Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

SPECIAL ISSUE: AUTOMATED SOFTWARE TESTING AND CONTINUOUS DEPLOYMENT

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, Canada
[2]Analytics of Software, Games and Repository Data (ASGAARD) lab, University of Alberta, Canada
[3]Department of Computer Science and Software Engineering, Concordia University, Canada
[4]BlackBerry, Canada

Correspondence
Cor-Paul Bezemer. Email: bezemer@ualberta.ca

Summary

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 tests often require a large amount of time, running from hours to even days, to execute. Nowadays, with the increased popularity of rapid releases and continuous deployment, testing time is at a premium and should be minimized while still delivering a complete test of the system. In our prior work, we proposed to reduce the execution time of a load test by detecting repetitiveness in individual performance metric values, such as CPU utilization or memory usage, 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 behaviour of a system.

Therefore, in this paper we revisit our prior approach, by proposing a new 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 conduct an experimental case study on three open source systems (CloudStore, PetClinic, and Dell DVD Store 2), 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. In addition, we show that our prior approach recommends a stopping time that is too early for two of the three studied systems. Finally, we discuss the challenges of applying our approach to an industrial setting, and we call upon the community to help us to address these challenges.

KEYWORDS:
load testing, performance testing, performance analysis

1 | INTRODUCTION

Performance issues in Ultra-Large-Scale (ULS) systems can cause impactful software failures. For example, due to a memory leak, Amazon Web Services experienced an outage that affected millions of users [36]. 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 [24]. 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 [23].

There exists a point in time during a load test at which the cost-effectiveness of the test is the highest, i.e., 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 [1], 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 [12]. Our prior approach does not capture such combinations, despite their importance for capturing the performance of a system.

In this paper, we revisit our prior approach [1] by focusing on combinations of performance metrics, using 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, i.e., 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 [11], PetClinic [33] and Dell DVD Store 2 [13]), in which we use our approach to reduce the execution time of a 24-hour 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-hour 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-hour test.

In summary, the main contributions of our paper are:

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 [1] 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 is 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., 1,000) of times. 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.

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

```
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 (1) reduce the execution time, (2) adapt the workload that drives a test or (3) 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 [1], the work that comes closest to our work is that of He et al. [18]. 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 [19] proposed an approach that recommends to stop a test when the variance in the response time is reduced to less than five percent. Jain's approach does not take into account the performance metrics about the physical level of a system, e.g., the CPU utilization. Other studies on reducing the execution time of tests depend on logs. Busany et al. [7] and Jiang et al. [21]

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 [12].

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. [34], Shivam et al. [32] and Ayala-Rivera et al. [5] adjust the load on a system until the system is saturated. Apte et al. [4] 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 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. [31] presented an approach to reduce the system resources required to conduct a browser-based load test itself, i.e., 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. [17] 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 [37] and Khatibsyarbini et al. [22].

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.

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 [1]. 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)	period p_1, low load
100	10 (S_1)	12 (S_1)	
101	10 (S_1)	10 (S_1)	
⋮	⋮	⋮	
149	49 (S_2)	100 (S_2)	period p_2, high load
150	50 (S_2)	100 (S_2)	
151	50 (S_2)	100 (S_2)	
⋮	⋮	⋮	
199	11 (S_1)	100 (S_2)	period p_3, overload
200	10 (S_1)	100 (S_2)	
201	10 (S_1)	100 (S_2)	

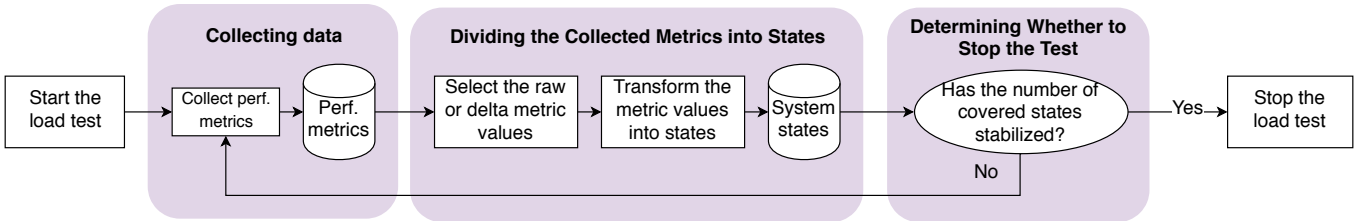

FIGURE 1 An overview of our approach

To make more efficient use of the time that is available for testing, Erica uses our prior approach [1] 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 [1], which searches for repetitiveness at the metric-level, p_3 would be considered repetitive and therefore redundant, as the observed response time 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, i.e., 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, e.g., 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 [1] 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.

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, i.e., 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.

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 response time 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 minutes 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.

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
```

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 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, i.e., 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 behaviour throughout the test. However, if the rate with which the metric increases or decreases is repetitive, the test exercises repetitive behaviour, and therefore we can stop the test.

To capture repetitiveness in the trend of a metric, we calculate the delta ($\triangle$) values. The $\triangle$ value of a metric is the difference between the current and the previous value of a metric. Figure 2 shows the raw and $\triangle$ values for the memory usage metric from the example in Table 2, together with a non-trended 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 [16]. We use the Spearman correlation because it is a nonparametric correlation measure, i.e., 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., $|\text{correlation value}| > 0.5$ and p-value < 0.05), we use the $\triangle$ metric values. Otherwise, we use the raw metric values. For our working example in Table 2, the memory metric has $|\text{correlation value}| = 1.00$ and p-value < 0.05. Therefore, we use the $\triangle$ values for the memory metric, whereas for the other metrics we use raw values.

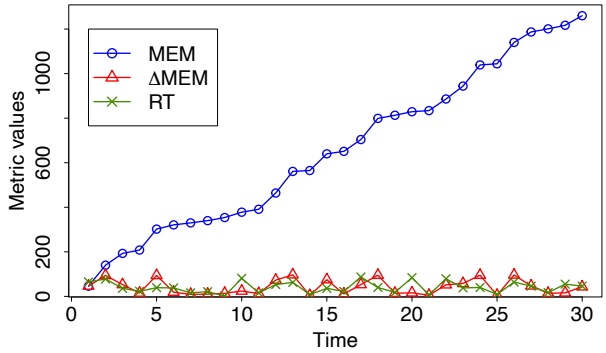

FIGURE 2 The MEM, △ MEM, and RT metrics from the example in Table 2

4.2.2 | Transforming the Metric Values into States

We use a binning algorithm (using the `bins()` function in `R` [29]) 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 3 states. The state boundaries of the states are the red vertical lines in the graphs.

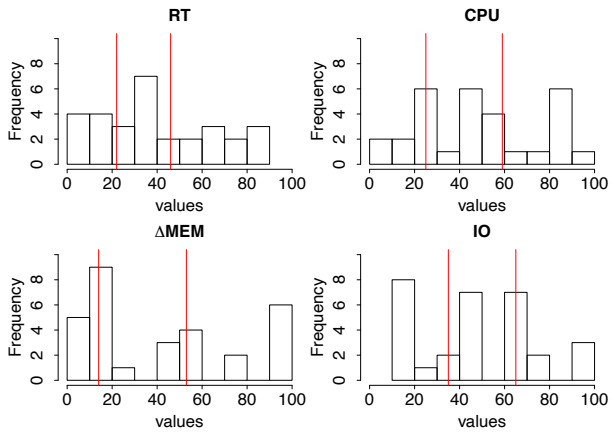

FIGURE 3 Transforming the values into states for the example from Table 2. The red vertical lines are the state boundaries determining the states

4.3 | Determining Whether to Stop the Test

We count the number of unique system states $\text{state}_{\text{unique}}$ that the test has exercised every τ minutes. Early in the test, $\text{state}_{\text{unique}}$ increases rapidly. However, as the test progresses, the exercised system states become repetitive, and $\text{state}_{\text{unique}}$ stabilizes. Our approach recommends to stop the test when $\text{state}_{\text{unique}}$ no longer increases. It is important to realize that the total number of 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 (e.g., for a large τ it will take longer to decide to stop a test than for a small τ).

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

TABLE 2 An example of performance metrics and metric states. 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)

t	Performance metric values					Metric state				System state
(mins)	RT	CPU	MEM	△ MEM	IO	RT	CPU	△ MEM	IO	exercised earlier?
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	✓

TABLE 3 The $state_{unique}$ for every 5 minutes 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

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 [1], to allow comparison of the results of our new and prior approaches. **CloudStore** [11] is an open-source e-commerce system, which follows the TPC-W performance benchmark standard [35]. CloudStore is built for performance benchmarking and contains 40.6k source lines of code (SLOC). **PetClinic** [33] 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 [9, 15, 20]. Dell DVD store (**DS2**, 3.3k SLOC) is an open-source web system [13] that benchmarks new hardware system installations by simulating an electronic commerce system. DS2 was also used in prior performance engineering research [27, 30].

5.2 | Deployment of the Subject Systems

Our experimental environment consists of 2 Intel CORE i7 servers that run Windows 8 with 16G 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, i.e., Tomcat 7.0.57 [3], and database, i.e., MySQL 5.6.21 [28], 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 [6]. The workloads of PetClinic and CloudStore are exercised using Apache JMeter [2]. We used a workload that was used in prior work on performance engineering for PetClinic [9] and for CloudStore [10]. The DS2 system has its own load generator, which generates a workload. The load tests of all three subject systems are run for 24 hours. We chose to run the tests for 24 hours, because although there exist load tests that run for longer, for practical reasons 24 hours is the maximum length we would like to assign to our industrial load tests. In addition, using 24 hours for our experiments allows us to compare the approach that is presented in this paper with our prior approach [1]. We used two different types of load intensity: random and steady. During the DS2 test, the load intensity is randomly changed every 10 minutes. 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 [26], 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 [14], the data that is 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 seconds. For example, PerfMon may record the CPU utilization at 12:01 (i.e., 12 minutes and 1 second), 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 seconds, we take the average of the metric values at 10, 20, and 30 seconds. 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 seconds 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 $d = 40$ minutes.

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.

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 [1] 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:

- 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 research question, 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, i.e., 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-hour 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 hours. 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 hours 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,

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

Subject systems	# of Covered states[1]	Stopping time[2]	Coverage[3]	Duration[4]
CloudStore	81	7:30	96%	31%
PetClinic	81	8:10	98%	34%
DS2	66	5:30	95%	23%

[1] the number of states that are covered by the 24-hour tests.
[2] reported as hours : minutes.
[3] percentage of all states that are exercised by the test at the stopping time.
[4] percentage of the full execution time, i.e., 24 hours.

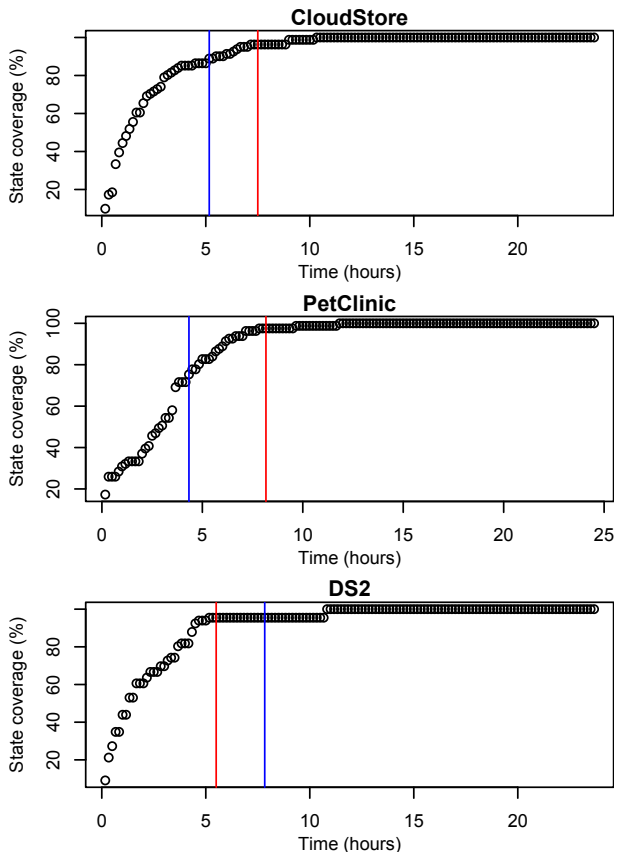

FIGURE 4 State coverage of our experiments over time. The red line is the stopping time that is recommended by our proposed approach. The blue line is the stopping time that is recommended by our prior approach [1]

TABLE 5 Comparison between the state coverage and the stopping times of new and prior [1] 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%

the increase rate of $\text{state}_{\text{unique}}$ 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 to 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, i.e., 95%, but with a delay of more than two hours.

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

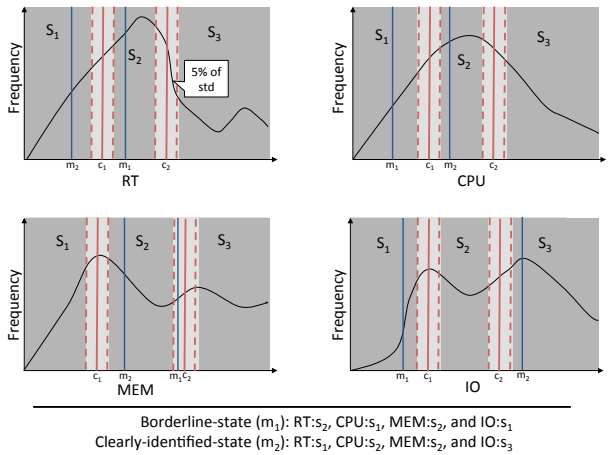

FIGURE 5 The difference between a *borderline-state* and a *clearly-identified-state*. A metric value that is inside the light grey background area is considered as close to a state boundary

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-hour load tests. In this research question, 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, i.e., m_1 and m_2 are indicated by the blue lines. The MEM value for m_1 is close to a state boundary, i.e., inside the light grey 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 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 are 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 are 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:

$$\textit{coverage-opportunity} = \frac{\textit{missed}_{clear} + \textit{missed}_{relevant}}{\textit{required-execution-time}}$$

TABLE 6 The states that are missed by our approach and captured by the 24-hour 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 [1] and captured by the 24-hour 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

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. **4 out of 8 states that are missed by our new approach in comparison to the 24-hour load tests are relevant.** Table 6 shows that our approach misses 1 relevant state out of 3 missed states for CloudStore, 2 relevant states out of 2 missed states for PetClinic, and 1 relevant state out of 3 missed states for DS2.

24 out of 32 states that are missed by our prior approach and captured by the 24-hour tests are relevant. Table 7 shows that our prior approach misses 3 relevant states out of 9 missed states for CloudStore, 20 relevant states out of 20 missed states for PetClinic, and 1 relevant state out 3 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 8 shows 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 is 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.

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 is 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 [24], 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-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.

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: (1) the algorithm used by the function divides the metric values into bins of equal size, and (2) 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.

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.

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 [9, 15, 20, 27, 30]. We selected systems that 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 [24], 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, e.g., tests that contain bursty workloads [8, 25].

The 24 Hours Execution Time of the Load Test. In our experiments we ran the load tests for the subject systems for 24 hours. We chose to run the tests for 24 hours 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. [18] who used one-week load tests instead of our 24 hours 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 4 shows that the 24-hour 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 [1], 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, i.e., 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:

1. Our proposed approach stops our load tests within 8.5 hours, while covering at least 95% of the system states that are covered by the original 24-hour load tests.

2. Our proposed approach recommends more cost-effective stopping times than our prior approach [1].

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.

References

[1] AlGhamdi HM, Syer MD, Shang W, Hassan AE (2016) An automated approach for recommending when to stop performance tests. In: International Conference on Software Maintenance and Evolution, IEEE, pp 279–289

[2] Apache Software Foundation (2017) Apache JMeter. http://jmeter.apache.org, last visited: Sep 29 2018

[3] Apache Software Foundation (2017) Apache Tomcat. http://tomcat.apache.org, last visited: Sep 29 2018

[4] Apte V, Viswanath T, Gawali D, Kommireddy A, Gupta A (2017) 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, pp 115–126

[5] Ayala-Rivera V, Kaczmarski M, Murphy J, Darisa A, Portillo-Dominguez AO (2018) 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, pp 211–222

[6] Binder R (2000) Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional

[7] Busany N, Maoz S (2016) Behavioral log analysis with statistical guarantees. In: Proceedings of the International Conference on Software Engineering, ACM, pp 877–887

[8] Casale G, Kalbasi A, Krishnamurthy D, Rolia J (2009) Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation. In: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware, pp 20:1–20:20

[9] Chen TH, Shang W, Jiang ZM, Hassan AE, Nasser M, Flora P (2014) Detecting performance anti-patterns for applications developed using object-relational mapping. In: Proceedings of the International Conference on Software Engineering, ACM, pp 1001–1012

[10] Chen TH, Shang W, Hassan AE, Nasser M, Flora P (2016) 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, pp 666–677

[11] CodeWizards GmbH (2017) Cloudstore. http://github.com/cloudstore/cloudstore, last visited: Sep 29 2018

[12] Cohen I, Chase JS, Goldszmidt M, Kelly T, Symons J (2004) 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, pp 231–244

[13] Dell (2010) Dell DVD store. http://linux.dell.com/dvdstore, last visited: Sep 29 2018

[14] Foo KC, Jiang ZM, Adams B, Hassan AE, Zou Y, Flora P (2010) Mining performance regression testing repositories for automated performance analysis. In: Proceedings of the International Conference on Quality Software, IEEE, pp 32–41

[15] Foo KC, Jiang ZMJ, Adams B, Hassan AE, Zou Y, Flora P (2015) An industrial case study on the automated detection of performance regressions in heterogeneous environments. In: Proceedings of the International Conference on Software Engineering, IEEE, pp 159–168

[16] Gautheir TD (2001) Detecting trends using Spearman's rank correlation coefficient. Environmental forensics 2(4):359–362

[17] Grano G, Laaber C, Panichella A, Panichella S (2019) Testing with fewer resources: An adaptive approach to performance-aware test case generation. IEEE Transactions on Software Engineering (TSE) pp 1–16

[18] He S, Manns G, Saunders J, Wang W, Pollock L, Soffa ML (2019) 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, New York, NY, USA, pp 188–199

[19] Jain R (1991) The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. John Wiley and Sons

[20] Jiang ZM, Hassan AE, Hamann G, Flora P (2009) Automated performance analysis of load tests. In: Proceedings of the International Conference on Software Maintenance, IEEE, pp 125–134

[21] Jiang ZM, Avritzer A, Shihab E, Hassan AE, Flora P (2010) 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, pp 131–140

[22] Khatibsyarbini M, Isa MA, Jawawi DN, Tumeng R (2018) Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology 93:74–93

[23] Leitner P, Bezemer CP (2017) 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, pp 373–384

[24] Malik H, Jiang ZM, Adams B, Hassan AE, Flora P, Hamann G (2010) 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, pp 222–231

[25] Mi N, Casale G, Cherkasova L, Smirni E (2008) Burstiness in multi-tier applications: Symptoms, causes, and new models. In: Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware, pp 265–286

[26] Microsoft (2017) PerfMon. http://technet.microsoft.com/en-us/library/bb490957.aspx, last visited: Sep 29 2018

[27] Nguyen TH, Adams B, Jiang ZM, Hassan AE, Nasser M, Flora P (2012) Automated detection of performance regressions using statistical process control techniques. In: Proceedings of the International Conference on Performance Engineering, ACM, pp 299–310

[28] Oracle Corporation (2017) MySQL. http://www.mysql.com, last visited: Sep 29 2018

[29] Sergei Izrailev (2016) Binning algorithm. https://cran.r-project.org/web/packages/binr/binr.pdf, last visited: Sep 29 2018

[30] Shang W, Hassan AE, Nasser M, Flora P (2015) Automated detection of performance regressions using regression models on clustered performance counters. In: Proceedings of the International Conference on Performance Engineering, ACM, pp 15–26

[31] Shariff SM, Li H, Bezemer C, Hassan AE, Nguyen THD, Flora P (2019) Improving the testing efficiency of selenium-based load tests. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), pp 14–20

[32] Shivam P, Marupadi V, Chase JS, Subramaniam T, Babu S (2008) Cutting corners: Workbench automation for server benchmarking. In: USENIX Annual Technical Conference, pp 241–254

[33] Spring Framework (2007) Spring PetClinic. https://github.com/spring-projects/spring-petclinic, last visited: Sep 29 2018

[34] Tchana A, Dillenseger B, De Palma N, Etchevers X, Vincent JM, Salmi N, Harbaoui A (2013) Self-scalable benchmarking as a service with automatic saturation detection. In: Middleware, Springer Berlin Heidelberg, pp 389–404

[35] TPC (2005) TPC. http://www.tpc.org/tpcw/, last visited: Sep 29 2018

[36] Williams A (2012) Amazon web services outage caused by memory leak and failure in monitoring alarm. https://techcrunch.com/2012/10/27/amazon-web-services-outage-caused-by-memory-leak-and-failure, last visited: Sep 29 2018

[37] Yoo S, Harman M (2012) Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 22(2):67–120

AUTHOR BIOGRAPHY

Hammam M. AlGhamdi is currently a Ph.D. student in the Department of Computer Science, University of Warwick, UK. His research interests include the analysis of multi-gigapixel digitized histopathological images using deep convolutional neural networks and other image processing techniques. His Master's degree, from Queen's University, Canada, was about finding the optimal execution time of performance tests using statistical learning and data mining techniques.

Cor-Paul Bezemer is an assistant professor in the Electrical and Computer Engineering department at the University of Alberta. He heads the Analytics of Software, GAmes And Repository Data (ASGAARD) lab. Before that, he was a postdoctoral research fellow in the Software Analysis and Intelligence Lab (SAIL) at Queen's University in Kingston, Canada. His research

interests cover a wide variety of software engineering and performance engineering-related topics. His work has been published at premier software engineering venues such as the TSE and EMSE journals and the ESEC-FSE, ICSME and ICPE conferences. He is one of the vice-chairs of the SPEC research group on DevOps Performance. Before moving to Canada, he studied at Delft University of Technology in the Netherlands, where he received his BSc (2007), MSc (2009) and PhD (2014) degree in Computer Science. For more information about Cor-Paul and the ASGAARD lab see: http://asgaard.ece.ualberta.ca/

Weiyi Shang is an Assistant Professor and Concordia University Research Chair in Ultra-large-scale Systems at the Department of Computer Science and Software Engineering at Concordia University, Montreal. He has received his Ph.D. and M.Sc. degrees from Queen's University (Canada) and he obtained B.Eng. from Harbin Institute of Technology. His research interests include big data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. His work has been published at premier venues such as ICSE, FSE, ASE, ICSME, MSR and WCRE, as well as in major journals such as TSE, EMSE, JSS, JSEP and SCP. His work has won premium awards, such as SIGSOFT Distinguished paper award at ICSE 2013 and best paper award at WCRE 2011. His industrial experience includes helping improve quality and performance of ultra-large-scale systems in BlackBerry. Early tools and techniques developed by him are already integrated into products used by millions of users worldwide. Contact him at shang@encs.concordia.ca; http://users.encs.concordia.ca/~shang

Ahmed E. Hassan is an IEEE Fellow, an ACM SIGSOFT Influential Educator, an NSERC Steacie Fellow, the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queen's University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves/d on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, and PeerJ Computer Science. More information at: http://sail.cs.queensu.ca/

Parminder Flora is currently the director of Performance and Test Automation for the BlackBerry Enterprise Server at BlackBerry. He founded the team in 2001 and has overseen its growth to over 40 people. He holds a BS degree in Computer Engineering from McMaster University and has been involved for over 10 years in performance engineering within the telecommunication field. His passion is to ensure that BlackBerry provides enterprise class software that exceeds customer expectations for performance and scalability.

How to cite this article: AlGhamdi H.M., Bezemer C., Shang W., Hassan A.E., and Flora P. (2019), Towards Reducing the Time Needed for Load Testing, *J. Softw. Evol. Proc.*, *2019;xx:1–...*