Check for updates

Received: 9 August 2019 | Revised: 1 April 2020 | Accepted: 9 May 2020

DOI: 10.1002/smr.2276

SPECIAL ISSUE - EMPIRICAL PAPER

Journal of Software: Evolution and Process WILEY

Towards reducing the time needed for load testing

Hammam M. AlGhamdi[1] | Cor-Paul Bezemer[2] | Weiyi Shang[3] | Ahmed E. Hassan[1] | Parminder Flora[4]

[1]Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Ontario, Canada

[2]Analytics of Software, Games and Repository Data (ASGAARD) lab, University of Alberta, Edmonton, Alberta, Canada

[3]Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada

[4]BlackBerry Limited, Canada

Correspondence
Cor-Paul Bezemer, Analytics of Software, Games and Repository Data (ASGAARD) lab, University of Alberta, Edmonton, Alberta, Canada.
Email: bezemer@ualberta.ca

Abstract

The performance of large-scale systems must be thoroughly tested under various levels of workload, as load-related issues can have a disastrous impact on the system. However, load testing often requires a large amount of time, running from hours to even days. In our prior work, we reduced the execution time of a load test by detecting repetitiveness in individual performance metric values, such as CPU utilization, that are observed during the test. However, as we explain in this paper, disregarding combinations of performance metrics may miss important information about the load-related behavior of a system. In this paper we revisit our prior approach, by proposing an approach that reduces the execution time of a load test by detecting whether a test no longer exercises new combinations of the observed performance metrics. We study three open source systems, in which we use our new and prior approaches to reduce the execution time of a 24-hour load test. We show that our new approach is capable of reducing the execution time of the test to less than 8.5 hours, while preserving a coverage of at least 95% of the combinations that are observed between the performance metrics during the 24-hour tests.

KEYWORDS

load testing, performance analysis, performance testing

1 | INTRODUCTION

Performance issues in ultra-large-scale systems can cause impactful software failures. For example, due to a memory leak, Amazon Web Services experienced an outage that affected millions of users.[1] Such performance failures can cause significant financial and reputational consequences.

Performance-related failures can often be avoided by properly testing the performance of a system under various levels of load before deploying it. However, load testing is expensive in terms of execution time: A load test can take hours or even days to complete.[2] In the current DevOps age of continuous deployment, it is important to make effective use of the available testing time. In addition, many software developers do not want to spend much time on load tests.[3]

There exists a point in time during a load test at which the cost-effectiveness of the test is the highest, that is, if the test continues to execute after this point, the amount of new information that is generated during the test is too small to be beneficial. Determining the most cost-effective length of a load test facilitates the load testing process, thereby reducing the probability of a load-related issue in a system.

In our prior work,[4] we proposed to reduce the time that is required to run a load test, by recommending to stop when performance metric values that are monitored during a test become repetitive. The disadvantage of this approach is that it focuses on repetitiveness of each metric individually. Each metric shows only one aspect of a system's performance. Instead, the performance of a system should be captured by a combination of all of those aspects.[5] Our prior approach does not capture such combinations, despite their importance for capturing the performance of a system.

 © 2020 John Wiley & Sons, Ltd.

In this paper, we revisit our prior approach[4] by focusing on combinations of performance metrics, using the so-called *system states*. Throughout the execution of a load test, the system goes through a variety of system states. A system state is a unique combination of ranges of performance metric values. For example, a system state can be the combination of low CPU utilization and high memory usage. Intuitively, as the execution of the test progresses, it exercises system states that are already exercised in an earlier stage of the test. Hence, the longer the test is executing, the less likely it becomes that new system states are being exercised. As a result, the cost-effectiveness of a load test, that is, the number of new states that are exercised in a time frame, goes down as the test progresses.

Our proposed approach periodically monitors how many new system states a load test exercises. Our approach recommends to stop the test when we no longer observe new system states in a given time frame during the test. We evaluate our approach by conducting experiments on three open-source systems (CloudStore,[6] PetClinic,[7] and Dell DVD Store 2[8]), in which we use our approach to reduce the execution time of a 24-h load test. We find that our approach recommends to stop the tests within 8 hours, which is a reduction of approximately 70% of the original 24-h execution time. At the recommended stopping times, the tests were able to exercise more than 95% of the system states that are exercised by the 24-h test.

In summary, the main contributions of our paper are as folows:

1 An approach for reducing the execution time of a load test, based on checking when a test no longer exercises new system states in a given time frame.
2 A comparison in which we show that our new approach recommends more cost-effective stopping times than our prior approach.

While our approach works well on smaller systems, we encountered several challenges when trying to adopt our approach in an industrial setting. We documented the challenges that we experienced to call upon the community to assign more research and practical efforts to solving the important industrial problem of making load tests more efficient in terms of time.

This paper is organized as follows. Section 2 discusses our prior approach and related work. Section 3 presents a motivational example for our approach. Section 4 describes our approach. Section 5 describes our experimental setup, followed by the results of our experiments in Section 6. We discuss the challenges of applying our approach in an industrial setting in Section 7. Section 8 discusses threats to the validity. We conclude the paper in Section 9.

2 | BACKGROUND

In this section, we discuss our prior approach for determining when to stop a load test. In addition, we discuss related work that addresses the challenges of having limited resources for performance or load testing. Note that despite their industrial significance, only a very small body of work exists on addressing these challenges.

2.1 | Our prior approach for determining when to stop a load test

Our prior work[4] aims at reducing a test's execution time by determining how long a load test should run. Our prior approach is based on the experience that much of the generated data from a load test are repetitive, when simulating a real-world workload. Therefore, if the load test would no longer provide any new information about the system's performance, the test can simply be stopped.

In our prior approach, we periodically measure repetitiveness in the values of performance counters that are already generated from the load test. In particular, we randomly pick a short time period (T_1) from the data and exhaustively search for another time period (T_2) that contains statistically indistinguishable data. Our prior approach leverages statistical analysis (e.g., the two-tailed unpaired Wilcoxon test) to determine whether the difference between the data in two time periods is statistically significant (p value <0.05). By the exhaustive search, if a time period can be found that contains statistically indistinguishable data, T_1 is considered as repetitive. In our prior approach, we repeat the process of picking a random time period and searching for a statistically indistinguishable time period a large number of times (e.g., 1000). Therefore, we can calculate the probability of a time period being repetitive. We use this probability as a measure of repetitiveness in the performance test results. The details of measuring repetitiveness are shown in Algorithm 1.

In order to use this measure of repetitiveness to determine whether to stop the performance test, practitioners may choose a threshold based on their own experiences. In addition, we proposed an automated approach that measures the first derivative of repetitiveness. Based on the first derivative of repetitiveness, our approach recommends to stop the performance test if the repetitiveness stabilizes.

As we show in Section 3, the limitation of our prior approach is that it does not consider combinations of performance metric values. In this paper, we revisit our prior approach to overcome this limitation.

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Algorithm 1 Algorithm for measuring repetitiveness in prior research[4]

```
Input  : Performance data in a list perfList
Output: RepetitivenessRatio
/ * Initialize the repetitiveness value * /
repetitiveness ← 0
for i<1000 do
    Data1 ← perfList.getRandomPeriod()
    / * Exhaustively search for repetitive data * /
    foundRepeat ← false
    foreach timePeriod in perfList do
        Data2 ← timePeriod
        / * Select new data if there is overlapping * / if Data1 overlaps Data2 then
            continue
        end
        if Data1.isSimilar(Data2) then
            foundRepeat ← true
            break
        end
    end
    / * Increase the repetitiveness if we find a similar period * /
    if foundRepeat then
        repetitiveness + +
    end
end
RepetitivenessRatio ← repetitiveness / 1000
return RepetitivenessRatio
```

2.2 | Related work

To reduce the resources that are required to run a performance or load test, we can either (a) reduce the execution time, (b) adapt the workload that drives a test, or (c) reduce the system resources that are required to execute the test. We discuss the related work for each method of reducing load testing resources below.

2.2.1 | Reducing the execution time of a load test

In addition to our prior work,[4] the work that comes closest to our work is that of He et al.[9] He et al. used a statistics-based performance testing approach that investigates whether the distribution of a performance metric is still changing after the execution of (part of) a test. Their approach focuses on dealing with the performance uncertainty in cloud applications. However, their approach studies the distribution of each performance metric individually (similar to our prior work). Jain[10] proposed an approach that recommends to stop a test when the variance in the response time (RT) is reduced to less than 5%. Jain's approach does not take into account the performance metrics about the physical level of a system, for example, the CPU utilization. Other studies on reducing the execution time of tests depend on logs. Busany and Maoz[11] and Jiang et al.[12] propose approaches that measure the repetitiveness of log traces in order to reduce the execution time of a test. The approach that we propose in this paper differs from prior work as we use combinations of metrics to decide how to reduce the execution time of a performance test. The advantage is that using combinations of metrics gives a more realistic view of the performance of a system.[5]

2.2.2 | Adapting the workload that drives a load test

Other work aims at adapting the execution time of a load test by dynamically adapting the workload that drives the test. For example, Tchana et al.,[13] Shivam et al.,[14] and Ayala-Rivera et al.[15] adjust the load on a system until the system is saturated. Apte et al.[16] use information that was retrieved from queueing models to plan the load on a system more efficiently. In our work, we do not adapt the load, but instead, we decide when a load no longer provides us with new information about the system. The advantage of our approach over approaches that adapt the workload

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

that drives a test is that our approach leaves the workload unchanged and, therefore, does not have the threat of adapting to a less realistic workload.

2.2.3 | Reducing the system resources required to execute a load test

Most work on load testing focuses on how to reduce the system resources that are required by the system under test to handle a given workload. Shariff et al.[17] presented an approach to reduce the system resources required to conduct a browser-based load test itself, that is, their work focuses on making the load test cluster more efficient. Their approach allows to share browser instances between simulated test users, thereby reducing the total number of browser instances that are required to simulate the required number of users. Grano et al.[18] presented an approach that generates functional test suites that are performance aware. Their approach generates test suites with a high coverage and low computational requirements at the same time. In contrast to our approach, Grano et al. focus on functional test suites.

In addition, many techniques on test case prioritization and selection have been proposed that allow to reduce the required system resources to execute a test by focusing on high-priority tests. As these techniques all focus on functional test suites, they are only tangentially related to load tests, and we do not give an overview of them. Instead, we refer the reader to the surveys by Yoo and Harman[19] and Khatibsyarbini et al.[20]

3 | A MOTIVATING EXAMPLE

Erica is a performance engineer who works for a large software company. Erica's task is to ensure that every new version of a software product satisfies the performance requirements. However, the company uses continuous delivery to deploy several new versions of the product a day. Ideally, a change to the software product is included as soon as possible in an upcoming release. Hence, the time it takes to test that change should be minimized.

To make more efficient use of the time that is available for testing, Erica uses our prior approach[4] to reduce the execution time of the existing load tests. However, Erica notices that some performance issues are missed by the tests. Table 1 shows parts of the performance metrics that are collected during the execution of the load test.

The table highlights three periods p_1, p_2, and p_3, which represent the system states under low load, high load, and being overloaded. Using our prior approach,[4] which searches for repetitiveness at the metric level, p_3 would be considered repetitive and therefore redundant, as the observed RT is similar to that of p_1 and the observed CPU utilization is similar to that of p_2.

However, p_3 clearly represents a state of the system, that is, an overloaded state, that is not observed earlier in the test. Table 1 shows that studying the repetitiveness based on the individual metric is not enough. Studying combinations of metrics can give more information about the test, such as the load type, for example, high or low load.

Therefore, Erica transforms the observed metrics in the metric states S_1 (low) and S_2 (high) and studies the combinations in which the states occur. Erica concludes that our prior approach[4] stops the test too early, as the system state S_1 for RT and S_2 for CPU is not covered. Instead, Erica now stops the load test when there are no new states exercised by the test anymore.

TABLE 1 A motivational example of a performance issue that is missed by a test of which the execution time is reduced using our prior approach.[4] As our prior approach detects repetition for each metric individually, period p_3 is considered repetitive (i.e., with p_1 for the response time (RT) metric and with p_2 for the CPU metric). The approach presented in this paper does not consider any of the periods repetitive

Time	RT	CPU	
100	11 (S_1)	10 (S_1)	
100	10 (S_1)	12 (S_1)	period p_1, low load
101	10 (S_1)	10 (S_1)	
⋮	⋮	⋮	
149	49 (S_2)	100 (S_2)	
150	50 (S_2)	100 (S_2)	period p_2, high load
151	50 (S_2)	100 (S_2)	
⋮	⋮	⋮	
199	11 (S_1)	100 (S_2)	
200	10 (S_1)	100 (S_2)	period p_3, overload
201	10 (S_1)	100 (S_2)	

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

In the remainder of this paper, we present our approach for deciding when to stop a load test based on the number of observed new system states in a given time frame.

4 | OUR APPROACH FOR DETERMINING WHEN TO STOP A LOAD TEST

Intuitively, we assume that during the execution of a long-running load test, there exists a point in time after which the test no longer provides new information about the performance of the system under test. Hence, the load test can be stopped at that point in time. The goal of our approach is to find this point in time after which the load test becomes repetitive.

In this section, we present our approach for determining when to stop a load test. Figure 1 gives an overview of our approach, whose steps are detailed in this section. Algorithm 2 depicts the pseudo code that describes our approach. First, we collect performance metrics from the system under test. We transform the collected metrics into metric states, that is, we group the metric values in ranges. We define a system state as the combination of metric states at a given time. Second, we monitor the number of newly exercised system states during the test execution. Once the approach finds that the test no longer exercises new system states in a given time frame, it recommends to stop the test. In the rest of this section, we explain our approach in detail.

The same comment as for Algorithm 1 applies to Algorithm 2.

Algorithm 2 Pseudo code of our new approach for determining whether to stop a load test

```
Input  : Performance data in a list perfList
Output: Whether to stop
/* Initialize the coverage as nothing */
coverage ← null
/* Get the latest performance data */
newData ← N last items in perfList
/* Get the existing performance data excluding the new data */
oldData ← perfList − newData
/* Get the existing exercised states in performance data */
foreach data in oldData do
    state ← data.transformToState()
    if coverage does not include state then
        add state to coverage
    end
end
/* Check if the new data exercises new states */
foundNew ← false
foreach data in newData do
    state ← data.transformToState()
    if coverage does not include state then
        foundNew ← true /* new data exercises new state */
        break
    end
end
/* If new data exercise new states, do not stop the test */
return !foundNew
```

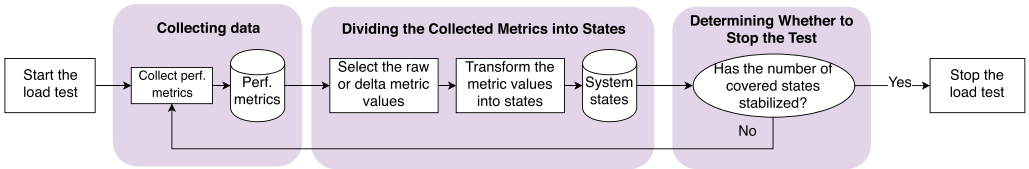

FIGURE 1 An overview of our approach

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4.1 | Collecting data

We start the load test and collect performance metrics every *s* seconds during the execution of the test. Our approach has no theoretical limitation on the number and selection of performance metrics that are collected. However, as we discuss in Section 7, selecting too many metrics can lead to a large number of possible states. In this chapter, we demonstrate our approach using the following metrics:

- Response time (RT): The average RT of the requests that were handled in the last *s* seconds.
- CPU utilization (CPU): The average CPU utilization in the last *s* seconds.
- Memory usage (MEM): The memory usage in the last *s* seconds.
- Disk I/O (IO): The average number of bytes that are read and written in the last *s* seconds.

Table 2 contains an example of metric values that are collected during 30 min of an imaginary load test. We use imaginary data in this section to improve the readability and clarity of our approach. We will use the example in Table 2 throughout this section to demonstrate our approach.

4.2 | Transforming the collected metrics into states

When running a load test, we are not particularly interested in the precise performance values that are exercised. Instead, we are interested in the range of values that are exercised. For example, the difference between 75% and 76% CPU utilization is minor, while the difference between 15% and 75% utilization is relevant. Therefore, we transform the performance metrics values into metric states, that is, ranges of values. The combination of these metric states at a given time represents a system state.

4.2.1 | Handling metrics with upward/downward trends

In some cases, a performance metric may show a trend, in which the exercised values are monotonically increasing or decreasing during the test. Such a monotonic increase or decrease in the values of a so-called *trended* metric has the consequence that the system is exercised with new performance behavior throughout the test. However, if the rate with which the metric increases or decreases is repetitive, the test exercises repetitive behavior, and therefore, we can stop the test.

To capture repetitiveness in the trend of a metric, we calculate the delta (△) values. The △ value of a metric is the difference between the current and the previous value of a metric. Figure 2 shows the raw and △ values for the memory usage metric from the example in Table 2, together with a nontrended metric (RT) for reference.

To determine whether to use the raw or delta values of a metric, we calculate the Spearman correlation between the metric values and the time.[21] We use the Spearman correlation because it is a nonparametric correlation measure, that is, it does not make assumptions about the distribution of the data. We calculate the Spearman correlation by using the `cor()` function in `R`. If the metric is correlated with the time (i.e., |correlation value|>0.5 and *p*value <0.05), we use the △ metric values. Otherwise, we use the raw metric values. For our working example in Table 2, the memory metric has |correlation value| = 1.00 and *p* value <0.05. Therefore, we use the △ values for the memory metric, whereas for the other metrics, we use raw values.

4.2.2 | Transforming the metric values into states

We use a binning algorithm (using the `bins()` function in `R`[22]) to transform the metric values into *n* states. A well-known usage of this algorithm is by the histogram (`hist()`) function, which uses the binning algorithm to divide a data set into bins to visualize its distribution. Figure 3 shows the division of metrics from the example in Table 2 into three states. The state boundaries of the states are the red vertical lines in the graphs.

4.3 | Determining whether to stop the test

We count the number of unique system states $state_{unique}$ that the test has exercised every τ minutes.

Early in the test, $state_{unique}$ increases rapidly. However, as the test progresses, the exercised system states become repetitive, and $state_{unique}$ stabilizes. Our approach recommends to stop the test when $state_{unique}$ no longer increases. It is important to realize that the total number of

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 An example of performance metrics and metric states

t (min)	Performance metric values					Metric state				System state exercised earlier?
	RT	CPU	MEM	△ MEM	IO	RT	CPU	△ MEM	IO	
1	65	43	46	46	69	S_3	S_2	S_2	S_3	
2	78	59	140	94	41	S_3	S_3	S_3	S_2	
3	36	32	193	53	12	S_2	S_2	S_3	S_1	
4	23	56	208	15	94	S_2	S_2	S_2	S_3	
5	39	10	302	94	13	S_2	S_1	S_3	S_1	
6	38	50	321	19	78	S_2	S_2	S_2	S_3	✓
7	16	100	330	9	36	S_1	S_3	S_1	S_2	
8	21	83	340	10	48	S_1	S_3	S_1	S_2	✓
9	2	13	354	14	75	S_1	S_1	S_2	S_3	
10	81	42	378	24	35	S_3	S_2	S_2	S_2	
11	16	88	391	13	66	S_1	S_3	S_1	S_3	
12	53	80	464	73	15	S_3	S_3	S_3	S_1	
13	63	23	561	97	48	S_3	S_1	S_3	S_2	
14	8	25	565	4	20	S_1	S_2	S_1	S_1	
15	36	64	640	75	92	S_2	S_3	S_3	S_3	
16	22	84	651	11	47	S_2	S_3	S_1	S_2	
17	87	84	704	53	61	S_3	S_3	S_3	S_2	✓
18	40	9	799	95	12	S_2	S_1	S_3	S_1	✓
19	18	89	813	14	65	S_1	S_3	S_2	S_3	
20	84	41	829	16	50	S_3	S_2	S_2	S_2	✓
21	7	25	834	5	21	S_1	S_2	S_1	S_1	✓
22	79	60	886	52	42	S_3	S_3	S_2	S_2	
23	38	21	944	58	13	S_2	S_1	S_3	S_1	✓
24	40	11	1039	95	14	S_2	S_1	S_3	S_1	✓
25	9	23	1044	5	19	S_1	S_1	S_1	S_1	
26	64	24	1140	96	49	S_3	S_1	S_3	S_2	✓
27	46	45	1187	47	70	S_3	S_2	S_2	S_3	✓
28	17	89	1201	14	67	S_1	S_3	S_2	S_3	✓
29	55	57	1217	16	93	S_3	S_2	S_2	S_3	✓
30	46	45	1260	43	70	S_3	S_2	S_2	S_3	✓

Note. Orange cells show the low-value metric state (i.e., S_1), blue cells show the medium-value metric state (i.e., S_2), and yellow cells show the high-value metric state (i.e., S_3).
Abbreviations: IO, disk I/O; MEM, memory usage; RT, response time.

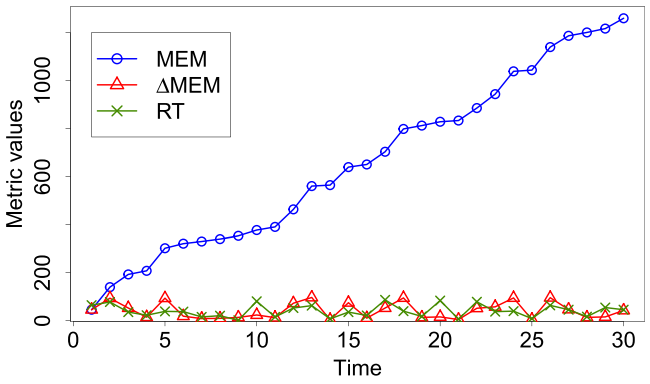

FIGURE 2 The memory usage (MEM), △ MEM, and response time (RT) metrics from the example in Table 2

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

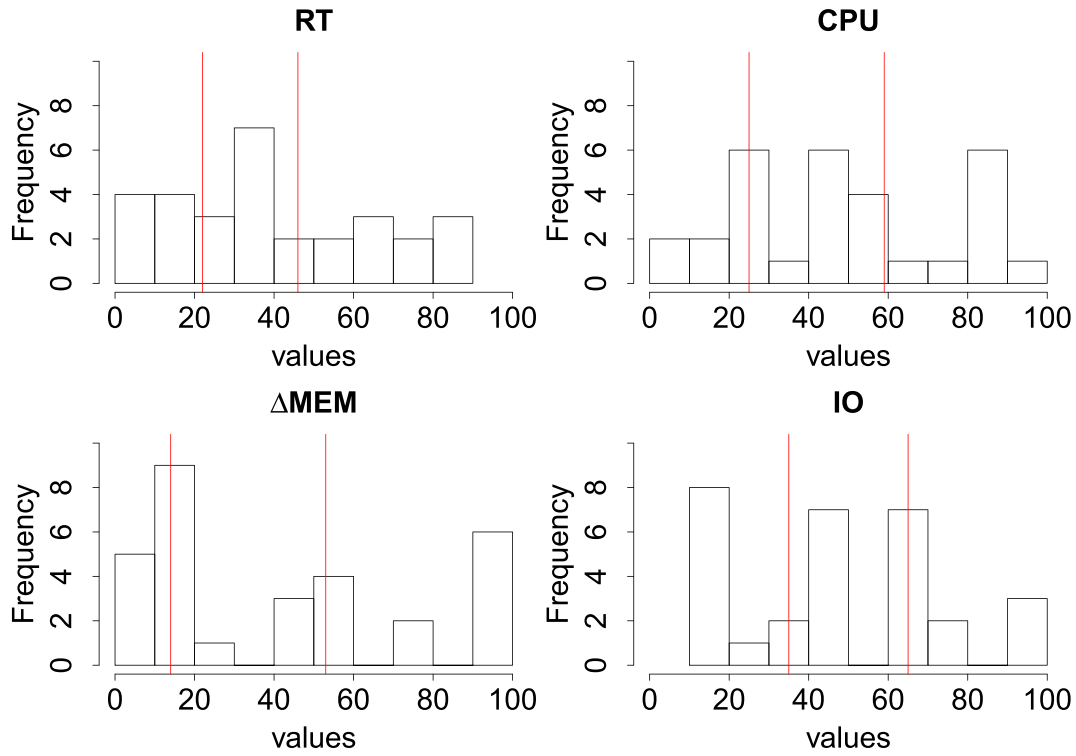

FIGURE 3 Transforming the values into states for the example from Table 2. The red vertical lines are the state boundaries determining the states. IO, disk I/O; MEM, memory usage; RT, response time

unique states that can be exercised by the test is not affected by the choice of τ. However, the sensitivity of when to stop the test is affected by this *choice* (e.g., for a large τ, it will take longer to decide to stop a test than for a small τ).

Table 3 shows $state_{unique}$ of the example in Table 2 for $\tau = 5$. $state_{unique}$ grows rapidly at the early minutes but stabilizes after $t = 25$. Our approach waits for the $state_{unique}$ for d minutes. For our working example, $d = 5$ min. Therefore, our approach recommends to stop the test at 30 min, as $state_{unique}$ did not change for 5 min.

5 | EXPERIMENTAL SETUP

We conduct an experiment on three open-source systems to evaluate our approach for reducing the execution time of a load test. In this section, the three systems are presented along with the workloads that are used in the test.

5.1 | Subject systems

We used the same subject systems as in our prior work,[4] to allow comparison of the results of our new and prior approaches. CloudStore[6] is an open-source e-commerce system, which follows the TPC-W performance benchmark standard.[23] CloudStore is built for performance

TABLE 3 The $state_{unique}$ for every 5 min in the example in Table 2

t	$state_{unique}$	Stop test?
5	5	No
10	8	No
15	13	No
20	15	No
25	17	No
30	17	Yes

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

benchmarking and contains 40.6k source lines of code (SLOC). PetClinic[7] is an open-source web system providing a simple yet realistic design of a web system. PetClinic (2k SLOC) was used in prior performance engineering studies.[24-26] Dell DVD store (DS2, 3.3k SLOC) is an open-source web system[8] that benchmarks new hardware system installations by simulating an electronic commerce system. DS2 was also used in prior performance engineering research.[27,28]

5.2 | Deployment of the subject systems

Our experimental environment consists of two Intel CORE i7 servers that run Windows 8 with 16 GB of RAM. We deploy the systems on one of the servers, whereas the other server is used to run the load driver. We use the same web server, that is, Tomcat 7.0.57,[29] and database, that is, MySQL 5.6.21,[30] for all subject systems.

5.3 | Load tests

To test our subject systems, we use workloads that mimic the real-life usage of the systems under test and cover all of the common features.[31] The workloads of PetClinic and CloudStore are exercised using Apache JMeter.[32] We used a workload that was used in prior work on performance engineering for PetClinic[24] and for CloudStore.[33] The DS2 system has its own load generator, which generates a workload. The load tests of all three subject systems are run for 24 h. We chose to run the tests for 24 h, because although there exist load tests that run longer, for practical reasons, 24 h is the maximum length we would like to assign to our industrial load tests. In addition, using 24 h for our experiments allows us to compare the approach that is presented in this paper with our prior approach.[4] We used two different types of load intensity: random and steady. During the DS2 test, the load intensity is randomly changed every 10 min. We used a steady load for CloudStore and PetClinic. This way, we evaluate our approach using two different types of load intensity.

We would like to emphasize that the purpose of our tests is not to reveal short spikes in performance or to stress test the subject systems. Instead, we test whether the systems can handle various expected levels of workload.

5.4 | Data collection

We collect two types of performance metrics during the execution of the load tests.

Physical metrics (CPU, MEM, and IO): The physical performance metrics measure the performance of the system during the load tests. In particular, we use a performance monitoring system, PerfMon,[34] to collect physical performance metrics.

Domain metric (RT): The domain performance metrics measure the performance of the system from the users' perspective. In our experiment, the DS2 load driver automatically calculates RT during the tests. We calculate RT in the CloudStore and PetClinic experiments by analyzing log files that are generated by JMeter during the tests.

It is challenging to analyze performance metrics that are generated from different sources. Because of the clock skew,[35] the data that are generated from PerfMon, JMeter log files, and the DS2's load generator may not be generated at exactly the same time. Those resources generate the performance metric measurements at every 10 s. For example, PerfMon may record the CPU utilization at 12:01 (i.e., 12 min and 1 s), 12:11 and 12:21, while the response time measurements are generated at 12:05, 12:15, and 12:25. To address this challenge, we calculate the average value of every three consecutive measurements, and every 30 s, we take the average of the metric values at 10, 20, and 30 s. Hence, we join the two metrics above by taking the average values of the metrics as the value for 12:30. The downside of this approach is that spikes in performance that are shorter than 30 s are hard to detect. However, as described in the previous subsection, detecting such short spikes is not the purpose of our tests.

5.5 | Parameters of our approach

During the experiment, we configure our approach with the following parameters:

1 n: The number of states in which we divide the metrics. In our experiments, we use $n = 3$: S_1, S_2, and S_3.
2 τ: The number of minutes, after which we measure $state_{unique}$ during the execution of the tests. We use $\tau = 10$.
3 d: The length of time that we wait for the test not exercising new system states. We use $d = 40$ min.

These parameter settings were chosen based on our experience with load testing in industry. The τ and d settings impact the sensitivity of our approach. We discuss the challenges of choosing n in Section 7.

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

6 | EXPERIMENTAL RESULTS

In this section, we present the results of the experiments that we conducted to evaluate our approach. In addition, we compare our approach with our prior work[4] that recommends to stop a load test based on repetitiveness at the performance metric level. In this section, we evaluate our approach by answering two research questions (RQs):

- RQ1: What is the state coverage at the stopping times that are recommended by our approach?
- RQ2: What is the relevance of the states that are not covered when stopping a load test early?

For each RQ, we discuss the motivation, approach, and results.

6.1 | RQ1: What is the state coverage at the stopping times that are recommended by our approach?

Motivation. The goal of our approach is to reduce the execution time of a load test without reducing the information that the test provides. Stopping a test later increases the chance of receiving more information from the test, that is, as it covers more system states, but comes at the cost of increased execution time of the test. On the other hand, stopping a test too early results in missing useful information about the systems' performance.

Our approach recommends to stop the execution of a performance test when it stops providing new information for *d* minutes. In this RQ, we evaluate the state coverage at the recommended stopping times.

Approach. We use the data of the 24-h test executions to evaluate the cost-effectiveness of the recommended stopping times. We calculate the state coverage by dividing $state_{unique}$ at the stopping time by $state_{unique}$ after exercising the tests for 24 h. In addition, we compare the state coverage at the stopping times that are recommended by our new and prior approach. Note that for each test execution, we recommend one stopping time.

Results. **Our approach reduces the execution time of the test by 70% while covering more than 95% of the system states that are exercised in the 24-h test.** Table 4 shows the stopping times that are recommended by our approach and the state coverage at those times. Table 4 shows that running the tests as long as the recommended stopping times will only yield 5% additional state coverage.

Once the number of unique states that are covered stabilizes, it stabilizes throughout the test. Figure 4 shows the $state_{unique}$ as the tests progress. In these graphs, the *x-axis* shows the time, and the *y-axis* shows the percentage of state coverage. Overall, while the tests are running, the increase rate of $state_{unique}$ decelerates over the time, until it stabilizes. We find that once the state coverage is stabilized, it remains stabilized until the end of the test.

Our prior approach misses information about the combinations of metrics for PetClinic and CloudStore. When stopping the test using our prior approach, some information that our new approach discovers is missed. Table 5 shows that stopping the tests at the stopping times that are recommended by our prior approach for CloudStore and PetClinic covers only 89% and 75% of the states, compared with 95% and 98% in our new approach. The stopping time that is recommended by our prior approach for DS2 covers exactly the same amount of information that the stopping time of our proposed approach does, that is, 95%, but with a delay of more than 2 h.

> *Our approach is capable of reducing the execution time of a 24-h load test for the three subject systems to 23%–34%, while preserving a state coverage of at least 95%.*

will this box be a screenshot? if so please improve the readability

TABLE 4 The recommended stopping times and the state coverage when applying our approach to the tests

Subject systems	# of Covered states[a]	Stopping time[b]	Coverage[a] (%)	Duration[b] (%)
CloudStore	81	7:30	96	31
PetClinic	81	8:10	98	34
DS2	66	5:30	95	23

[a]The number of states that are covered by the 24-h tests.
[b]Reported as hours:minutes.
[c]Percentage of all states that are exercised by the test at the stopping time.
[d]Percentage of the full execution time, that is, 24 h.

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

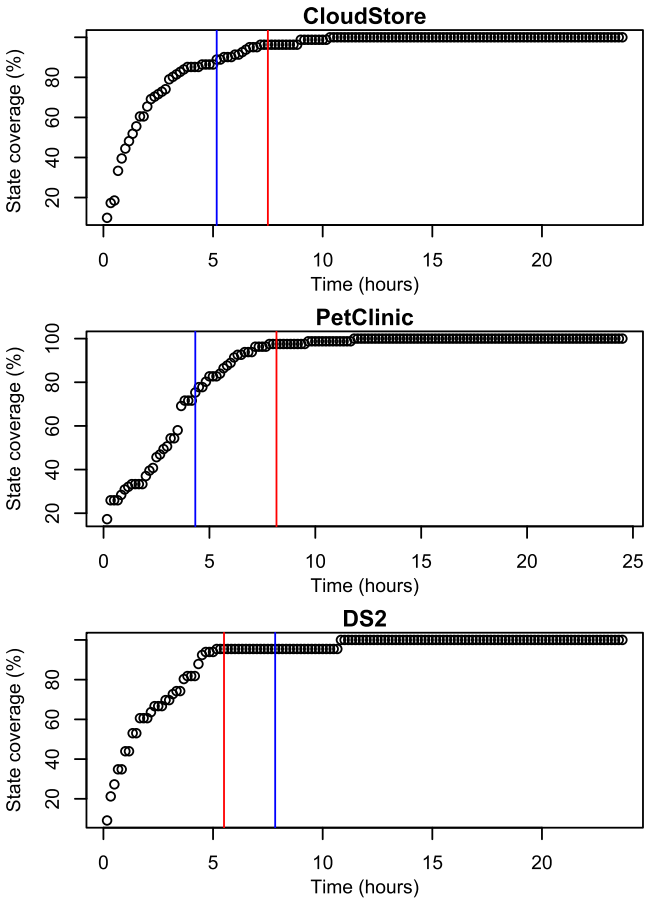

FIGURE 4 State coverage of our experiments over time. The red line is the stopping time that is recommended by our proposed approach. The blue line is the stopping time that is recommended by our prior approach[4]

TABLE 5 Comparison between the state coverage and the stopping times of new and prior[4] approaches

	Our approach		Our prior approach		
Subject systems	Stopping times	State coverage (%)	Stopping times		State coverage (%)
CloudStore	7:30	96	5:10	(earlier)	89
PetClinic	8:10	98	4:20	(earlier)	75
DS2	5:30	95	7:50	(later)	95

6.2 | RQ2: What is the relevance of the states that are not covered when stopping a load test early?

Motivation. Although our new approach recommends early stopping times at which a test achieves a high state coverage (more than 95%), there are still states that are missed by our approach but captured by the 24-h load tests. In this RQ, we study the relevance of these missed states.

Figure 5 shows an example of the distribution of the performance metrics. In each distribution, two missed system states, that is, m_1 and m_2, are indicated by the blue lines. The MEM value for m_1 is close to a state boundary, that is, inside the light gray background area. The state, into which a metric value that is close to a state boundary is indicated, might be affected by a few outliers of metric values as such outliers can slightly change the state boundaries. Therefore, such borderline values might be less relevant if one of the states that the value is close to is already covered.

Approach. To check the relevance of the missed states, we categorize them into two types of states:

- *Borderline states*: states of which at least one of their metric values is close to a state boundary.
- *Clearly identified states*: states of which none of their values is close to a state boundary.

We consider a metric value close to a state boundary when the difference between the value and the state boundary is less than 5% of the standard deviation (*std*) of the entire metric distribution. The solid red lines in Figure 5 are the state boundaries for the metrics, which transform the metric values into states. The dashed red lines indicate 5% of the *std* around the state boundaries, which we use to

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

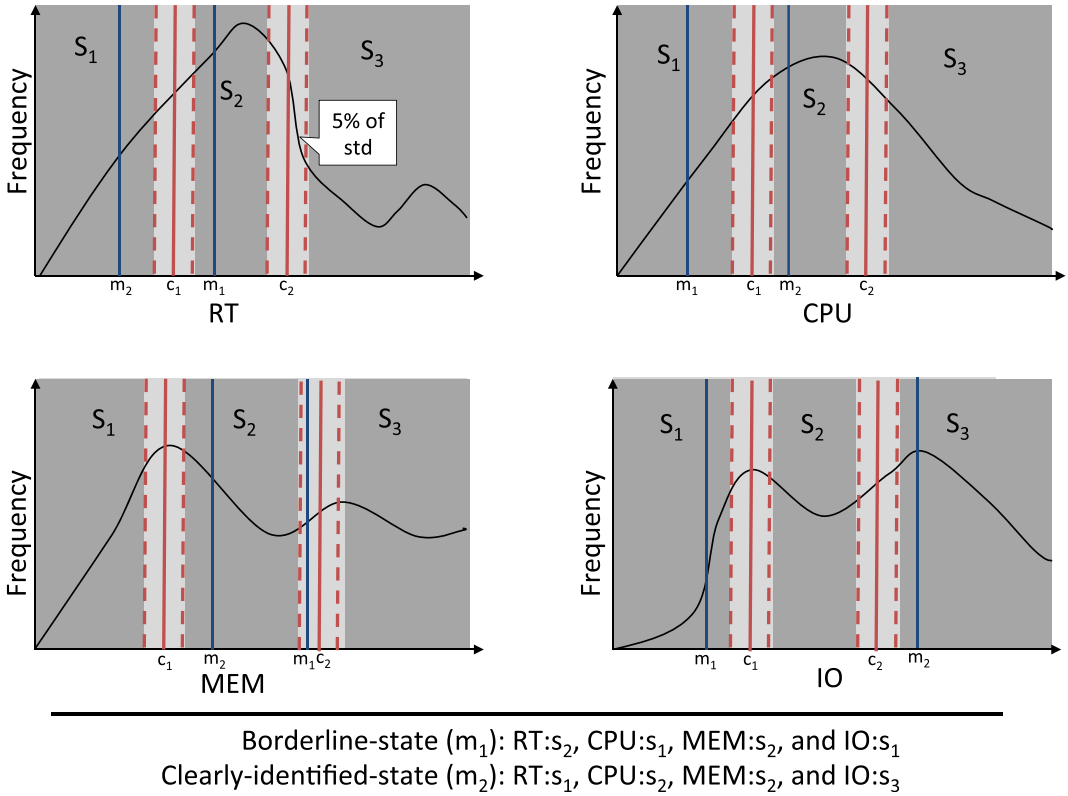

FIGURE 5 The difference between a *borderline state* and a *clearly identified state*. A metric value that is inside the light gray background area is considered as close to a state boundary. IO, disk I/O; MEM, memory usage; RT, response time

categorize the missed states into either borderline states or clearly identified states. Because the MEM value of m_1 is close to one of the state boundaries, we categorize m_1 as a borderline state. We categorize m_2 as a clearly identified state, because none of its metric values is close to a state boundary.

Furthermore, a borderline state is considered *relevant*, if the borderline state contains new information about the test. The example borderline state in Figure 5 can represent two system states, depending on the state boundaries:

- RT:S_2, CPU:S_1, MEM:S_3, and IO:S_1, or
- RT:S_2, CPU:S_1, MEM:S_2, and IO:S_1.

If neither of these two system states is covered by the test before the recommended stopping time, we consider the borderline state as a relevant borderline state, because the state contains new information that our approach misses to capture, regardless of the state boundary of the MEM metric.

We compare our two approaches using the *coverage-opportunity* metric. Coverage opportunity is measured as the number of new relevant states (clearly identified states or relevant borderline states) that can be covered by running the test longer. We calculate and compare the coverage-opportunity metrics of the three subject systems at the stopping times that are recommended by our two approaches. The coverage-opportunity metric is calculated as follows:

$$coverage-opportunity = \frac{missed_{clear} + missed_{relevant}}{required-execution-time}.$$

Where the *required-execution-time* is the duration of time (in hours) the test requires to cover the missed states, $missed_{clear}$ is the number of missed clearly identified states, and $missed_{relevant}$ is the number of missed relevant borderline states. If the coverage opportunity is small, it is not cost-effective to run the test longer.

Results. **Four out of eight states that are missed by our new approach in comparison with the 24-h load tests are relevant.** Table 6 shows that our approach misses one relevant state out of three missed states for CloudStore, two relevant states out of two missed states for PetClinic, and one relevant state out of three missed states for DS2.

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 6 The states that are missed by our approach and captured by the 24-h load tests

	Missed states	Borderline states	Relevant borderline	Clearly identified
CloudStore	3	2	1	0
PetClinic	2	0	0	2
DS2	3	2	1	0

TABLE 7 The states that are missed by our prior approach[4] and captured by the 24-h load tests

	Missed states	Borderline states	Relevant borderline	Clearly identified
CloudStore	9	7	2	1
PetClinic	20	0	0	20
DS2	3	2	1	0

TABLE 8 The *coverage opportunity* metrics of our approaches

	Coverage-opportunity metric	
	Prior approach	New approach
CloudStore	0.6	0.4
PetClinic	3.2	0.8
DS2	0.4	0.2

Twenty-four out of 32 states that are missed by our prior approach and captured by the 24-h tests are relevant. Table 7 shows that our prior approach misses three relevant states out of nine missed states for CloudStore, 20 relevant states out of 20 missed states for PetClinic, and one relevant state out three missed states for DS2. Hence, our prior approach not only misses more states than our new approach, the portion of missed states that are relevant is also higher.

The coverage opportunity metric values of our new approach are smaller than of our prior approach for all subject systems. Table 8 shows the coverage opportunity metrics for all three subject systems. Table 8shows that the stopping times that are recommended by our approach are more cost-effective than the stopping times that are recommended by our prior approach.

Our approach misses a small number of relevant states for all subject systems. The coverage-opportunity metric values of the approach that are presented in this section are smaller across all systems than the coverage-opportunity metric values of our prior approach, which indicates that our new approach recommends more cost-effective stopping times.

7 | CHALLENGES AND LESSONS LEARNED

In this paper, we propose an approach for deciding when to stop a load test, while taking combinations of performance metrics into account. We are partially successful in our approach. In our experiments on three open-source systems, our experimental results show that we can successfully recommend when to stop the load test. However, during our experiments, we ran into several challenges, which currently prevent us from applying our approach in a larger, industrial setting. In this section, we elaborate on these challenges. We call upon the community to address these challenges in future studies.

7.1 | Challenge C1: State explosion

The applicability of our approach is currently limited by the choice of n, the number of states in which we divide the metrics, and the number of performance metrics p. As the theoretical number of system states is n^p, the number of system states explodes for larger choices of n and p. As the differences between these states are often minor, the system will likely exhibit many different states during the test. Hence, detecting the stability of a load test is difficult for a large number of system states.

A possible solution to the state explosion problem is to prune the total number of states that a test can theoretically cover based on several heuristics. First, we should remove correlated performance metrics, as many performance metrics are highly correlated,[2] which should reduce the number of metrics. In addition, we should focus only on system states that are realistic and relevant for the application. For example, a memory-

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

intensive application may not exhibit all system states with a high CPU utilization when executing a representative workload. Hence, when detecting the stability of a load test for such an application, we should only look whether states that would occur during a realistic workload were already exhibited by the test. Such states can be captured from production systems. To provide a more generic approach, we are calling upon the community to provide such performance data from production systems.

7.2 | Challenge C2: Deciding on the state boundaries

We currently use the `bins()` function in `R` to transform the metric values into *n* states. This function has two disadvantages: (a) The algorithm used by the function divides the metric values into bins of equal size, and (b) the number of bins needs to be configured. We realized during our experiments that for many performance metrics, the values are not uniformly distributed. Therefore, based on the choice of *n*, states could have a very small range. This small range can be a problem when reaching the maximum capacity of a resource (e.g., 90%–95% vs. 95%–100% CPU utilization). We currently (partially) address this problem by introducing borderline states and clearly identified states in Section 6.2, but the interpretation and correctness of these 'fuzzy' state boundaries requires a large amount of domain knowledge. Future studies should investigate more automated approaches for deciding on the state boundaries, such as automated clustering approaches.

7.3 | Challenge C3: Determining the correctness of the recommended stopping point

Unlike functional testing, for which it is straightforward to determine whether a test passes or fails, load tests require more interpretation of their results. Hence, it is not a simple task to determine whether the reduced tests still fully represent the characteristics of the original load test. The most intuitive approach is to include domain experts in reviewing the reduced tests. The domain experts may indicate whether a reduced test is representative enough. If the reduced tests misses important information, pointing out the exact missed information can be of great help for us to improve the approach. However, first of all, such activities requires a large effort from practitioners, and more importantly, the reviewing itself is challenging. Even the experts may not have a clear vision of whether two load test results are representative of each other. Knowing the representativeness of a load test is still an open research challenge.

7.4 | Challenge C4: Reservation of practitioners

The last challenge of adopting our approach in industry is not technology related, but rather relates to social factors. On the one hand, practitioners often consider the long-running load tests as resource consuming and cost ineffective. On the other hand, when we propose an automated approach to recommend when to stop a load test to practitioners, they are often reluctant to use and depend on that approach in their work environment. The reason is that for many practitioners, the danger of missing critical performance information outweighs the burden of a long-running load test.

Unfortunately, the solution for this challenge lies in the solution for challenge C3, as we need a decisive way to determine whether our suggested stopping points are indeed correct. Future studies should investigate the important problem of whether a load test is representative of a system's real workload.

8 | THREATS TO VALIDITY

This section discusses the threats to the validity of our study.

8.1 | Threats to internal validity

Determining the system states. In our approach, every state is represented by a range of values for a metric. We determine a state by calculating cutting lines for each metric. These cutting lines might be too stringent. For example, if high CPU utilization is between 100% and 75%, and the test generates two consecutive values for the CPU metric of 74% and 76%, they are transformed into two different states, even though their difference is small. In future work, we will experiment with techniques for defining more fuzzy state boundaries.

8.2 | Threats to external validity

Our subject systems. To evaluate our proposed approach, we conducted experiments on three open-source online transaction processing systems (i.e., CloudStore, PetClinic, and DS2). All studied systems are used in prior work on performance engineering.[24-28] We selected systems that

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

use different programming languages: PHP for DS2 and Java for CloudStore and PetClinic. Future studies are necessary to evaluate our approach on large industrial systems and on systems from different domains.

Identifying metrics with upward/downward trends. In Section 4.2.1, we describe that we identify trends in metrics using the Spearman correlation. An internal threat to the validity of our approach is that we assume that the trend of a metric is constantly increasing or decreasing throughout the whole duration of the test. An alternative approach is to consider trends within a shorter time window. Future studies should investigate how one can best define the size of the time window during which it is decided whether a metric is trended.

Our load tests. In our experiment, we used two different types of loads: a random load for DS2 and a steady load for PetClinic and CloudStore. Also, we used two load drivers: the JMeter and DS2 load drivers. Future studies should evaluate our approach using other load test drivers and more complex (e.g., industrial) workloads.

8.3 | Threats to construct validity

The choice of our performance metrics. During a typical load test, hundreds of performance metrics are collected, most of which are highly correlated[2] and under a limited number of categories. When conducting our experimental study, we used only four metrics but from different categories. Our approach is not theoretically limited to a specific number of metrics, but as discussed in Section 7, there is a practical limit. Future studies should evaluate our approach with a larger set of metrics, to better understand what this practical limit is. In addition, it is essential to select the right metrics to capture the performance state of an application. The metrics we chose are examples of common choices, but these choices may depend strongly on various aspects of the application (e.g., the type, the environment, and the implementation language(s)). For example, for Java applications, it may be necessary to monitor both process and JVM memory usage. The ideal choice of performance metrics is one that should be made based on experience with the application and environment. We assume that if metrics are not monitored, they do not matter for testing the application under load.

Repetitiveness of a test. We assume that a load test contains a certain degree of repetitiveness. Future studies are necessary to investigate how our approach works for tests with changing workloads, for example, tests that contain bursty workloads.[36,37]

The 24-h execution time of the load test. In our experiments, we ran the load tests for the subject systems for 24 h. We chose to run the tests for 24 h because it is the maximum execution time we would be comfortable with for an industrial load test. The difficulty of deciding when to stop a load test is one of the main drivers for our research. In practice, the stopping times of load tests are chosen based on the experience of load testers. As a result, 'golden standard' load tests do not exist, which makes the evaluation of our approach challenging, a limitation of our approach that was also present in our prior work and related work (e.g., by He et al.[9] who used 1-week load tests instead of our 24-h load tests). Our approach aims at making this choice more systematic rather than based on gut feeling.

We acknowledge that the improvements that we report in terms of execution time depend on the length of the original load test. However, the exact improvements that our approach achieves are not very important, as they rely on the system under test and the workload as well. Table 4shows that the 24-h test is long enough for the CloudStore and the PetClinic systems, as all 81 theoretically possible states are exercised during the test. The load test for the DS2 system exercises only 66 out of the 81 theoretically possible states, which indicates that for some systems, it may be more difficult to exercise all system states. It is possible that allowing the test to run longer exercises more states. Future studies on more systems are necessary to investigate whether there exists an 'ideal' load test length that generalizes to more than one system.

9 | CONCLUSION

Conducting a proper load test before releasing a software system is critical, as such a test helps to ensure the performance of the system in the field. However, the time that is available to conduct a load test is usually limited. In our prior work,[4] we proposed an approach that recommends when to stop a load test based on repetitiveness in observed metric values. In this paper, we propose an improved approach that considers combinations of performance metrics as well.

Our approach measures the number of new system states, that is, the combinations of metric value states, that a test exercises. Once the test no longer exercises new system states in a given time frame, our approach recommends to stop the test. We evaluate our approach in experiments on three open-source systems (i.e., CloudStore, PetClinic, and Dell DVD Store). The most important results of our experiments are as follows:

1. Our proposed approach stops our load tests within 8.5 h, while covering at least 95% of the system states that are covered by the original 24-h load tests.
2. Our proposed approach recommends more cost-effective stopping times than our prior approach.[4]

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

While our new approach is an improvement over our prior approach, there exist several challenges, which currently prevent us from applying our approach in a larger, industrial setting. The most important challenges are the risk of state explosion and the difficulty of deciding on the representativeness of the shorter load test. We call upon the community to dedicate more research to this important industrial problem.

ORCID

Cor-Paul Bezemer https://orcid.org/0000-0002-0474-5718

REFERENCES

1. Williams A. Amazon Web Services outage caused by memory leak and failure in monitoring alarm. last visited: Sep 29 2018; 2012.
2. Malik H, Jiang ZM, Adams B, Hassan AE, Flora P, Hamann G. Automatic comparison of load tests to support the performance analysis of large enterprise systems. In: Proceedings of the European conference on software maintenance and reengineering IEEE; 2010:222-231.
3. Leitner P, Bezemer C-P. An exploratory study of the state of practice of performance testing in Java-based open source projects. In: Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE). ACM; 2017:373-384.
4. AlGhamdi HM, Syer MD, Shang W, Hassan AE. An automated approach for recommending when to stop performance tests. In: International Conference on Software Maintenance and Evolution IEEE; 2016:279-289.
5. Cohen I, Chase JS, Goldszmidt M, Kelly T, Symons J. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In: Proceedings of the Symposium on Operating Systems design And Implementation; 2004:231-244.
6. CodeWizards GmbH. Cloudstore. last visited: Sep 29 2018; 2017.
7. Spring Framework. Spring PetClinic. last visited: Sep 29 2018; 2007.
8. Dell. Dell DVD store. last visited: Sep 29 2018; 2010.
9. He S, Manns G, Saunders J, Wang W, Pollock L, Soffa M. L.. A statistics-based performance testing methodology for cloud applications. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). ACM; 2019; New York, NY, USA:188-199.
10. Jain R. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*: John Wiley and Sons; 1991.
11. Busany N, Maoz S. Behavioral log analysis with statistical guarantees. In: Proceedings of the international conference on software engineering ACM; 2016:877-887.
12. Jiang ZM, Avritzer A, Shihab E, Hassan AE, Flora P. An industrial case study on speeding up user acceptance testing by mining execution logs. In: Proceedings of the International Conference on Secure Software Integration and Reliability Improvement IEEE; 2010:131-140.
13. Tchana A, Dillenseger B, De Palma N, Etchevers X, Vincent J-M, Salmi N, Harbaoui A. Self-scalable benchmarking as a service with automatic saturation detection. *Middleware*: Springer Berlin Heidelberg; 2013:389-404.
14. Shivam P, Marupadi V, Chase J. S., Subramaniam T, Babu S. Cutting corners: workbench automation for server benchmarking. In: Usenix Annual Technical Conference; 2008:241-254.
15. Ayala-Rivera V, Kaczmarski M, Murphy J, Darisa A, Portillo-Dominguez A. Omar. One size does not fit all: in-test workload adaptation for performance testing of enterprise applications. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE). ACM; 2018: 211-222.
16. Apte V, Viswanath TVS, Gawali D, Kommireddy A, Gupta A. Autoperf: automated load testing and resource usage profiling of multi-tier internet applications. In: Proceedings of the 8th acm/spec on international conference on performance engineering (icpe). ACM; 2017:115-126.
17. Shariff S. M., Li H, Bezemer C, Hassan A. E., Nguyen T. H. D., Flora P. Improving the testing efficiency of selenium-based load tests. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST); 2019:14-20.
18. Grano G, Laaber C, Panichella A, Panichella S. Testing with fewer resources: an adaptive approach to performance-aware test case generation. *IEEE Trans Softw Eng (TSE)*. 2019:1-16.
19. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Softw Test Verif Reliab*. 2012;22(2):67-120.
20. Khatibsyarbini M, Isa MA, Jawawi DNA, Tumeng R. Test case prioritization approaches in regression testing: a systematic literature review. *Info Softw Technol*. 2018;93:74-93.
21. Gautheir TD. Detecting trends using Spearman's rank correlation coefficient. *Environ Forensics*. 2001;2(4):359-362.
22. Izrailev S. Binning algorithm. https://cran.r-project.org/web/packages/binr/binr.pdf. last visited: Sep 29 2018; 2016.
23. TPC. TPC. http://www.tpc.org/tpcw/. last visited: Sep 29 2018; 2005.
24. Chen T-H, Shang W, Jiang Z. M., Hassan A. E., Nasser M, Flora P. Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the International Conference on Software Engineering ACM; 2014:1001-1012.
25. Jiang Z. M., Hassan A. E., Hamann G, Flora P. Automated performance analysis of load tests. In: Proceedings of the International Conference on Software Maintenance IEEE; 2009:125-134.
26. Foo KC, Jiang ZM(J), Adams B, Hassan AE, Zou Y, Flora P. An industrial case study on the automated detection of performance regressions in heterogeneous environments. In: Proceedings of the International Conference on Software Engineering. IEEE; 2015:159-168.
27. Shang W, Hassan AE, Nasser M, Flora P. Automated detection of performance regressions using regression models on clustered performance counters. In: Proceedings of the international conference on performance engineering ACM; 2015:15-26.
28. Nguyen TH, Adams B, Jiang ZM, Hassan AE, Nasser M, Flora P. Automated detection of performance regressions using statistical process control techniques. In: Proceedings of the international conference on performance engineering ACM; 2012:299-310.
29. Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org. last visited: Sep 29 2018; 2017.
30. Oracle Corporation. MySQL. http://www.mysql.com. last visited: Sep 29 2018; 2017.
31. Binder R. *Testing object-oriented systems: models, patterns, and tools*: Addison-Wesley Professional; 2000.
32. Apache Software Foundation. Apache JMeter. http://jmeter.apache.org. last visited: Sep 29 2018; 2017.
33. Chen T-H, Shang W, Hassan AE, Nasser M, Flora P. Cacheoptimizer: helping developers configure caching frameworks for Hibernate-based database-centric web applications. In: Proceedings of the International Symposium on the Foundations of Software Engineering. ACM; 2016:666-677.

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

34. Microsoft. PerfMon. http://technet.microsoft.com/en-us/library/bb490957.aspx. last visited: Sep 29 2018; 2017.
35. Foo KC, Jiang ZM, Adams B, Hassan AE, Zou Y, Flora P. Mining performance regression testing repositories for automated performance analysis. In: Proceedings of the International Conference on Quality Software IEEE; 2010:32-41.
36. Mi N, Casale G, Cherkasova L, Smirni E. Burstiness in multi-tier applications: symptoms, causes, and new models. In: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware; 2008:265-286.
37. Casale G, Kalbasi A, Krishnamurthy D, Rolia J. Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware; 2009:20:1-20:20.

How to cite this article: AlGhamdi HM, Bezemer C-P, Shang W, Hassan AE, Flora P. Towards reducing the time needed for load testing. *J Softw Evol Proc*. 2023;35(3):e2276. https://doi.org/10.1002/smr.2276

20477481, 2023, 3, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/smr.2276 by Queen'S University, Wiley Online Library on [26/07/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License