Flakiness in Microservce Applications

Shan Jiang
shanjiang@utexas.edu
The University of Texas at Austin
Austin, Texas, USA

ABSTRACT

In the present industry landscape, microservices architecture is highly prevalent. It involves the creation of small, independent services deployable and scalable autonomously, requiring diverse middleware stacks. Microservices architecture aims to overcome monolithic architecture limitations, enhancing scalability, agility, and reliability. However, it introduces challenges like flaky tests, increased memory consumption, and complex request timeouts. This project specifically focuses on microservices-induced flakiness, identifying the root causes of such problem. Recognizing the importance of regression testing, the project seeks to minimize flaky test impact. Our study addresses a research gap, identifying 5 root causes and substantiating their real existence. Each root cause category is accompanied by potential solutions, offering actionable insights to developers for reducing flaky tests during application development and testing suite creation.

CCS CONCEPTS

• **Software and its engineering** → *Software testing and debugging*.

KEYWORDS

Software Testing, Mutation Testing, Test Flakiness

ACM Reference Format:
Shan Jiang. 2023. Flakiness in Microservce Applications. In *Proceedings of Software Testing in the Era of Nondeterminism (ECE 382V'23Fa)*. UT Austin, Austin, TX, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software testing is the dominant method for validate the software quality. Software testing is important to reduce the cost of software failure, but testing itself is very expensive. Currently regression testing has became a crucial part of software development, software developers use regression test to check if software changes inject some faults in the software, break previous functionality. They integrate many tests into a regression test suite, the outcome of this test suite is important. If all the tests in the test suite pass, developers will not look at these tests further. If any test fails, developers will look into them and find the cause of this failure, to understand which changes break the functionality. However, is this assumption really convincing? The answer is NO. Unfortunately, there exists some flaky tests, which means that test results is not reliable in the same version of code, it non-deterministic pass or fail in each run, these flaky tests destroy regression test. On the one hand, test failures caused by flaky tests are hard to reproduce, developers may waste much time on these flaky failure. On the other hand, test success cannot be trusted because sometimes these tests may fail, developers have to put an eye on tests that already passed. It is more harmful that flaky tests may hide real code bugs, developers will lose confidence to software testing and even if the test cases failed, they may ignore their failures and assume they are flaky tests, so the real code bugs were missed.

Flaky tests are more likely to exist in large codebases, many researchers in the industry report that they have found many flaky tests in their codebases. For example, Google use their TAP systems to detect more than 1.6M test failures each day and 73K out of 1.6M (4.56%) tests failures were caused by flaky tests. Nowadays, software systems are becoming more and more complex, the codebases are growing larger. So more flaky tests may hide in these code bases and cause a lot of resource waste. Hence, finding the root cause of flaky tests is very valuable, developers can avoid flaky tests or easily finding what cause flaky tests. There exists some approaches to detect flaky tests, the most common one is to rerun the test suite multiple times, if one of them passed, it was declared passing. Google reruns each previously failed tests for 10 times and if it passed in any of these 10 times, it is marked flaky. iDFlakies is a framework to detect order-dependent flaky test TODO: Description of iDFlakie. If we can provide the root cause of flaky tests to the developers of such tools, they can improve their tools based on them. So finding the root cause of flaky tests is very important and valuable.

In today's industry world, microservice architecture is very popular. Microservice architecture provide series of small services which may be deployed and scaled independently, and also need different middleware stacks for the implementation. Microservice architecture is designed to reduce regression tests and to overcome the shortcoming of monolithic architecture. It separate the data and application logic from single unit to multiple microservices, improving the scalability, agility and reliability of software systems. However, microservice architecture also introduce some new shortcomings to software, including flaky tests, memory consumption, and more complex request timeouts. In this project, we focus on the flakiness caused by microservice architecture, finding root cause of these flaky tests in this area.

2 BACKGROUND

Under the current popular software development process of CI/CD, the advent of microservice architecture has revolutionized how applications are designed, deployed, and scaled. Microservices, as

ECE 382V'23Fa, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

opposed to traditional monolithic structures, decompose applications into smaller, independently deployable micro services. This architectural paradigm offers a range of benefits, including enhanced scalability, agility, and reliability.

One of the persistent challenges accompanying the adoption of microservices is the emergence of flaky tests within the regression testing. Flaky tests, characterized by non-deterministic pass or fail outcomes, have become a notable concern, particularly in the context of microservice applications. The inherent complexities of microservices, with their distributed nature and varied middleware stacks, introduce new dimensions of flaky anti-patterns and uncertainty in regression testing. Testing of individual services has been studied by many researchers. However, no research has identified the potential risks involved in microservice applications *as a whole*. More specifically, consider there are 2 service shown in Figure 1, Service A and Service B. Service A is a message producer, it put message into the RabbitMQ. Service B is a message comsumer, it consume the message from the RabbitMQ. Currently researchers focus on each service themselves and RabbitMQ itself. They run all test cases in Service A, and if they get the right results, we think the test pass. However, when we see the whloe system as the test object, it is possible that each part of the system (each services) work well, but the whole system didn't work as we imagined.

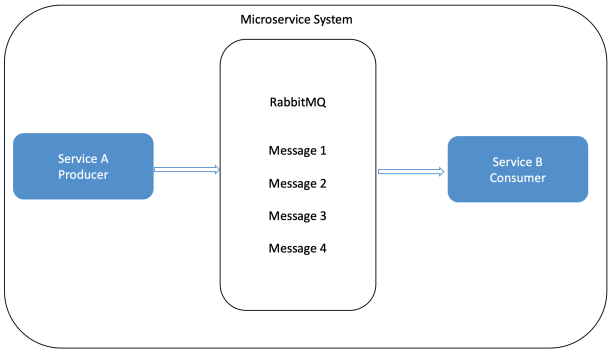

Figure 1: Microservice Architecture System

Microservices empower development teams to work on discrete, specialized components, allowing for rapid deployment and scaling of individual services. However, the autonomy of these services can lead to intricacies in testing, where dependencies and interactions among microservices may result in flaky test behavior. The decentralized nature of microservices, while advantageous for system scalability, also brings about challenges related to test consistency and reliability.

Unlike monolithic architectures, where a unified codebase undergoes comprehensive regression testing, microservices necessitate a more intricate testing strategy. Coordinating and validating interactions between microservices becomes critical, and the potential for flaky tests to obscure genuine issues becomes more pronounced. As the software industry increasingly embraces microservices to meet evolving demands, addressing the unique challenges associated with flaky tests in this context becomes imperative. The high complexity of microservice-based applications, compounded by the diversity of middleware stacks employed across services, amplifies the likelihood of encountering flaky tests. These tests not only impede the efficiency of regression testing but also pose a risk of masking actual code defects.

Recognizing the significance of flaky tests in microservices, this project focuses on identifying and understanding the root causes of flakiness specific to this architectural paradigm. By delving into the intricacies of microservice interactions, dependencies, and deployment models, we aim to provide insights that enable developers to mitigate flaky tests effectively. Addressing the challenges posed by flaky tests in microservices contributes to the overall reliability and robustness of software testing methodologies in the era of distributed, microservice-oriented architectures.

3 CAUSES OF FLAKINESS IN MICROSERVICE ARCHITECTURE

First we analyze the root cause of test flakiness in microservice architecture and classify them into 10 categories. Then we try our best to find if this category of root cause can be found in each project of our evaluation dataset. Here I will introduce each category of root causes one by one.

3.1 Unstable Environments

For microservice architecture, a simple request may need a lot of service work together. The test environments may be unstable, this will cause flaky tests. Variances in operating systems, libraries, environment variables, number of CPUs, or network speed can produce flaky tests. While it's impossible to have 100% identical systems, tightly controlling library versions and being consistent during builds can help avoid instability. Even minor version changes in a library can introduce unexpected behavior or even bugs. Keeping the environment as equal as possible throughout the CI process reduces the chance of creating unstable tests.

In addition, differences between local development machine, test environment and CI fall into this category, too. Although we don't have resources to test our results in real production environment, we do believe that in different machine, this problem will be more frequently happen. Containers like Docker are great for controlling what goes into the application environment and isolating the code from OS-level influence, we strongly suggest developers to use containers to avoid such flaky tests.

Figure 2 is an example of this cause, several tests run in parallel change the shared resource and cause flaky tests.

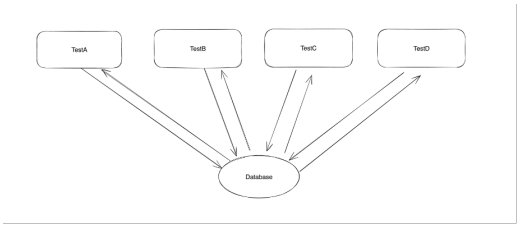

Figure 2: Unstable Environments

3.2 Asynchronous Communication

Flaky tests can happen when the test suite and the application run in separate processes. When a test performs an action, the application needs some time to complete the request. After that, it will check if the action yielded the expected result.

Consider the following case: In a microservices architecture, services communicate with each other over a network, and their interactions are vital for the proper functioning of the application. Consider a scenario where multiple microservices are deployed across different servers or containers within a cloud environment. And the network is unstable and the latency changed a lot. Due to the dynamic nature of cloud environments, network conditions may vary, resulting in intermittent latency or, in extreme cases, network outages. This variability in network performance can directly impact the communication between microservices, leading to unexpected delays or failures in data transmission.

This problem may be more harmful when there existing some cascading RPCs. A microservice application workflow begins with a user request, which triggers a series of cascading requests. As the client processes the initial request, it delegates certain tasks to its downstream services, which, in turn, may invoke additional services to handle various aspects of the request. Rather than establishing direct communication between these services, microservices architecture employs a sidecar pattern. This involves running an additional container alongside the service container to facilitate inter-service communication. While the sidecar design effectively ensures the scalability and isolation of different services, it faces a notable limitation in efficiently managing timeouts. When a service initiates a timeout, downstream services further along the chain may not be promptly notified and may continue processing requests that have already been abandoned. This inefficient handling of timed-out requests can accumulate, leading to reduced throughput and impaired overall performance, especially in scenarios where the network load is heavy.

This may cause flaky problems because only the first service know that the user's request is time-out, the following services will continue to deal with this request and change the state of system. This is because microservices are designed to interact seamlessly under normal network conditions, and the test case passes consistently during typical testing scenarios. However, in an unstable environment with intermittent network latency, the timing of service interactions becomes unpredictable. The test may pass in some runs when the network is stable but fail in others due to delays or timeouts caused by network fluctuations. As shown in Figure 3, network latency is unstable when the load increase. And the change pattern of each kind of request is not the same. Remember that this is just 3 different request in the same microservice application, considering different applications using different frameworks, it is hard to imagine how much network latency will fluctuate.

Developers can use network virtualization tools to simulate different network conditions during testing, allowing for a more controlled environment. In addition, service simulation can also be used to isolate microservices during testing and reduce the dependence of certain scenarios on the actual network.

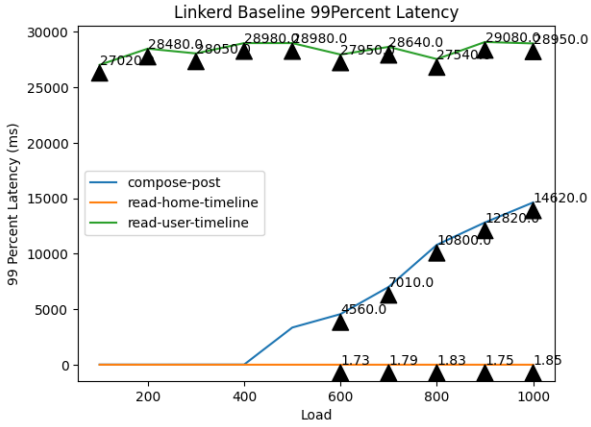

Figure 3: Latency under different workloads.

3.3 Deferred Operation

There exists a rare cause of flakiness that is the use of schedulers or deferred operations. We can use buy a product from an E-Commerce system as an example, this is also a real test case in Section 4 Evaluation. When we submitted the order, we declared a scheduler and he gave us fifteen minutes to pay for our order. We wanted to pay the US dollar order through RMB, but the bank's foreign exchange system operates between 22:00-23:00 every day for maintenance, currency exchange is not possible: Then, during the test run, it unexpectedly worked because the test happened to be running at that time. The developer who wrote the test did not anticipate such a side effect. To avoid this behavior, developers can configure a schedule. This will make the application easier to maintain (no need to rebuild the code to change the schedule). It also fine for developers to override the schedule in the test so that the scheduler only runs manually.

Besides, we also found that when the workload is very large, e.g. requesting 100,000 new order in 1 second, the number of orders present in the system is not equal to the number of requests, which means non-deterministic behavior occurs.

3.4 Cache

The caching mechanism is one of the major reason for flaky tests in microservice applications. In today's industry, almost all microservice applications use caching, which indeed reduces the need to access the underlying slower storage layer and thus improves data retrieval performance. But at the same time it also brings uncertainty. If the refresh rate of cache is set too large, we may get outdated data after an update operation. And when cache breakdown, some data is returned by cache and others returned by database, it is non-determinism.

Usually, in order to ensure the consistency of the data in the cache and the data in the database, we set an expiration time for the data in Redis. When the cached data expires, if the data accessed by the user is not in the cache, the business system needs to regenerate the cache, so The database will be accessed and the data will be updated to Redis, so that subsequent requests can directly hit the

cache. Then, when a large amount of cached data expires (invalid) at the same time or Redis crashes, if there are a large number of user requests at this time, they cannot be processed in Redis, so all requests directly access the database, causing a sudden pressure on the database. If it increases, it will seriously cause database downtime, thus forming a series of chain reactions and causing the entire system to collapse. In addition, there are usually several data in the business that are frequently accessed. Such frequently accessed data is called hot data. If a certain hotspot data in the cache expires and a large number of requests access the hotspot data, it cannot be read from the cache and directly access the database. The database can easily be overwhelmed by high concurrent requests. This can also cause the system to crash, producing non-deterministic results.

3.5 Context Switching

This category of flaky tests is not a general problem. By contrast, it only occurs in microservice applications developed by Spring Framework.

These flaky tests can occur when you use Spring framework and using multiple test classes, but this kind of flaky tests is very hard to reproduce and I am not sure about the context switching logic of Spring test. Whenever we add a new test class to our project, this problem may happen.

Since this problem is not a common problem of microservice architecture but is caused by the implementation of Spring Framework itself, this article will not discuss it too much.

4 EVALUATION

To evaluate the correctness of root causes mentioned above, I choose 4 different microservice applications developed by C++, Golang, Java(with Spring Boot and Spring Cloud Alibaba), and we want to find these causes in microservice applications.

4.1 Environmental Setup

In order to conduct the experiments, first we need to run the microservice applications, then we can simulate some real world operations and find if there exist some flaky tests. Different microservice applications may have different features. We select Death Star benchmark [2] as our evaluation benchmark, specifically we run e projects (Hotel Reservation, Media Service, and Social Network) in this benchmark and gather data from each of them. DeathStar Benchmark is a novel, open-source benchmark suite built with microservices that is representative of large end-to-end services, modular and extensible. We injected each microsevice with linkerd sidecar, the reason we used linkerd is because it is simple and it comes with a mechanism to adjust timeout so it fits with our project objective. Finally, we used wrk2 to conduct stress test to the microservice application, and we used the wrk2 report for data analysis. We run all experiments on CloudLab with Ubuntu 20.04, using Kubernetes and helm to set up each projects.

4.2 Results

Table 1 shows the general results of this project. We can see that unstable environments and cache problems exist in all of the 4 microservice applications, because dependencies and cache mechanisms exist in almost all microservice applications, and these four projects are no exception. Deferred operation flaky tests appears in hotel reservation and E-Commerce system project, this is because they have payment-related service, this category of flakiness always lies in such services. Asynchronous communication also appears in every projects, this is a general problem related to network latency, database performance, etc. But Context switching only exist in E-Commerce system project because it is the only Spring project in my benchmark, and this problem is a Spring Framework problem.

4.2.1 Media Service. Media service application is a platform which can be used to view movies, and the platform will recommend movies for users. We found unstable environments, asynchronous communication, and cache flaky tests in this application. For unstable environments, it is easy to reproduce. First you can add some data to the database and change them when some tests have finished and others are still running, or you can make some test change the databases before tests using these data, this will cause unstable environment flaky tests. For asynchronous communication, you can use wrk2 to generate high workloads and check if the user get right response, in my results I found that when we set a timeout and retry in the sidecar, we will get wrong movie for 21 out of 100000 request. For cache, we can set a limit refresh time of cache update and update the database frequently (less than limit refresh time), in this way we can see the flakiness.

4.2.2 Hotel Reservation. Hotel reservation application is a platform where users can book hotels for their trips. We found unstable environments, asynchronous communication, deferred operation and cache flaky tests in this application. For unstable environments, you can delete some hotels from the database in some tests after the tests which have successfully summit the order to this hotel but didn't make payment, in this test case, sometimes it will pass, sometimes it will fail. For asynchronous communication, you can also use wrk2 to generate high workloads and check if the user get right response. For cache, we can set a limit refresh time of cache update and update the database frequently (less than limit refresh time), in this way we can see the flakiness. For deferred operation, it is related to the payment service, we can set a maintain time period of bank system and then run tests in that time and not in that time period, and we will see the flakiness.

4.2.3 E-Commerce System. E-Commerce system is shopping platform facilitating essential functions like shopping cart, order/inventory tracking, payment processing, and high-demand flash sales. We found unstable environments, asynchronous communication, deferred operation, context switching and cache flaky tests in this application. The methods for viewing the other four flaky tests are basically the same as before and will not be described again here. The only difference is Context switching, which is a problem caused by the Spring framework. We only need to break the test code into multiple java test classes and run these tests multiple times to observe the flaky test. This problem is caused by the Spring implementation. I am not sure when and why this problem occurs. It seems to occur every time a new test class is added. The potential reason may be related to the context management mechanism of the spring framework. , interested researchers can view his source code to determine the cause of the problem.

Category	Media Service	Hotel Reservation	E-Commerce System	Social Network
Unstable Environments	✓	✓	✓	✓
Asynchronous Communication	✓	✓	✓	✓
Deferred Operation		✓	✓	
Context Switching			✓	
Cache	✓	✓	✓	✓

Table 1: General results

4.2.4 Social Network. Social network is a message platform where users can add friends and chat. We found unstable environments, asynchronous communication, and cache flaky tests in this application. This application does not have a unique flaky root cause. The way to observe the existing root cause is similar to several previous projects, so I will not go into details here.

5 THREAT TO VALIDATE

Here we discuss threats to validity of our study following the classification by Wohlin et al.[7]

5.1 Conclusion Validity

The main conclusion of this project are derived from 4 different microservice applications. I have tried my best to expand the richness of the data set, using three different programming languages: Java, C++ and Golang, and I also selected different frameworks among various middleware choices including message queue, cache, database, etc. However, these four applications are not completely representative of the numerous industrial microservice applications. So the conclusion validate may be different with real industry. We deal with this problem by selecting most popular frameworks (e.g. Spring Framework), database (e.g. MySQL, PostgreSQL), and middlewares (e.g. Redis, RabbitMQ, Kafka). We do not have access to the infrastructure of companies, but the conclusion we get is very similar to theirs since their infrastructure and open-source popular frameworks are used to deal with similar problems.

5.2 Internal Validity

Flaky tests are non-deterministic in nature, so this create potential threats to the validity of our conclusions. Some of our conclusion only occur when the workload is high, but it is not always appear when in high load. We believe that there exists some flakiness anti-pattern we did not identified and there are some flaky tests in high workload but we did not find.

Due to the fact that flaky tests non-deterministically pass or fail, the flakiness may be hard to reproduce, especially for test failures. Fortunately, we revealed the root cause of flaky tests and these causes are easy to understand. This reduces internal validity to a certain extent.

5.3 Construct and External Validity

As we discussed before, the results from our study on 4 microservice application may be not complete. We know that current large companies like Google have very large codebases (billions line of code) in different language. We think there must be some root causes of flaky tests that appear only in extremely large code bases, or lies in coorperation between services developed in different coding language. Overall, we believe that our conclusions are still applicable to larger industrial code bases, but may not be complete because very large code bases may have unique problems and we obviously do not have sufficient permissions and resources to conduct research on these issues.

6 RELATED WORK

Cost of flaky tests. Luo et al.[3] proposed the first extensive study of flaky tests which some of flaky tests were very hard to reproduce. After that, much researchers put effort on that problem and find that flaky tests are very common in real world software systems. For example, Palomba and Zaidman[4] found 8829 flaky tests out of 19532 JUnite tests from 18 projects, which means that 45% of tests are flaky. More harmfully, some these flakiness happens frequently that only run 10 times of tests can see both pass and fail behavior, but some flaky tests in the database is hard to reproduce that even the researcher themselves cannot reproduce the test pass or fail. This greatly hurts software developers' trust in software testing technology. Not only hurts the trust, flaky tests also cause the software system crash. Rahman and Rigby[5] showed that ignoring the flaky tests will cause Firefox to crash in production. Flaky tests exist in various scenarios. For example, dong and others found that concurrency flaky problems are often hidden in Android projects. Hence we can believe that microservice application also have some flaky problems as we discussed before.

Root cause of flaky tests. Zhang et al.[8] pointed out that test suites can suffer from test order dependency, which means the test results can be affected by the run order of test cases. Wing Lam et al. proposed iDFlakies to detect such order-dependent flaky tests. Such kind of flaky tests is common but not the focus of this project. The subject close to this project is Non Order-Dependent flaky tests. These root causes can be asynchronous wait, concurrency, race condition, deadlocks, and test dependencies. Throve et al. also found these root causes for Android projects. But keep in mind that such causes are still *in* the service, this project discussed problems in higher lever, the microservice application level which can be a collection of many projects. There exist some flaky tests in each project (which have been well researched by previous work), but when these projects works together, there also exist some new flakiness. This is what this project do.

Microservice architecture. The microservices architecture has emerged as a prevailing choice in the realm of service-oriented software. It represents a distinctive approach to system design, emphasizing the segmentation of the overall system into small, lightweight services, each purposefully crafted to execute a specific and

cohesive business function. This architectural paradigm is an evolutionary step beyond traditional service-oriented architecture[6], and it is succinctly defined in [1] as "the minimal independent process that interacts via messaging." In essence, microservices architecture embodies a distributed application where each module operates as an independent microservice. The core advantages widely acknowledged in adopting this architectural style encompass a spectrum of benefits. These include heightened agility, increased developer productivity, enhanced system resilience, scalability, reliability, maintainability, separation of concerns, and streamlined deployment processes. However, it is important to note that these advantages are accompanied by a set of challenges inherent to the microservices approach. These challenges encompass aspects such as service discovery across the network, effective security management, optimization of communication channels, handling data sharing, and addressing performance considerations. This article focuses on the flaky tests problem that may be caused by microservice architecture and conducts an in-depth discussion. This is also a pain point of microservice architecture because when the whole system behavior goes wrong, it is difficult to find out which service or middleware goes to the wrong way. Flaky tests make system performance elusive and may lead to a huge waste of resources.

7 CONCLUSION

Regression testing is important but can be greatly undermined by flaky tests. In this project, we conducted an in-depth study of microservice applications, filling the gap in flake testing-related research in this area. We have found 5 root causes of flaky tests in microservice application and found real exsitence for each of them. For each category of root cause, we provide potential solutions to help developers minimize flaky tests when writing applications and testing suites. I believe this project is not only useful for microservice application developers, but also for developers of flaky tests detection tools. It can also help them to improve their detection tools for problems in microservice scenarios.

8 ACKNOWLEDGE

This project focuses more on theoretical analysis of microservices architecture, the main coding part is in setting up all four microservice applications in my benchmark, so I provide the setup script of my benchmark and the repository address of my benchmark. The quantifiable experimental results are mainly my timeout and network delay measurements for the asynchronous communication root cause. I also provide these results in the code repository.

After setup the applications, you can also reproduce the corresponding flaky problem as described in section 4.2. Because these flaky tests were manually run and observed by myself, I did not have an automated script to implement automatic observations. But fortunately, these root causes are not difficult to understand. Compared with previous work, this project focuses on higher-level testing, treating the microservice system as a whole to observe its *'behavior'*.

This is the code base address of scirpt and results: https://github.com/shanjiang98/Flaky-microservice

REFERENCES

[1] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, 195–216.

[2] Yu Gan et al. 2019. An open-source benchmark suite for cloud and iot microservices. *arXiv preprint arXiv:1905.11055*.

[3] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 643–653.

[4] Fabio Palomba and Andy Zaidman. 2017. Notice of retraction: does refactoring of test smells induce fixing flaky tests? In *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[5] Md Tajmilur Rahman and Peter C Rigby. 2018. The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 857–862.

[6] Johannes Thönes. 2015. Microservices. *IEEE software*, 32, 1, 116–116.

[7] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[8] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D Ernst, and David Notkin. 2014. Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 385–396.