi, S., Saeki, M.: How do developers select and prioritize code smells? a preliminary study. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 484–488 (2017)
21. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for Software Design Smells: Managing Technical Debt. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (2014)
22. Tahir, A., Dietrich, J., Counsell, S., Licorish, S., Yamashita, A.: A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. Information and Software Technology **125**, 106333 (2020). https://doi.org/10.1016/j.infsof.2020.106333
23. Taibi, D., Lenarduzzi, V.: On the definition of microservice bad smells. IEEE Software **35**(3), 56–62 (2018). https://doi.org/10.1109/MS.2018.2141031
24. Tarjan, R.: Depth-first search and linear graph algorithms. In: 12th Annual Symposium on Switching and Automata Theory (swat 1971). pp. 114–121 (1971)
25. Van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02). p. 97. WCRE '02, IEEE Computer Society, USA (2002)
26. Zhou, X., Peng, X., Xie, T., Sun, J., Xu, C., Ji, C., Zhao, W.: Benchmarking microservice systems for software engineering research. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE. pp. 323–324. ACM (2018). https://doi.org/10.1145/3183440.3194991