DOI 10.31891/2307-5732-2024-331-71
УДК 004.005:004.9

DOBROVOLSKY YURIY
Chernivtsi National University named after Yu. Fedkovicha
https://orcid.org/0000-0002-1248-3615
e-mail: y.dobrovolsky@chnu.edu.ua
PROKHOROV PAVLO
Chernivtsi National University named after Yu. Fedkovicha
https://orcid.org/0009-0008-0965-5771
e-mail: prokhorov.pavlo@chnu.edu.ua

LIGHTWEIGHT ENVIRONMENTS FOR TESTING SPEED AND RELIABILITY OF SOFTWARE BASED ON OPERATING SYSTEM LEVEL VIRTUALIZATION

An environment for software performance testing is described, which is based on the use of containerization technology using appropriate software (Docker or other) on the one hand, and a tool to investigate its reliability by measuring performance and carrying out load testing, such as ApacheBench or similar. The advantage of containerization technology for achieving the goal was evaluated. It is shown that in this case, the test environment can be deployed on different computers separately, rather than being deployed centrally. This provides following advantages: less cost in terms of computing resources compared to other ways of deploying environments on local computers. Certain types of environment configurations can be easily distributed and replicated between individual computers and workstations through the distribution and application of prepared configuration files. Compared to cloud environments, this approach does not require additional material costs for support. The scheme of operation of such an environment for non-functional testing has been analyzed.. The environment also allows you to simulate more complex configurations, for example, to study the operation of various load balancing algorithms/systems. The created testing environment was tested on the example of performance of testing web applications. It showed that the created environment allows you to calculate the approximate response time of the application and the presence of failures depending on the number of users who use the application at the same time. Also, load testing allows you to calculate the normal and critical number of application users and predict and prevent potential failures. The mechanism for creating test environments considered in this work can be used to build models that are necessary for studying and/or modeling certain types of software testing based on loads, load balancing algorithms between workstations, and others. Since such an approach can be used on almost any personal workstation with relatively small requirements for computing resources and the deployment and distribution of environments is quite simple, such an approach can be implemented not only in business, but also in universities for courses that involve the use similar environments.

Keywords: software performance testing, software engineering, OS-level virtualization, software reliability,containers.

.

ДОБРОВОЛЬСЬКИЙ ЮРІЙ, ПРОХОРОВ ПАВЛО
Чернівецький національний університет ім. Ю. Федьковича

ЛЕГКОВАГІ СЕРЕДОВИЩА ДЛЯ ТЕСТУВАННЯ ШВИДКОДІЇ ТА НАДІЙНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ НА ОСНОВІ ВІРТУАЛІЗАЦІЇ РІВНЯ ОПЕРАЦІЙНОЇ СИСТЕМИ

Створено середовище для нефункціонального тестування програмного забезпечення, засноване на використанні технології контейнеризації з одного боку, та інструменту для швидкодії та проведення тестування на надійність, а саме - на навантаження типу Apache Bench, або аналогічного. Створене середовище дозволяє розрахувати орієнтовний час відповіді додатку і наявність відмов в залежності від кількості користувачів, які одночасно користуються додатком, а також дозволяє розрахувати нормальну і критичну кількість користувачів додатку та спрогнозувати і запобігти потенційним відмовам.

Ключові слова:функціональне тестування, програмна інженерія, контейнеризація, швидкодія ПЗ, надійність ПЗ, навантаження

Introduction

During the software development process, it is necessary to verify its reliability and quality. According to the ISTQB definition, the quality and stability of any software are defined by matching the requirements given in the specifications and other requirements definition documents. Such requirements are usually divided into functional and non-functional requirements. Functional requirements are usually related to functionality, which in many cases means matching business logic rules. Non-functional requirements encompass all other requirements, including performance parameters. Some software products are designed to handle large numbers of simultaneous connections and have short response times. Load and stress testing approaches should be used to test software under development or maintenance to meet these requirements. Most recent software is designed with a large number of users in mind, which is why testing its performance is an important task during software engineering and development.

Related work

General definitions and characteristics of performance testing types and their relationship are given in [1]. Conduction of different performance testing types requires special testing environments deploying, for which the balance between complexities and costs of environment deployment and deployment of developed software in this environment are very important. Another important question is how to make testing environments as much similar to real ones as possible, to make sure that results of testing using special testing environments could be useful in terms of real load predictions. These issues are described in detail in [2-4].

Performance testing methodology requires using special performance testing software like Apache JMeter/Gatling/Locust and many others. Architecture, components and general principles of work for such tools are described in [5].

Commercial software development nowadays is being done using widely known agile software development methodologies, such as XP, Scrum or Agile. The Agile methodology is designed to make minimal viable products as quickly as possible through series of small working time periods which called sprints. Recent studies [6] reveal that performance testing integration into teams and projects that use Agile could be difficult, because the methodology itself doesn't give direct approaches of how and when performance testing should be conducted and why it should be performed before the end of current sprint. That's why Implementation of efficient performance testing in such conditions require additional costs.

Recent decades software development of web applications in most of cases follow microservice architectures [7], which assume splitting big software modules into small modules with single or small number of purposes. Instead of older monolith architectures approach, microservices allow to achieve better robustness, scalability, and transparency of software. But it also multiplies the amount of performance testing work, because in practice you need not only to test single modules, but some set of working simultaneously and the software as whole.

It is worth to mention that during modern software development there are some not obvious practices that also could affect performance of final solution. The first one is runtime performance, because modern compilers and virtual machines are also very complicated software, performance testing in practice should consider some specific runtime traits, as it shown in [8, 9] according to JVM special performance downscale examples. The second one is the process of software refactoring, that usually means syntax and structure code changes that should increase the performance, readability and transparency of software. However, there are numerous examples of cases when changes should not affect the performance of software product, but actually did it [10]. That is why for successful development performance tests, just as any other, should be conducted for any code changes being made, even in cases when it is not clearly obvious that they could affect in any way.

Also, during the last decade web-applications in most cases are deployed using cloud-computing infrastructures (PaaS, IaaS approaches), and not using local or dedicated hosting resources. Cloud-specific best practices for performance testing are shown in [11].

Thus, we conclude that the practice of modern software development involves testing almost continuously [12], which means there is a need for tools and reproducible practices [13] that allow you to organize this with minimal costs.

Aim of the study

As we can see from the review of modern publications, there are many problems in the field of software testing. Test resources work on the basis of real or cloud computing servers. However, especially in the case of high workloads, the cost of deploying and maintaining such environments ciould be quite high. And this is one of the main problems of modern non-functional software performance testing.

In this regard, the goal of the study is to create an environment for software performance testing, which consists in the simple deployment of test environments of sufficient complexity, with the help of which it is possible to simulate the load on the software and conduct performance testing checks.

To achieve this goal, we're going to investigate and complete following tasks:

1. Evaluate container technologies in terms of lightweight testing environments creation.
2. Study the convenience of using such environments
3. Check that proposed approach is suitable for performance testing tasks.

Methodology

In this paper, an approach based on the use of containerization technology and the use of appropriate software is proposed for the task of deploying a test environment (Docker/LXC/Podman etc.)[14] and ApacheBench performance and load testing tool [15] or any similar to it.

Let's consider the main properties of the proposed tools. Docker is a tool for isolated Linux-like containers hosting and management. In the current work, containers are used to solve the problem of relatively simple and fast preparation of a reproducible environment that can be tested for performance and failures. Apache Bench is a load testing tool for WEB-applications performance/stress/load testing, that could measure performance of application through sending large number of HTTP requests using concurrent worker processes to it. This happens until the total number of requests for what to do is exhausted (the parameter is set arbitrarily in the command line).

The main idea of the approach is to use containerization software to deploy test environments on different computers separately, instead of deploying it centrally. This provides the following benefits:

1. Using containerization is less expensive in terms of computing resources than other ways of deploying environments on local computers, such as using virtual machines [16].

2. Certain types of environment configurations can be easily distributed and replicated between individual computers and workstations by distributing and deploying prepared configuration files.

3. Compared to cloud environments, this approach does not require additional material costs for support.

So, as part of the proposed approach to deploy a test environment and conduct performance or load testing, the user should perform the following steps:

1. Install required software:
    - Container hosting software (Docker/LXC/Podman or any other that meets the requirements).

- Load testing tool (ApacheBench/Apache JMeter/Gatling or any other that meets the requirements).

2. In accordance with the tasks, the necessary containers with the appropriate software are launched with the help of prepared configuration files and/or command shell commands.

It is worth to mention that the existing containerization systems allow performing such complex configurations as creating a network of containers, limiting the computing resources of an individual container, and accordingly the software running in it, connecting containers to file storage, and others. This allows us to create test environment models of almost any complexity.

Next, use the load testing tool to apply the required load. The obtained results can be analyzed according to the set tasks.

Fig. 1 illustrates the operation scheme of this environment (Control system means any Load testing tool as set of modules responsible for generation load and load results monitoring and evaluation).

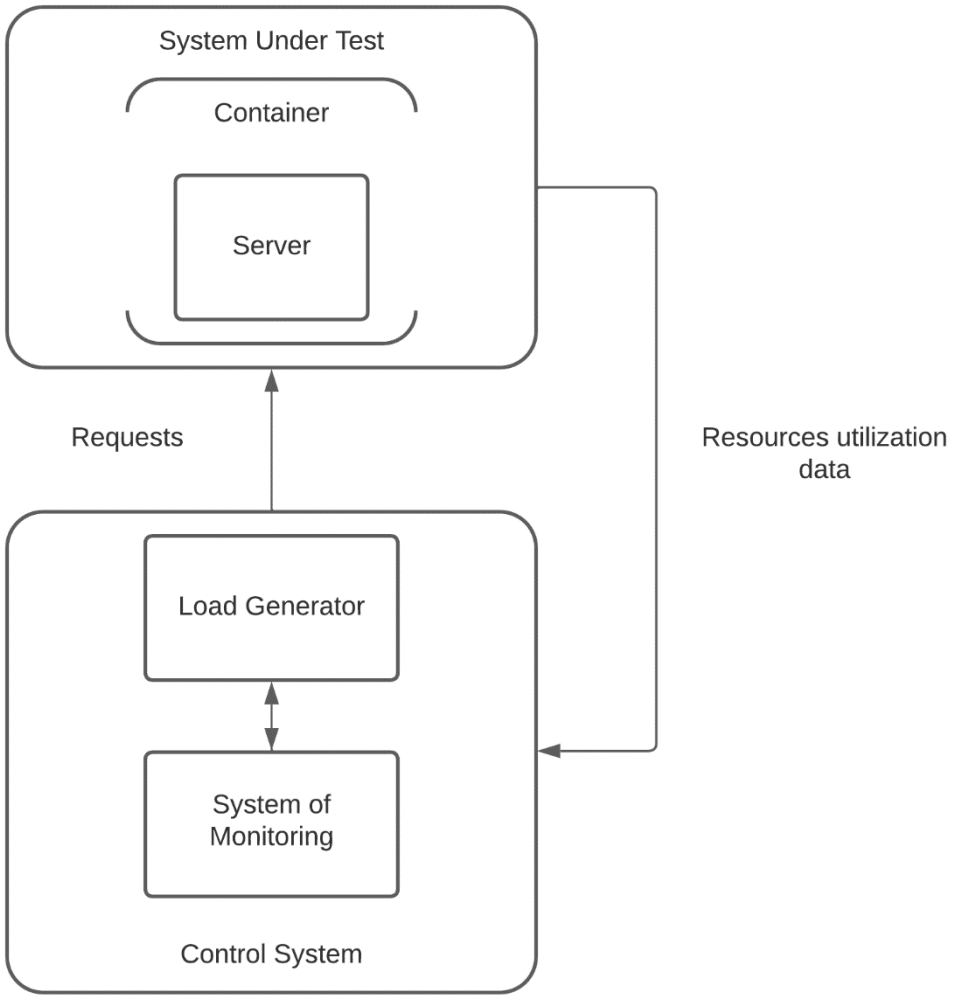

Fig. 1. Scheme of the test environment for load testing

Proposed approach can also be used to model more complex configurations, such as performance measurements of different load balancing algorithms and systems. [17]. Example of such environment is shown in Fig. 2.

To demonstrate the proposed technology, it is proposed to simulate failures during load testing. Namely: we deploy a certain web application with parameters that limit the RAM of the container and load it. To reproduce the case of failure two methods are proposed:

1. Limiting the computing resources of the container itself, which leads to the limitation of resources that server could utilize for processing requests.

2. Using an HTTP application with a server that is not optimized for high levels of loads (a standard HTTP server built into the Python standard library will be used as an example).

Next, we launch the Docker container with the image of the NGINX web server using the console command:

```
$ sudo docker run -it --rm -d -p 8080:80  --name web nginx
```

Start the load on the deployed application using the console command:

```
$ ab -n 2000 -c 200 -k  http://127.0.0.1:8080/
```

As a result of the load, we will receive the report shown in Fig. 3.

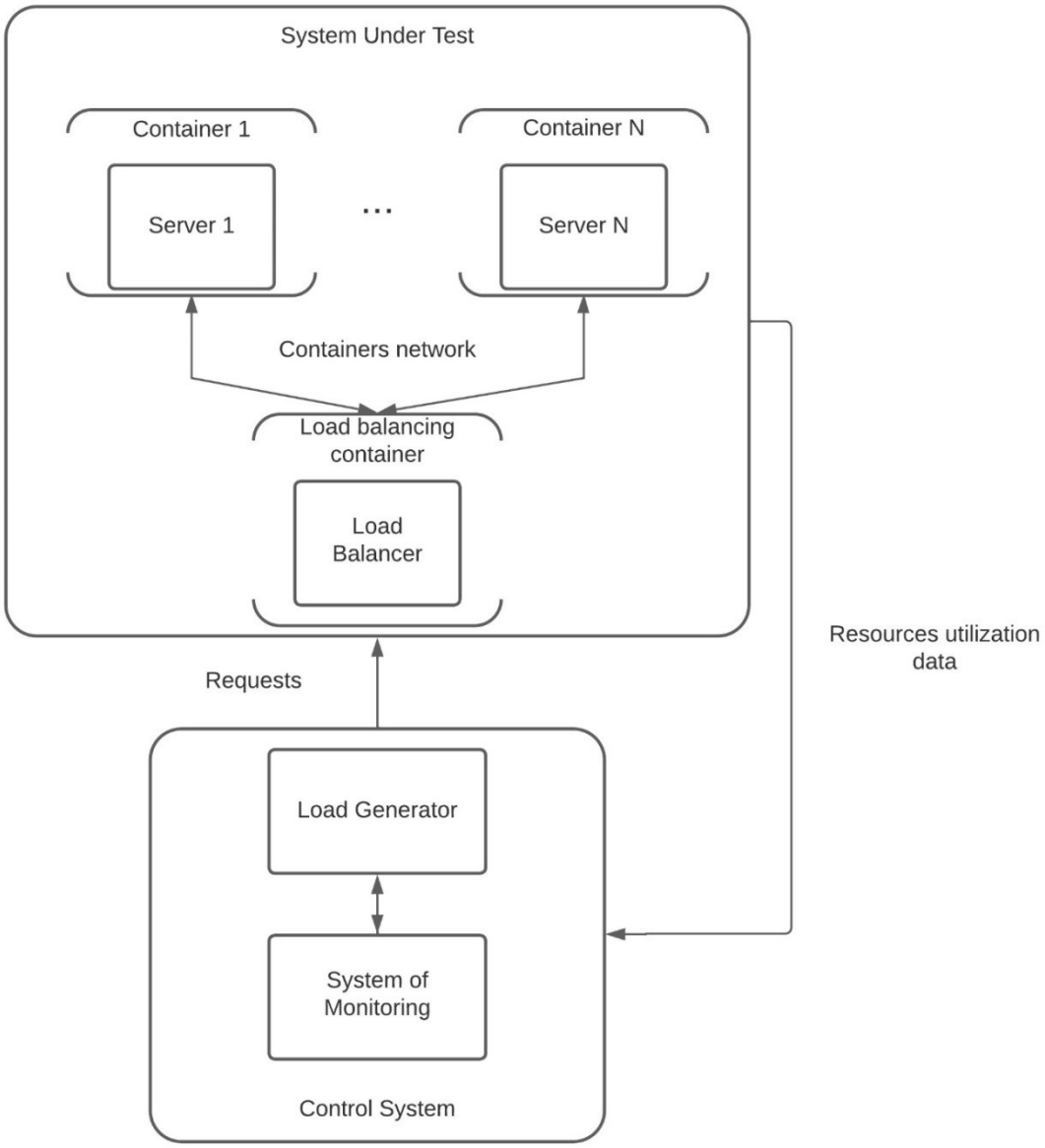

Fig. 2. Scheme of the test environment for load testing with load balancer

```
Server Software:        nginx/1.23.3
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /
Document Length:        615 bytes

Concurrency Level:      200
Time taken for tests:   0.303 seconds
Complete requests:      2000
Failed requests:        0
Keep-Alive requests:    2000
Total transferred:      1706000 bytes
HTML transferred:       1230000 bytes
Requests per second:    6601.73 [#/sec] (mean)
Time per request:       30.295 [ms] (mean)
Time per request:       0.151 [ms] (mean, across all concurrent requests)
Transfer rate:          5499.29 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    1   1.9      0       9
Processing:     0   27  32.8     19     191
Waiting:        0   27  32.8     19     191
Total:          0   28  34.2     19     197

Percentage of the requests served within a certain time (ms)
  50%     19
  66%     24
  75%     28
  80%     33
  90%     55
  95%    132
  98%    152
  99%    160
 100%    197 (longest request)
```

Fig. 3. Load testing report

As we could see, in the line "Failed requests" we have the result "0", Which means that during our load all requests have been handled without errors. In order to simulate the case of failure we limited amount of computing resources that container could utilize for handling requests (8 MB RAM and 2 CPU cores) and launched it:

$ sudo docker run -it --rm -d -p 8080:8000 -m 6m -c 2 --name web nginx

We start the load on the deployed application using the console command

$ ab -n 2000 -c 200 -k http://127.0.0.1:8080/

As a result of the load, we will get the result shown in Fig. 4. where you can clearly see that there were failures (2787 requests returned with an error). Also, the waiting time for response has increased. These failures, which in this case took the form of the unavailability of the server's response at the specified time, were caused by the technical imperfection of the server and the load that was set.

```
Server Software:        nginx/1.25.3
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /
Document Length:        615 bytes

Concurrency Level:      200
Time taken for tests:   0.328 seconds
Complete requests:      2000
Failed requests:        2787
   (Connect: 0, Receive: 0, Length: 1586, Exceptions: 1201)
Keep-Alive requests:    435
Total transferred:      371055 bytes
HTML transferred:       267525 bytes
Requests per second:    6091.15 [#/sec] (mean)
Time per request:       32.834 [ms] (mean)
Time per request:       0.164 [ms] (mean, across all concurrent requests)
Transfer rate:          1103.59 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    4   3.0      4      12
Processing:     0   27  50.4      6     240
Waiting:        0   13  41.5      0     184
Total:          0   32  50.4     11     243

Percentage of the requests served within a certain time (ms)
  50%     11
  66%     13
  75%     26
  80%     33
  90%     89
  95%    185
  98%    189
  99%    242
 100%    243 (longest request)
```

Fig. 4. Load testing report with failures

Next, we could make tests with different numbers of workers and analyze it to test different hypotheses and metrics, set in the form of rules of the type "no more than n seconds for handle request and send response". The results of such tests are given in the tables 1 - 3.

Table. 1

Load test results with different number of concurrent workers (25, 50 and 100 respectively). The percentage of requests falling within the limit of not more than 2 second of response time

System Under Test load parameters (ApacheBench CLI)	NGINX (default settings)	NGINX (mem 8 MB, CPU 2)	SimpleHTTP (default settings)	SimpleHTTP (mem 8 MB, CPU 2)
-n 100, -c 25, -s 180	99%	97%	96%	88%
-n 100, -c 50, -s 180	99%	94%	90%	80%
-n 100, -c 100, -s 180	99%	90%	87%	60%

Table. 2

Load test results with different number of concurrent workers (25, 50 and 100 respectively). The percentage of requests falling within the limit of not more than 2 second of response time

System Under Test load parameters (ApacheBench CLI)	NGINX (default settings)	NGINX (mem 8 MB, CPU 2)	SimpleHTTP (default settings)	SimpleHTTP (mem 8 MB, CPU 2)
-n 100, -c 25, -s 180	99%	99%	99%	95%
-n 100, -c 50, -s 180	99%	97%	95%	90%
-n 100, -c 100, -s 180	99%	96%	94%	70%

Table. 3

Load test results with different number of concurrent workers (25, 50 and 100 respectively). The percentage of requests falling within the limit of not more than 2 second of response time

System Under Test load parameters (ApacheBench CLI)	NGINX (default settings)	NGINX (mem 8 MB, CPU 2)	SimpleHTTP (default settings)	SimpleHTTP (mem 8 MB, CPU 2)
-n 100, -c 25, -s 180	99%	99%	99%	99%
-n 100, -c 50, -s 180	99%	99%	99%	98%
-n 100, -c 100, -s 180	99%	99%	99%	90%

As a result of testing, taking into account the data in the previous table, the following rules could be noted (for example, 2 rules are given):

- For 100 simultaneous users (100 simultaneous connections), the response time of any operation in 99% of cases should not exceed 4 seconds when using NGINX with standard settings. The probability of an error (server response code other than 200) should not exceed 0.1% of all cases.
- For 25 simultaneous users (25 simultaneous connections), the response time of any operation in 99% of cases should not exceed 4 seconds when using SimpleHTTP with standard settings. The probability of an error (server response code other than 200) should not exceed 0.1% of all cases.

The mechanism for creating test environments considered in this work can be used to build models that are necessary for studying and/or modeling certain types of software testing based on loads, load balancing algorithms between workstations, and others. Since such an approach can be used on almost any personal workstation with relatively small requirements for computing resources and the deployment and distribution of environments is quite simple, such an approach can be implemented not only in business, but also in universities when studying disciplines that involve the use similar environments.

The work was carried out as part of the scientific work of the department of software of computer systems of the Chernivtsi National University named after Yu. Fedkovich on the cathedral topic "Research, modeling and software development of complex dynamic systems". The results of the work are implemented in the educational process during the study of the course "Software Reliability Engineering".

Conclusion

1. An environment for performance software testing has been created, based on the use of containerization technology and the use of appropriate software (Docker or others) on the one hand, and a tool for measuring performance and carrying out load testing such as ApacheBench, or similar.

2. The advantage of containerization technology for achieving the goal was evaluated. It is shown that in this case, the test environment can be deployed on different computers separately, rather than being deployed centrally. This does provide advantages: less cost in terms of computing resources compared to other ways of deploying environments on local computers. Certain types of environment configurations can be easily distributed and replicated between individual computers and workstations through the distribution and application of prepared configuration files. Compared to cloud environments, this approach does not require additional material costs for support.

3. The scheme of operation of such an environment for performance testing has been analyzed. It is shown that the combination of load generation with the monitoring system and load testing tools are integrated into the control system as well. The environment also allows you to simulate more complex configurations, for example, to study the operation of various load balancing algorithms/systems.

4. The created performance testing environment was verified using the example of testing web applications. It showed that the created environment allows you to calculate the approximate response time of the application and the presence of failures depending on the number of users who use the application at the same time. Also, load testing allows you to calculate the normal and critical number of application users and predict and prevent potential failures.

References

1. Keith Yorkston: “Performance Testing: An ISTQB Certified Tester Foundation Level Specialist Certification Review”. – Apress Berkeley, CA – 2021. DOI: https://doi.org/10.1007/978-1- 4842-7255-8

2. Avramenko A.S., Avramenko V.S., Koseniuk H.V. Testuvannia prohramnoho zabezpechennia. Navchalnyi posibnyk. – Cherkasy: ChNU imeni Bohdana Khmelnytskoho, 2017. – 284 s.

3. Hrytsiuk Yu. I. Systema kompleksnoho otsiniuvannia yakosti prohramnoho zabezpechennia. Naukovyi visnyk NLTU Ukrainy. 2022. № 2(32). S. 81–95 s.

4. Ushakova I. O. Pidkhody do zabezpechennia yakosti prohramnoho zabezpechennia. Suchasni informatsiini tekhnolohii i systemy : monohrafiia. Kharkiv : «Stylizdat», 2021. S. 125–140.

5. Sarojadevi H. Performance Testing: Methodologies and Tools. Journal of Information Engineering and Applications Vol 1, No.5, 2011

6. Traini L (2022) Exploring performance assurance practices and challenges in agile software development: an ethnographic study. Empir Softw Eng 27(3):74. https://doi.org/10.1007/s10664-021-10069-3

7. V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," in IEEE Access, vol. 11, pp. 88339-88358, 2023, doi: 10.1109/ACCESS.2023.3305687

8. Traini L, Di Pompeo D, Tucci M, Lin B, Scalabrino S, Bavota G, Lanza M, Oliveto R, Cortellessa V (2021) How software refactoring impacts execution time. ACM Trans Softw Eng Methodol 31(2). https://doi.org/10.1145/3485136

9. Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, Michele Tucci. (2022). Towards effective assessment of steady state performance in Java software: are we there yet? Empirical Software Engineering, № 1. https://doi.org/10.1007/s10664-022-10247-x

10. Martin Grambow, Christoph Laaber, Philipp Leitner, David Bermbach. (2021). Using application benchmark call graphs to quantify and improve the practical relevance of microbenchmark suites, PeerJ Computer Science, p. e548. https://doi.org/10.7717/peerj-cs.548

11. D. Taibi, V. Lenarduzzi, Claus Pahl. Architectural Patterns for Microservices: A Systematic Mapping Study. CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018

12. Ding Z, Chen J, Shang W (2020) Towards the use of the readily available tests from the release pipeline as performance tests: are we there yet? In: Rothermel G, Bae D (eds) ICSE '20: 42nd international conference on software engineering, Seoul, South Korea, 27 June–19 July, 2020. https://doi.org/10.1145/3377811.3380351. ACM, pp 1435–1446

13. Papadopoulos A V, Versluis L, Bauer A, Herbst N, von Kistowski J, Ali-Eldin A, Abad C L, Amaral J N, Tuma P, Iosup A (2021) Methodological principles for reproducible performance evaluation in cloud computing. IEEE Trans Softw Eng 47(8):1528–1543. https://doi.org/10.1109/TSE.2019.2927908

14. Satya Bhushan Vermaa, Brijesh Pandeyb, and Bineet Kumar Gupta: "Containerization and its Architectures: A Study". – ADCAIJ: Advances in Distributed Computing and Artificial Intelligence Journal – Regular Issue, Vol. 11 N. 4 (2022), 395-409. DOI: https://doi.org/10.14201/adcaij.28351

15. Measuring performance with Apache Benchmark Mar 15, 2022. URL: https://barryvanveen.nl/articles/78224837-measuring-performance-with-apache-benchmark

16. Amit M. Potdar, Narayan D. G., Shivaraj Kengondc, Mohammed Moin Mullad: "Performance Evaluation of Docker Container and Virtual Machine". – Procedia Computer Science – Volume 171, 2020, Pages 1419-1428. DOI: https://doi.org/10.1016/j.procs.2020.04.152

17. Dalia Abdulkareem Shafiq, N.Z. Jhanjhi, Azween Abdullah: "Load balancing techniques in cloud computing environment: A review". – Journal of King Saud University – Computer and Information Sciences 34 (2022) 3910–3933. DOI:https://doi.org/10.1016/j.jksuci.2021.02.007