INDIAN JOURNAL OF SCIENCE AND TECHNOLOGY

RESEARCH ARTICLE

Performance of Dynamic Retry Over Static Towards Resilience Nature of Microservice

E Punithavathy[1]*, **N Priya**[2]

1 Assistant Professor, Department of Computer Applications, Madras Christian College, Chennai, Tamil Nadu, India
2 Associate Professor, PG Department of Computer Science, Shrimathi Devkunvar Nanalal Bhatt Vaishnav College for Women, Chennai, Tamil Nadu, India

OPEN ACCESS

Received: 01-04-2024
Accepted: 20-05-2024
Published: 30-05-2024

Citation: Punithavathy E, Priya N (2024) Performance of Dynamic Retry Over Static Towards Resilience Nature of Microservice. Indian Journal of Science and Technology 17(22): 2316-2323. https://doi.org/10.17485/IJST/v17i22.1041

* **Corresponding author**.

punithavathy@mcc.edu.in

Funding: None

Competing Interests: None

Copyright: © 2024 Punithavathy & Priya. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Published By Indian Society for Education and Environment (iSee)

ISSN
Print: 0974-6846
Electronic: 0974-5645

Abstract

Objectives: The main objective of this study is to enhance the flexibility of microservice-based cloud applications, so they can cope with transient or short-term failure conditions more effectively, providing a higher throughput of cloud applications as a result. **Methods:** The comparison is done between the existing or manual resilient patterns such as circuit breaker and retry with a dynamically proposed retry patterns using a microservice application. **Findings:** The short-term or transient failures in cloud based microservice applications can occur for many reasons. Examples such as Network glitches, service failures, timeouts of requests etc. These failures can bring down the entire application resulting in a cascading of failures. Resiliency patterns are used to protect these applications from failures. The widely used patterns are circuit breaker and retry, with static configurations. It is possible that the static configurations used in resiliency patterns might not support all types of failures. Hence, a dynamic approach has been proposed to satisfy all these transient failures of cloud. The analysis proves that the application efficiency is balanced in dynamic retry than the static configurations of resiliency pattern. Hence, performance can be increased up to 34.3%, which means the availability can be ensured, in transient failure cases. Very little research has been carried out on this resiliency pattern, and there are no standards available for this pattern. When comparing with the existing adaptive retry pattern the performance is increased up to 88.52%, while using a microservice application. **Novelty:** A dynamically modified resilient pattern is been proposed by incorporating additional parameters such as execution time to the existing parameters of the static retry pattern. This is an individual resilience pattern, which can perform independently when compared with other dynamic resilience patterns.

Keywords: Dynamic retry; Request time; Static circuit breaker; Resilience; Static retry

1 Introduction

Errors in networks arise for many reasons, they are network glitches, hardware issues, and busy traffic. Furthermore, the unavailability of services or resources causes an application to fail. To survive these failures, cloud applications must be resilient and bounce back from those failures.[1,2] In microservice, resilience is achieved using patterns like a circuit breaker, retry, bulkhead, rate limiter, etc. These patterns are available in libraries, which can be utilized by specifying the manual configuration parameters. From the available patterns, it is found that the circuit breaker pattern and the retry pattern was the widely implemented pattern due to its simplicity and efficient way of handling of failures.[3,4] These patterns are implemented either through method-based implementation or proxy-based implementation. These patterns are widely implemented for transient failure cases. And transient failures can be of many types, occasionally that might not suit the parametric values of these patterns. The demand for these patterns is very high, but the problem of using these patterns is due to their manual configuration.[5]

For the successful survival of these applications, the values or retrial requests have to be generated efficiently. The developers, who develop such applications might be less aware of transient cases and may fail by storing misconfigured values. These kinds of mistakes will bring the application down very often.[6] Hence the process of manual configuration has to be replaced by auto-configuration procedures. The main motivation is to ensure a reliable and consistent deployment process of microservice applications, which can reduce manual errors and provide a quicker recovery from cloud failures. The optimization of requests is a crucial component of microservices applications. When it is not handled properly, the entire application that is made up of services will fail cascading.

This research work is carried out to develop an automated retry pattern, which will execute based on the conditions of the system. It is important to increase the availability of cloud applications hosted by distributed systems.[7] Various types of cloud failures exist, and they cannot be prevented under real time scenarios. Since static configurations can handle only one type of failure, they are unsuitable for other cases. In terms of failures and execution time, the automated resilience pattern plays an important role in selecting the appropriate values.

Research is carried out in these areas, for overcoming this problem of static configurations as follows: In the previous work, an adaptive retry and adaptive circuit breaker was implemented, which is an automated resilience pattern. This pattern incorporates the automation of circuit breaker pattern, by which based on the failure counter and the state of the circuit breaker, the dynamic retry pattern was implemented.[8] The problem with this kind of dynamically formed retry pattern, is it increases the complexity of the application since the number of retries and waiting time is determined based on the parameters of circuit breaker pattern. In another analysis work, the retry pattern with initial backoff delay performs in a well-balanced way, compared with exponential delay and maximum retries.[9]

In the proposed methods, the dynamic retry pattern is calculated easily by targeting the execution time and failure counter. The existing pattern involves either the static configurations or based on the behaviour of other patterns, these patterns are dynamically calculated. Moreover, the proposed method is initially built inside the coding part, and it works successfully for small to large applications.

2 Methodology

A cloud application was implemented using a microservice architecture named an Election Management System (EMS) using a combination of two services. Each service was created in Java using Intellij Idea and libraries were added using spring boot components. Additional dependency files are added using Maven. Whenever the user sends a request for a voting service that runs in port 8081, it redirects the requests to the verify service, which runs in port 8080 for verification. The voting process is carried on only if the verification process is successful. After the successful completion of voting, the successful response is returned to the client that generates the request. In case of failure of requests, an error message will be displayed on the client side. To generate requests, an Application Programming Interface was created using the Postman tool. It is used to record all the get and post requests to the service.

The Figure 1 describes the process of microservice application. To make an application resilient, failures are injected into it, and its behaviour pattern is analysed. Injecting a failure can be accomplished by shutting down the service, increasing CPU load pressure, or increasing network latency.[1,10] By keying the manual configurations, the static circuit breaker pattern and the retry pattern were added to this application. Failures are incorporated using the principles of request overload and noisy neighbours.[11] The term request overload is bombarding the service with many requests at a given time. Applications with many services, often suffer from this kind of request in cloud traffic, which might be happening for some specific services. The term noisy neighbours refer to the concept of problems with their next services. In the existing pattern of circuit breaker and retry patterns, during failures, the fallback pattern was incorporated into the code. This pattern will prevent the errors from being displayed on the client's screen.

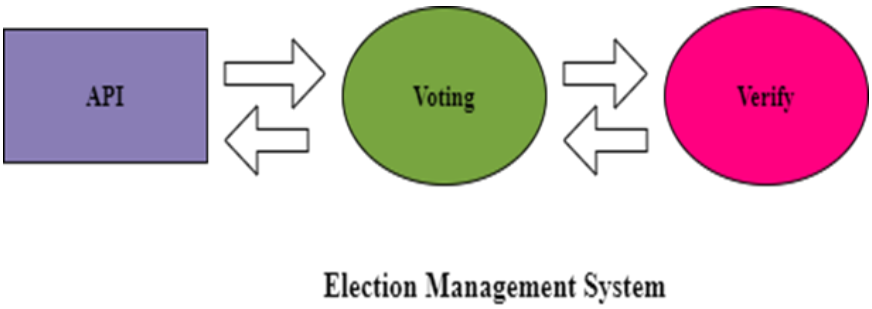

Fig 1. Sample Microservice Application

2.1 Resilience Pattern

As a renowned feature of cloud applications, the microservice architecture implements these patterns from available libraries such as Hystrix, Resilience 4j, Polly, Selenium, etc. The widely used patterns of resilience are circuit breaker and retry patterns, which have been implemented using the static configurations of the parameters. To fit into the distributed systems, applications must be equipped with dynamically modified parameters to survive in these distributed systems.[12]

2.1.1 Circuit breaker

It is similar to the one used in electrical appliances. To free the service from overloaded requests, this pattern is used. The retry pattern is often used for short-term failures, whereas this pattern is used for handling persistent failures.[13] Out of all the patterns, it is widely used due to its efficiency in surviving failure states. It is composed of three states, they are open, half-open, and closed states. The closed state will forward the request as usual. When the service encounters repeated failures, the state is shifted to open, which will stop the incoming service requests. After a stipulated duration, the state is again shifted to half-open where only certain calls are permitted to service until they get a successful response. The behaviour of the circuit breaker depends upon the parameter values specified in the parameters. The property specifications of circuit breakers are:

Table 1. Configuration parameters of Circuit breaker

Parameter	Descriptions
failurerate_threshold	Maximum failure limit, it can hold
Permitted_number_of_calls_in_half-open_state	Permitted requests in the half-open state
maxWaitdurationinhalfopenstate	Maximum waiting time in open state
Slidingwindowtype	Count-based or time-based to track the success and failure of requests
Minimumnumberofcalls	Minimum calls that are required to track
Waitduration in open-state	Waiting time in open state
Automatictransitionfromopentohalfopenenabled	Whether it can auto-shift its state from open to half-open
Slowcallratethreshold	Maximum timeout requests limit
Slowcallduration threshold	Maximum timeout limit

The Table 1 describes the parameter required for working of circuit breaker, all these parameters will be assigned an initial value, even before facing the transient failures.

The Figure 2 describes the working process of the circuit breaker diagram, where the closed state represented in green colour, will pass all the requests to the specified service, without any interruption. The open state shown in red colour will stop the incoming requests for the specified time duration. Finally, the half-open state, shown in yellow colour will allow only a few requests to the specified service. These colours represent the traffic signals colour, as they perform in the same way.

2.1.2 Retry

The simplest and easy-to-use pattern is the Retry pattern. It is one of the oldest and most widely used patterns across networks. During the request failure, this pattern is used to retry the number of calls for a specified time, with a specified time interval. They were just used to retry the requests, to the service. There are different types of retries based on the difference in waiting

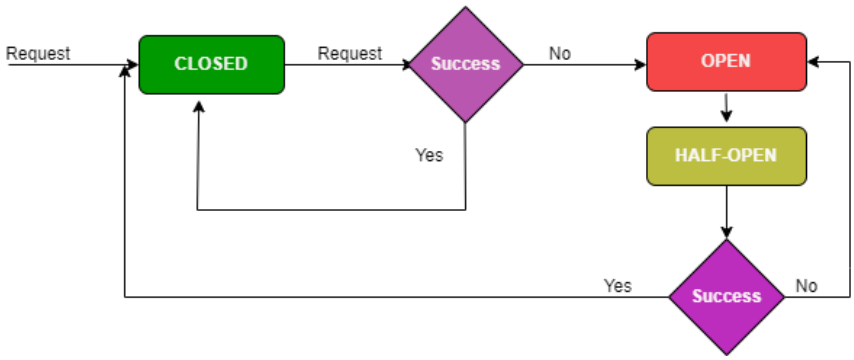

Fig 2. Process flow of Circuit breaker Pattern

time. They are simple retry, retry with an exponential pattern, and retry with the randomized interval.[14] These patterns are dependent on the configuration of parameter properties.

Table 2. Configuration parameters of retry pattern

Parameters	Descriptions
Max attempts	Maximum number of retries
Waitduration	Base wait time
Interval function	The time interval between each retries
Failaftermaxattempts	Failure counter

The following Table 2 discusses the retry pattern configurations where, it discusses about the number of attempts that can be made from system side. It also specifies the waiting time, to be included in each retry.

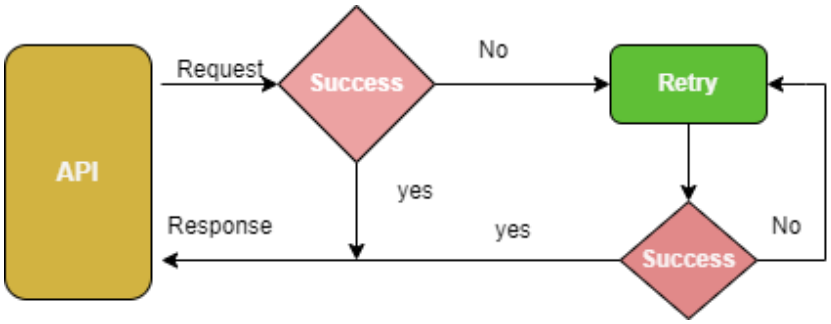

Fig 3. Process Flow of Retry pattern

The Figure 3 explains the process flow of retry pattern. When retry is initialized, a system made request is initiated by the system to the service end. On successful response to the request, the retry pattern will reset its value and move back to its usual phase.

2.1.3 Proposed dynamically modified retry:

The problem with the static circuit breaker and static retry pattern is that the wait time in the states may vary based on their failure nature. Since the parameters are already specified even before the failure, the better performance of the application might not happen. Failures can happen sometimes due to dependencies or other service-based factors. The idea of having a manual configuration does not match the above failures, resulting in a prolonged open state. The minimum or maximum amount of waiting time might increase or decrease the service idle time but affect the service execution time to a great amount.[7,15]

The Figure 4 explains the sequential working model of the retry pattern. The most important parameters are the basic wait time and the number of retries to be performed. The retry comes into action, only in failure cases as shown. In case of an

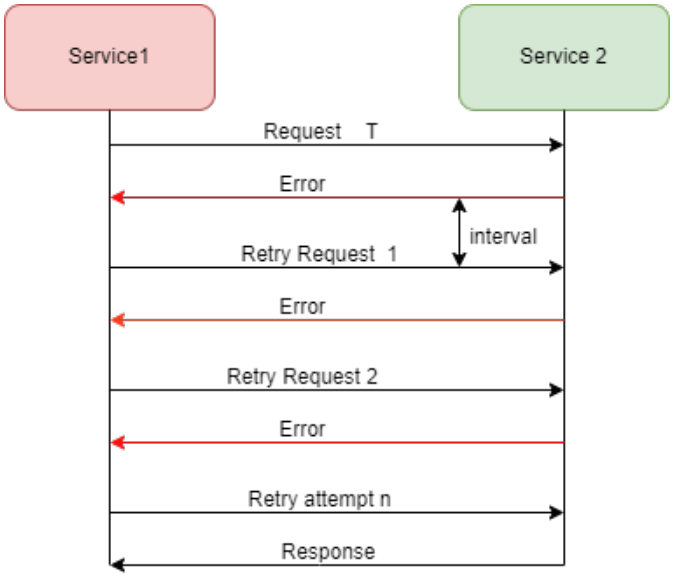

Fig 4. Sequential diagram Retry Pattern

exponential pattern, the waiting time will be increased regularly. The random pattern will set the minimum and maximum values out of which a random value will be predicted as a waiting time. The simple retry follows the duration specified in properties.

Algorithm for improvised retry pattern

Step1: : Set f_count=0 (number of failures) and waiting time=time_limit

Step2: set retry=initial_value and retrylimit=max(maximum retry)

Step3: if status==success then return response**.**

Else if status==error then f_count++

Step4:if Exectime>avg_time or f_count>0 then retry=retry/2

Step 5: else retry=retry+1

Step 6: if retry>retrylimit then reset

Step7: set waiting time=max(time_limit, Exec_time)

In the modified algorithm, the main parameters of the retry pattern have been fixed dynamically using the execution time taken for each request, as an input. The actual waiting time and the number of retry attempts were determined from the request time. In case of a successful response, the retry pattern does not function and has its value reset. When there is a failure, it starts with initial values, and in continuation with the failures, the pattern determines the value of waiting time-based on the request time taken for each request. When the request time is greater than the specified limit, or the failure counter is greater the retry attempt is half of the total request time. Similarly, when the request time is lesser than the average time, the attempt would be a sum of three-fourths value of the request time. In any case, if the attempt reaches the maximum limit, all the values will be reset.

In the process of dynamic retry. When there is no problem with the service the retry will not perform, whereas when they encounter any problems, the dynamic retry pattern will come into action. In dynamic retry, the waiting time and the number of retries are calculated based on the time taken for each request.

3 Results and Discussion

This section discusses the details of execution times that are recorded using API.

Here the failures are created using the concept of request overload and noisy neighbours. The performance of these patterns at these failure cases were observed and, these responses were recorded for each request and an average value of five requests are shown in the table. A total of 100 requests were recorded.

The average execution time is taken, using circuit breaker=50.9ms

The average execution time is taken using a retry pattern=81.3ms

Table 3. Execution time while using circuit breaker, dynamic retry, and static retry

Requests	Circuit breaker(ms)	Dynamic retry(ms)	Retry(ms)
5	112	20	56
10	45	24	51
15	39	19	68
20	41	30	87
25	30	31	100
30	38	25	78
35	65	60	90
40	43	32	120
45	53	29	89
50	41	25	167
55	47	39	80
60	80	45	80
65	56	34	77
70	46	32	78
75	42	30	60
80	56	65	89
85	50	32	49
90	54	26	59
95	42	50	89
100	38	20	59

The average execution time is taken using dynamic retry= 33.4ms

This section discusses the throughput of the application, when compared with the throughput of application used by adaptive retry pattern.[8]

The performance of the system can be increased by replacing the static retry and circuit breaker with dynamic retry by 58.3% and 34.3% respectively.

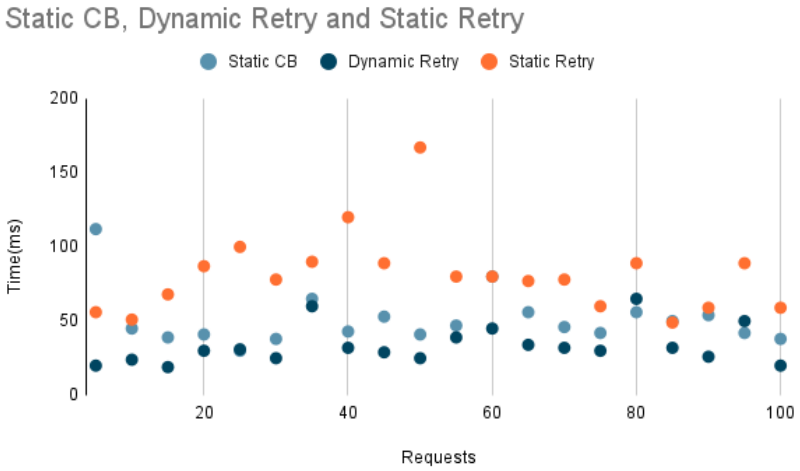

Fig 5. Execution time of circuit breaker, retry and dynamic retry patterns

From Figure 5, it is known that the analysis of different request times taken while implementing these resilience patterns. The graph in Figure 4 proves that the request time for static retry and the static circuit breaker is quite long when compared with the dynamic retry pattern. In the static configuration of the retry pattern, an increase in the number of retries harms the

survival of cloud applications.[13,15] It was, furthermore, increasing the waiting time feature of requests harms by predicting the failure rate of the system. These will cause downtime in the performance of the system. On average, the system's performance can be increased to 34.3%, while using a dynamic retry pattern.

Very few research has been performed in the area of resiliency pattern of microservices. One such work is called adaptive retry pattern, which works with the support of circuit breaker pattern. The carried throughput of an microservice application is lesser than the proposed dynamic retry pattern. The throughput of an microservice application can be calculated as follows:

Throughput= (count of completed requests*100)/ time taken for request.

The following Table 4 displays the carried throughput of an application while implementing these dynamically modified patterns. It also states the number of parameters required to calculate the dynamic pattern.

Table 4. Comparison of throughput between dynamic retry and adaptive retry

Dynamic Patterns	Carried throughput	Parameters
Adaptive Retry [8]	86.73%	Failure counter, response time, circuit breaker open state, status
Dynamic Retry	88.52%	Failure counter, response time, max_attempts, status

The results from the table shows that the throughput obtained from dynamic retry is greater when compared with that of adaptive retry pattern. And moreover, the existing work is not an independent pattern and cannot work as an individual resiliency pattern, whereas this dynamically modified circuit breaker pattern can work as an individual pattern, without depending upon any other patterns.

The other major problems with the static resiliency pattern are that it does not guarantee a recovery, since it would be retried a specified number of times. Hence, the dynamic pattern is designed in a way that it helps the system to recover back from its failure conditions, by specifying proper waiting time and number of retry patterns till it gets a successful request.

4 Conclusion

The retry pattern, the easy and one of the oldest patterns in cloud systems, which has been dynamically modified to cope with transient failures, they face in deployment. It also proves that the throughput is greatly increased when compared with other static resilience patterns as well as the adaptive retry of the existing works. It works as an independent pattern and improves the efficiency of the system to the maximum of 88.52%. This dynamically modified retry pattern, this is used only for retrying requests and does not care about the other features such as limiting the requests. A complete framework has to be provided in such a way that from the beginning to the end, all suitable dynamically modified patterns should be incorporated which will contribute to the success of the application.

References

1) Raj P, David GSS. Engineering Resilient Microservices toward System Reliability: The Technologies and Tools. In: Cloud Reliability Engineering. CRC Press. 2021;p. 77–116. Available from: https://www.taylorfrancis.com/chapters/edit/10.1201/9781003030973-3/engineering-resilient-microservices-toward-system-reliability-technologies-tools-pethuru-raj-sobers-smiles-david.
2) Mendonca NC, Aderaldo CM, Camara J, Garlan D. Model-Based Analysis of Microservice Resiliency Patterns. In: 2020 IEEE International Conference on Software Architecture (ICSA). IEEE. 2020;p. 114–124. Available from: https://doi.org/10.1109/ICSA47634.2020.00019.
3) Cerny T, Abdelfattah AS, Maruf AA, Janes A, Taibi D. Catalog and detection techniques of microservice anti-patterns and bad smells: A tertiary study. *Journal of Systems and Software*. 2023;206:1–43. Available from: https://dx.doi.org/10.1016/j.jss.2023.111829.
4) Bogner J. On the evolvability assurance of microservices : metrics, scenarios, and patterns. 2020. Available from: http://dx.doi.org/10.18419/opus-10950.
5) Carvalho L, Colanzi TE, Assunção WKG, Garcia A, Pereira JA, Kalinowski M, et al. On the Usefulness of Automatically Generated Microservice Architectures. *IEEE Transactions on Software Engineering*. 2024;50(3):651–667. Available from: https://dx.doi.org/10.1109/tse.2024.3361209.
6) Yin K, Du Q. On Representing Resilience Requirements of Microservice Architecture Systems. *International Journal of Software Engineering and Knowledge Engineering*. 2021;31(06):863–888. Available from: https://dx.doi.org/10.1142/s0218194021500261.
7) De Iasio A, Zimeo E. Avoiding Faults due to Dangling Dependencies by Synchronization in Microservices Applications. In: 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE. 2020;p. 169–176. Available from: https://doi.org/10.1109/ISSREW.2019.00068.
8) Sedghpour MRS, Garlan D, Schmerl B, Klein C, Tordsson J. Breaking the Vicious Circle: Self-Adaptive Microservice Circuit Breaking and Retry. In: 2023 IEEE International Conference on Cloud Engineering (IC2E). 2023;p. 32–42. Available from: https://doi.ieeecomputersociety.org/10.1109/IC2E59103.2023.00012.
9) Aderaldo CM, Mendonca ND. How The Retry Pattern Impacts Application Performance: A Controlled Experiment. In: SBES '23: Proceedings of the XXXVII Brazilian Symposium on Software Engineering. 2023;p. 47–56. Available from: https://doi.org/10.1145/3613372.3613409.
10) Wagner L. Simulating Scenario-based Chaos Experiments for Microservice Architectures. Universitätsstraße 38, D–70569 Stuttgart, Germany. 2021. Available from: https://elib.uni-stuttgart.de/bitstream/11682/12384/1/Wagner2021SimulatingScenarioBasedChaosExperimentsForMicroserviceArchitectures.pdf.

11) Sun X, Cui B, Cai Z. Deep Q-Learning Based Circuit Breaking Method for Micro-services in Cloud Native Systems. In: CCF Conference on Computer Supported Cooperative Work and Social Computing;vol. 2012 of Communications in Computer and Information Science. Singapore. Springer. 2024;p. 348–362. Available from: https://doi.org/10.1007/978-981-99-9637-7_26.
12) Escalona MJ, Mayo FD, Majchrzak TA, Monfort V. Web Information Systems and Technologies. Springer Nature. 2020. Available from: https://books.google.co.in/books?id=qNXKDwAAQBAJ&dq=Escalona+MJ,+Mayo+FD,+Majchrzak+TA,+Monfort+V.+Web+Information+Systems+and+Technologies.&lr=&source=gbs_navlinks_s.
13) O'Neill V, Soh B. Orchestrating the Resilience of Cloud Microservices Using Task-Based Reliability and Dynamic Costing. In: 2022 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE). IEEE. 2023;p. 1–6. Available from: https://doi.org/10.1109/CSDE56538.2022.10089320.
14) Benz SL, Kuhlmann J, Schreckenberg D, Wothge J. Contributors to Neighbour Noise Annoyance. *International Journal of Environmental Research and Public Health*. 2021;18(15):1–14. Available from: https://dx.doi.org/10.3390/ijerph18158098.
15) Frank S, Wagner L, Hakamian A, Straesser M, Van Hoorn A. MiSim: A Simulator for Resilience Assessment of Microservice-Based Architectures. In: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS). IEEE. 2023;p. 1014–1025. Available from: https://doi.org/10.1109/QRS57517.2022.00105.