Continuous Validation of Load Test Suites

Mark D. Syer
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
mdsyer@cs.queensu.ca

Zhen Ming Jiang
Department of Electrical Engineering &
Computer Science
York University, Canada
zmjiang@cse.yorku.ca

Meiyappan Nagappan, Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
Queen's University, Canada
{mei, ahmed}@cs.queensu.ca

Mohamed Nasser, Parminder Flora
Performance Engineering
BlackBerry, Canada

ABSTRACT

Ultra-Large-Scale (ULS) systems face continuously evolving field workloads in terms of activated/disabled feature sets, varying usage patterns and changing deployment configurations. These evolving workloads often have a large impact on the performance of a ULS system. Hence, continuous load testing is critical to ensuring the error-free operation of such systems. A common challenge facing performance analysts is to validate if a load test closely resembles the current field workloads. Such validation may be performed by comparing execution logs from the load test and the field. However, the size and unstructured nature of execution logs makes such a comparison unfeasible without automated support. In this paper, we propose an automated approach to validate whether a load test resembles the field workload and, if not, determines how they differ by compare execution logs from a load test and the field. Performance analysts can then update their load test cases to eliminate such differences, hence creating more realistic load test cases. We perform three case studies on two large systems: one open-source system and one enterprise system. Our approach identifies differences between load tests and the field with a precision of $\geq$75% compared to only $\geq$16% for the state-of-the-practice.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Software Quality Assurance (SQA)*

1. INTRODUCTION

The rise of Ultra-Large-Scale (ULS) systems (e.g., Amazon.com, GMail and AT&T's telecommunication infrastructure) poses new challenges for the software performance field [26]. ULS systems require near-perfect up-time and support millions of concurrent connections and operations. Failures in such systems are typically associated with an inability to scale, than with feature bugs [15, 30, 47].

Load testing has become essential in ensuring the problem-free operation of such systems. Load tests are usually derived from the field (i.e., alpha or beta testing data or actual production data). The goal of such testing is to examine how the system behaves under realistic workloads to ensure that the system performs well in the field. However, ensuring that load tests are "realistic" (i.e., that they accurately reflect the current field workloads) is a major challenge. Field workloads are based on the behaviour of thousands or millions of users interacting with the system. These workloads continuously evolve as the user base changes, as features are activated or disabled and as user feature preferences change. Such varying field workloads often lead to load tests that are not reflective of the field [9, 46], yet these workloads have a major impact on the performance of the system [15, 49].

Performance analysts monitor the impact of field workloads on the system's performance using performance (e.g., response time and memory usage) and reliability counters (e.g., mean time-to-failure). Performance analysts must determine the cause of any deviation in the counter values from the specified or expected range (e.g., response time exceeds the maximum response time permitted by the service level agreements or memory usage exceeds the average historical memory usage). These deviations may be caused by changes to the field workloads [15,49]. Such changes are common and may require performance analysts to update their load test cases [9, 46]. This has led to the emergence of "continuous load testing," where load test cases are continuously updated and re-run even after the system's deployment.

A major challenge in the continuous load testing process is to ensure that load test cases accurately reflect the current field workloads. However, documentation describing the expected system behaviour is rarely up-to-date and instrumentation is not feasible due to high overhead [38]. Hence execution logs are the only data available to describe and monitor the behaviour of the system under a workload. Therefore, we propose an automated approach to validate load test cases by comparing system behaviour across load tests and the field. We derive system signatures from execution logs, then use statistical techniques to identify differences between the system signatures of the load tests and the field.

Such differences can be broadly classified as feature differences (i.e., differences in the available features), intensity differences (i.e., differences in how often each feature is exercised) and issue differences (i.e., new errors appearing in the field). These identified differences can help performance analysts improve their load tests in the following two ways.

ICPE'14, March 22–26, 2014, Dublin, Ireland.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2733-6/14/03 ...$15.00.
http://dx.doi.org/10.1145/2568088.2568101.

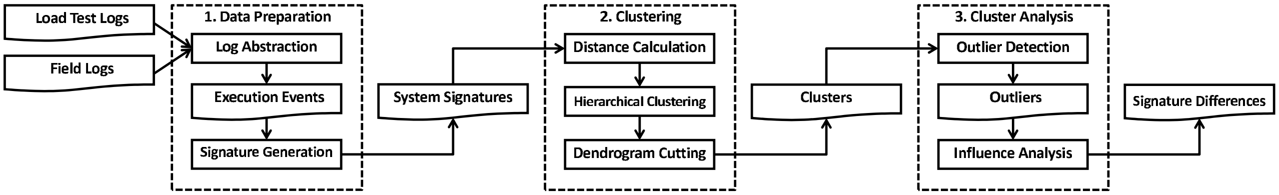

Figure 1: An Overview of Our Approach.

First, performance analysts can tune their load tests to more accurately represent current field workloads. For example, the test workloads can be updated to better reflect the identified differences. Second, new field errors, which are not covered in existing testing, can be identified based on the differences. For example, a machine failure in a distributed system may raise new errors that are often not tested.

This paper makes three contributions:

1. We develop an automated approach to validate the representativeness of load test cases by comparing the system behaviour between load tests and the field.

2. Our approach identifies important execution events that best explain the differences between the system's behaviour during a load test and in the field.

3. Through three case studies on two large systems, one open-source system and one enterprise ULS system, we show that our approach is scalable and can help performance analysts validate their load test cases.

1.1 Organization of the Paper

This paper is organized as follows: Section 2 provides a motivational example of how our approach may be used in practice. Section 3 describes our approach in detail. Section 4 presents our case studies. Section 5 discusses the sensitivity of our case study results to changes in the statistical measures used by our approach. Section 6 outlines the threats to validity and Section 7 presents related work. Finally, Section 8 concludes the paper and presents our future work.

2. MOTIVATIONAL EXAMPLE

Jack, a performance analyst, is responsible for continuously load testing a ULS system. Given the continuously evolving field workloads, Jack often needs to update his load test cases to ensure that the load test workloads match, as much as possible, the field workloads. Jack monitors the field workloads using performance counters (e.g., response time and memory usage). When one or more of these counters deviates from the specified or expected range (e.g., response time exceeds the maximum response time specified in the requirements or memory usage exceeds the average historical memory usage), Jack must investigate the cause of the deviation. He may then need to update his load test cases.

Although performance counters will indicate *if* the field workloads have changed, the only artifacts that Jack can use to understand *how* the field workloads have changed, and hence how his load test cases should be updated, are gigabytes of load test and field logs. Execution logs describe the system's behaviour, in terms of important execution events, during the load test and in the field.

Jack monitors the system's performance in the field and discovers that the system's memory usage exceeds the average historical memory usage. Pressured by time (given the continuously evolving nature of field workloads), management (who are keen to boast a high quality system) and the complexity of log analysis, Jack is introduced to an automated approach that can validate whether his load test cases are actually reflective of the field workloads and, if not, determine how his load test cases differ from the field. This approach automatically derives system signatures from gigabytes of execution logs and compares the signatures from the load test against the signatures in the field to identify execution events that differ between a load test and the field.

Using this approach, Jack is shown key execution events that explain the differences between his load test and field workloads. Jack then discovers a group of users who are using a memory-intensive feature more strenuously than in the past. Finally, Jack is able to update his load test cases to better reflect the users' changing feature preferences and hence the system's behaviour in the field.

3. APPROACH

This section outlines our approach for validating load test cases by automatically deriving system signatures from execution logs and comparing the signatures from a load test against the signatures from the field. Figure 1 provides an overview of our approach, and we describe each phase in detail below. We also demonstrate our approach with a working example of a hypothetical chat application.

3.1 Execution Logs

Execution logs record notable events at runtime and are used by developers (to debug a system) and operators (to monitor the operation of a system). They are generated by output statements that developers insert into the source code of the system. These output statements are triggered by specific events (e.g., starting, queueing or completing a job) and errors within the system. Compared with performance counters, which usually require explicit monitoring tools (e.g., PerfMon [5]) to be collected, execution logs are readily available in most ULS systems to support remote issue resolution and legal compliance (e.g., the Sarbanes-Oxley Act [6] requires logging in telecommunication and financial systems).

The second column of Table 1 and Table 2 presents the execution logs from our working example. These execution logs contain both static information (e.g,. `starts a chat`) and dynamic information (e.g., `Alice` and `Bob`) that changes with each occurrence of an event. Table 1 and Table 2 present the execution logs from the field and the load test respectively. The load test has been configured with a simple use case (from 00:01 to 00:06) which is repeatedly executed at a rate of one request per second.

Table 1: Abstracting Execution Logs to Execution Events: Execution Logs from the Field

Time	User	Log Line	Execution Event	Execution Event ID
00:01	Alice	starts a chat with Bob	starts a chat with ___	1
00:01	Alice	says ‘hi, are you busy?’ to Bob	says ___ to ___	2
00:17	Bob	says ‘yes’ to Alice	says ___ to ___	2
00:05	Charlie	starts a chat with Dan	starts a chat with ___	1
00:05	Charlie	says ‘do you have file?’ to Dan	says ___ to ___	2
00:08	Dan	Initiate file transfer to Charlie	Initiate file transfer (to ___)	3
00:12	Dan	says ‘got it?’ to Charlie	says ___ to ___	2
00:14	Charlie	says ‘thanks’ to Dan	says ___ to ___	2
00:14	Charlie	ends the chat with Dan	ends the chat with ___	4
00:18	Alice	says ‘ok, bye’ to Bob	says ___ to ___	2
00:18	Bob	says ‘bye’ to Alice	says ___ to ___	2
00:20	Alice	ends the chat with Bob	ends the chat with ___	4

Table 2: Abstracting Execution Logs to Execution Events: Execution Logs from a Load Test

Time	User	Log Line	Execution Event	Execution Event ID
00:01	USER1	starts a chat with USER2	starts a chat with ___	1
00:02	USER1	says ‘MSG1’ to USER2	says ___ to ___	2
00:03	USER2	says ‘MSG2’ to USER1	says ___ to ___	2
00:04	USER1	says ‘MSG3’ to USER2	says ___ to ___	2
00:05	USER2	says ‘MSG4’ to USER1	says ___ to ___	2
00:06	USER1	ends the chat with USER2	ends the chat with ___	5
00:07	USER3	starts a chat with USER4	starts a chat with ___	1
00:08	USER3	says ‘MSG1’ to USER4	says ___ to ___	2
00:09	USER4	says ‘MSG2’ to USER3	says ___ to ___	2
00:10	USER3	says ‘MSG3’ to USER4	says ___ to ___	2
00:11	USER4	says ‘MSG4’ to USER3	says ___ to ___	2
00:12	USER3	ends the chat with USER4	ends the chat with ___	5

3.2 Data Preparation

Execution logs are difficult to analyze because they are unstructured. Therefore, we abstract the execution logs to execution events to enable automated statistical analysis. We then generate system signatures that represent the behaviour of the system's users.

3.2.1 Log Abstraction

Execution logs are not typically designed for automated analysis. Each occurrence of an execution event results in a slightly different log line, because log lines contain static components as well as dynamic information (which may be different for each occurrence of a particular execution event). Therefore, we must remove this dynamic information from the log lines prior to our analysis in order to identify similar execution events. We refer to the process of identifying and removing dynamic information from a log line as "abstracting" the log line.

Our technique for abstracting log lines recognizes the static and dynamic components of each log line using a technique similar to token-based code clone detection techniques [28]. The dynamic components of each log line are then discarded and replaced with `___` (to indicate that dynamic information was present in the original log line). The remaining static components of the log lines (i.e., the abstracted log line) describe execution events.

Table 1 and Table 2 present the execution events and execution event IDs (a unique ID automatically assigned to each unique execution event) for the execution logs from the field and from the load test in our working example. These tables demonstrate the input (i.e., the log lines) and the output (i.e., the execution events) of the log abstraction process. For example, the `starts a chat with Bob` and `starts a chat with Dan` log lines are both abstracted to the `starts a chat with ___` execution event.

3.2.2 Signature Generation

We generate system signatures that characterize a user's behaviour in terms of feature usage expressed by the execution events. Therefore, a system signature represents the behaviour of one of the system's users. We use the term "user" to describe any type of end user, whether a human or software agent. For example, the end users of a system such as Amazon.com are both human and software agents (e.g., "shopping bots" that search multiple websites for the best prices). Signatures are generated for each user because workloads are driven by the combined behaviour of the system's users.

System signatures are generated in two steps. First, we identify all of the unique user IDs that appear in the execution logs. "User IDs" may include email addresses, device IDs or IP addresses that uniquely identify a human or software agent. The second column of Table 3 presents all of the unique user IDs identified from the execution logs of our working example. Second, we generate a signature for each user ID by counting the number of times that each type of execution event is attributable to each user ID. For example, from Table 1, we see that `Alice` starts one chat, sends two messages and ends one chat. Table 3 shows the signatures generated for each user using the events in Tables 1 and 2.

Table 3: System Signatures

		(Execution Event ID)			
	User ID	1	2	3	4
Field Users	Alice	1	2	0	1
	Bob	0	2	0	0
	Charlie	1	2	0	1
	Dan	0	1	1	0
Load Test Users	USER1	1	2	0	1
	USER2	0	2	0	0
	USER3	1	2	0	1
	USER4	0	2	0	0

3.3 Clustering

The second phase of our approach is to cluster the system signatures into groups of users where a similar set of events have occurred. The clustering phase in our approach consists of three steps. First, we calculate the dissimilarity (i.e., distance) between every pair of system signatures. Second, we use a hierarchical clustering procedure to cluster the system signatures into groups where a similar set of events have occurred. Third, we convert the hierarchical clustering into k partitional clusters (i.e., where each system signature is a member in only one cluster). We have automated the clustering phase using robust and scalable statistical techniques.

3.3.1 Distance Calculation

Each system signature is represented by one point in an n-dimensional space (where n is the number of unique execution events). Clustering procedures rely on identifying points that are "close" in this n-dimensional space. Therefore, we must specify how distance is measured in this space. A larger distance between two points implies a greater dissimilarity between the system signatures that these points represent. We calculate the distance between every pair of system signatures to produce a distance matrix.

We use the Pearson distance (a transform of the Pearson correlation [21]), as opposed to the many other distance measures [1,2,20,21], as the Pearson distance often produces a clustering that is closer to the true clustering (i.e., a closer match to the manually assigned clusters) [25, 40]. We find that the Pearson distance performs well when clustering system signatures (see Section 5).

We first calculate the Pearson correlation (ρ) between two system signatures using Equation 1. This measure ranges from -1 to +1, where a value of 1 indicates that two signatures are identical, a value of 0 indicates that there is no relationship between the signatures and a value of -1 indicates an inverse relationship between the signatures (i.e., as the occurrence of specific execution events increase in one system signature, they decrease in the other).

$$\rho = \frac{n\sum_i^n x_i \times y_i - \sum_i^n x_i \times \sum_i^n y_i}{\sqrt{(n\sum_i^n x^2 - (\sum_i^n x)^2) \times (n\sum_i^n y^2 - (\sum_i^n y)^2}} \qquad (1)$$

where x and y are two system signatures and n is the number of execution events.

We then transform the Pearson correlation (ρ) to the Pearson distance (d_ρ) using Equation 2.

$$d_\rho = \begin{cases} 1-\rho & \text{for } \rho \geq 0 \\ |\rho| & \text{for } \rho < 0 \end{cases} \qquad (2)$$

Table 5 presents the distance matrix produced by calculating the Pearson distance between every pair of system signatures in our working example.

3.3.2 Hierarchical Clustering

We use an agglomerative, hierarchical clustering procedure to cluster the system signatures using the distance matrix calculated in the previous step. The clustering procedure starts with each signature in its own cluster and proceeds to find and merge the closest pair of clusters (using the distance matrix), until only one cluster (containing everything) is left. Every time two clusters are merged, the distance matrix is updated. One advantage of hierarchical clustering is that we do not need to specify the number of clusters prior to performing the clustering. Further, performance analysts can change the number of clusters (e.g., to produce a larger number of more cohesive clusters) without having to rerun the clustering phase.

Hierarchical clustering updates the distance matrix based on a specified linkage criteria. We use the average linkage, as opposed to the many other linkage criteria [20,45], as the average linkage is the de facto standard [20,45]. The average linkage criteria is also the most appropriate when little information about the expected clustering (e.g., the relative size of the expected clusters) is available. We find that the average linkage criteria performs well when clustering system signatures (see Section 5).

When two clusters are merged, the average linkage criteria updates the distance matrix in two steps. First, the merged clusters are removed from the distance matrix. Second, a new cluster (containing the merged clusters) is added to the distance matrix by calculating the distance between the new cluster and all existing clusters. The distance between two clusters is the average distance (as calculated by the Pearson distance) between the system signatures of the first cluster and the system signatures of the second cluster [20, 45].

Figure 2 shows the dendrogram produced by hierarchically clustering the system signatures using the distance matrix (Table 5) from our working example.

Table 5: Distance Matrix

	Alice	Bob	Charlie	Dan	USER1	USER2	USER3	USER4
Alice	0	0.184	0	1.000	0	0.184	0	0.184
Bob	0.184	0	0.184	0.423	0.184	0	0.184	0
Charlie	0	0.184	0	1.000	0	0.184	0	0.184
Dan	1.000	0.423	1.000	0	1.000	0.423	1.000	0.423
USER1	0	0.184	0	1.000	0	0.184	0	0.184
USER2	0.184	0	0.184	0.423	0.184	0	0.184	0
USER3	0	0.184	0	1.000	0	0.184	0	0.184
USER4	0.184	0	0.184	0.423	0.184	0	0.184	0

3.3.3 Dendrogram Cutting

The result of a hierarchical clustering procedure is a hierarchy of clusters. This hierarchy is typically visualized using hierarchical cluster dendrograms. Figure 2 is an example of a hierarchical cluster dendrogram. Such dendrograms are binary tree-like diagrams that show each stage of the clustering procedure as nested clusters [45].

To complete the clustering procedure, the dendrogram must be cut at some height. This height represents the maximum amount of intra-cluster dissimilarity that will be accepted within a cluster before that cluster is further divided. Cutting the dendrogram results in a clustering where each system signature is assigned to only one cluster. Such a cutting of the dendrogram is done either by 1) manual (visual) inspection or 2) statistical tests (referred to as stopping rules).

Although a visual inspection of the dendrogram is flexible and fast, it is subject to human bias and may not be reliable. We use the Calinski-Harabasz stopping rule [11], as opposed to the many other stopping rules [11, 19, 36, 37, 39], as the Calinski-Harabasz stopping rule most often cuts the dendrogram into the correct number of clusters [36]. We find that the Calinski-Harabasz stopping rule performs well when cutting dendrograms produced by clustering system signatures (see Section 5).

The Calinski-Harabasz stopping rule is a pseudo-F-statistic, which is a ratio reflecting within-cluster similarity and between-cluster dissimilarity. The optimal clustering will have high within-cluster similarity (i.e., the system signatures within a cluster are similar) and a high between-cluster dissimilarity (i.e., the system signatures from two different clusters are dissimilar).

The dotted horizontal line in Figure 2 shows where the Calinski-Harabasz stopping rule cut the hierarchical cluster dendrogram from our working example into three clusters (i.e., the dotted horizontal line intersects with solid vertical lines at three points in the dendrogram). Cluster A contains one user (Dan), cluster B contains four users (Alice, Charlie, USER1 and USER3) and cluster C contains three users (Bob, USER2 and USER4).

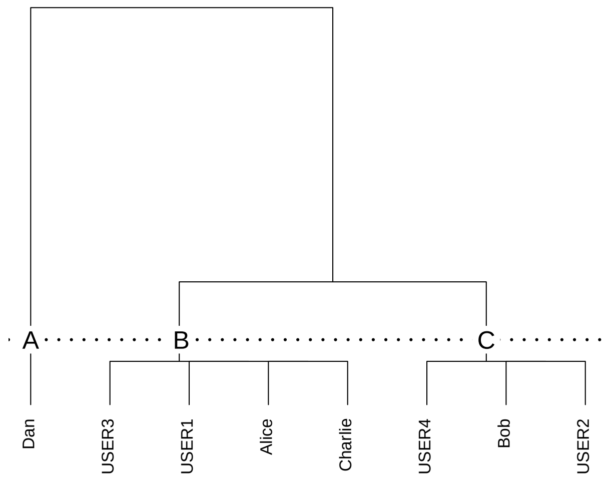

Figure 2: Sample Dendrogram.

3.4 Cluster Analysis

The third phase in our approach is to identify the execution events that correspond to the differences between the load test and field signatures. The cluster analysis phase of our approach consists of two steps. First, outlying clusters are detected. Second, the key execution events of the outlying clusters are identified. We refer to these execution events as "signature differences". Knowledge of these signature differences may lead performance analysts to update their load test cases. "Event A occurs 10% less often in the load test relative to the field" is an example of a signature difference that may lead performance analysts to update a load test case such that Event A occurs more frequently in the load test. We use robust and scalable statistical techniques to automate this step.

3.4.1 Outlier Detection

We identify outlying clusters using z-stats. Z-stats measure an anomaly's deviation from the majority (expected) behaviour [31]. Larger z-stats indicate an increased probability that the majority behaviour is the expected behaviour. Hence, as the z-stat of a particular cluster increases, the probability that the cluster is an outlying cluster also increases. Equation 3 presents how the z-stat of a particular cluster is calculated.

$$z(m,n) = \frac{\frac{m}{n} - p_o}{\frac{p_o \times (1-p_o)}{n}} \tag{3}$$

where m is the number of system signatures from the load test or the field (whichever is greater) in the cluster, n is the total number of system signatures in the cluster and p_o is the probability of the errors (by convention, p_o is typically assigned a value of 0.9 [31]).

Table 6 presents the size (i.e., the number of system signatures in the cluster), breakdown (i.e., the number of system signatures from the load test and the field) and z-stat for each cluster in our working example (i.e., each of the clusters that were identified when the Calinski-Harabasz stopping rule was used to cut the dendrogram in Figure 2).

Table 6: Identifying Outlying Clusters

		# Signatures from:		
Cluster	**Size**	**Field**	**Load Test**	**z-stat**
A	1	1	0	1.111
B	4	2	2	-17.778
C	3	1	2	-18.889

From Table 6, we identify Cluster A as an outlying cluster because its z-stat (1.111) is larger than the z-stats of Cluster B (-17.778) or Cluster C (-18.889).

3.4.2 *Signature Difference Detection*

We identify the differences between system signatures in outlying clusters and the average ("normal") system signature by performing an influence analysis on the signatures. This analysis quantifies the importance of each execution event in differentiating a cluster. Knowledge of these events may lead performance analysts to update their load test cases.

First, we calculate the centre of the outlying clusters and the universal centre. These centres represent the location, in an n-dimensional space (where n is the number of unique execution events), of each of the outlying clusters, as well as the average ("normal") system signature. The centre of a cluster is the average count for each unique event across either 1) all of the signatures in the cluster (for the centre of an outlying cluster) or 2) all of the signatures in all of the clusters (for the universal centre).

Table 7 presents the universal centre and cluster A centre.

Table 7: Universal Centre and Cluster A Centre

	System Signatures (Execution Event ID)			
	1	2	3	4
`Universal Centre`	0.5	1.875	0.125	0.5
`Cluster A Centre`	0	1	1	0

Second, we calculate the Pearson distance (d_ρ) between the centre of the outlying cluster and the universal centre. This "baseline" distance quantifies the difference between the system signatures in outlying clusters and the universal average system signature. The Pearson distance between the centre of Cluster A and the Universal Centre is 0.625.

Third, we calculate the change in the baseline distance between the outlying cluster's centre and the universal centre with and without each execution event. This quantifies the influence of each event. When an overly influential event is removed, the outlying cluster becomes more similar to the universal average system signature (i.e., closer to the universal centre) hence these events will have a negative Δd_ρ.

Table 8 presents the change in the distance between the centre of Cluster A and the Universal Centre when each event is removed from the distance calculation.

Table 8: Identifying Influential Execution Events

Event ID	Δd_ρ
1	0.0613
2	0.375
3	-0.625
4	0.0613
$\mu_{\Delta d_\rho}$	-0.0320
$\sigma_{\Delta d_\rho}$	0.422

Finally, we identify the influential events as any execution event that, when removed from the distance calculation, decreases the distance between the outlying cluster and the universal centre by more than twice the standard deviation less than the average. Decreasing the distance between two clusters indicates that they have become more similar. This analysis is similar to how dfbeta residuals are used to identify observations that have a disproportionate influence on the estimated coefficient values in a regression model [14,17].

From Table 8, the average change in distance ($\mu_{\Delta d_\rho}$) is -0.0320 and the standard deviation of the changes in distance ($\sigma_{\Delta d_\rho}$) is 0.422. Therefore, no execution events are identified as outliers because no change in distance is more than two standard deviations less than the average change in distance (i.e., no Δd_ρ value is $\leq \mu_{\Delta d_\rho} - 2 \times \sigma_{\Delta d_\rho} = -0.877$). For the purposes of this example, we use one standard deviation (as opposed to two standard deviations). Therefore, we identify execution event 3 (i.e., initiating a file transfer) as overly influential.

Our approach identifies one system signature (i.e., the system signature representing the user Dan) as a key difference between the load test and the field. In particular, we identify one execution event (i.e., initiating a file transfer) that is not well represented in the load test (in fact it does not occur at all). Performance analysts should then adjust the load intensity of the file transfer functionality in the load test.

In our simple working example, performance analysts could have examined how many times each execution event had occurred during the load test and in the field and identified events that occur much more frequently in the field compared to the load test. However, in practice, data sets are considerably larger. For example, our first enterprise case study contains over 1,400 different types of execution events and over 17 million log lines. Further, some execution events have a different impact on the system's behaviour based on the manner in which the event is executed. For example, our second enterprise case study identifies an execution event that only causes errors when over-stressed by an individual user (i.e., one user executing the event 1,000 times has a different impact on the system's behaviour than 100 users each executing the event 10 times). Therefore, in practice performance analysts cannot simply examine occurrence frequencies.

4. CASE STUDIES

This section outlines the setup and results of our case studies. First, we present a case study using a Hadoop application. We then discuss the results of two case studies using an enterprise system. Table 9 outlines the systems and data sets used in our case studies.

Our case studies aim to determine whether our approach can detect system signature differences due to 1) feature differences, 2) intensity differences and 3) issue differences between a load test and the field. Our case studies include systems whose users are either human (Enterprise System) or software (Hadoop) agents.

We compare our results with the current state-of-the-practice. Currently, performance analysts validate load test cases by comparing the number of times each execution event has occurred during the load test compared to the field and investigating any differences. Therefore, we rank the events based on the difference in occurrence frequency between the load test and the field. We then investigate the events with the largest differences. In practice, performance analysts do not know how many of these events should be investigated. Therefore, we examine the minimum number of events such that the state-of-the-practice approach identifies the same problems as our approach. We then compare the precision of our approach to the state-of-the-pratice.

Table 9: Case Study Subject Systems.

	Hadoop	Enterprise System	
Application domain	Data processing	Telecom	
License	Open-source	Enterprise	
Load Test Data			
# Log Lines	10,145	17,128,625	11,590,898
Notes	Load test driven by a standard Hadoop application.	Use-case load test driven by a load generator.	Load test driven by a replay script.
Field Data			
Execution Events	15,516	8,194,869	11,745,435
Notes	The system experienced a machine failure in the field.	System experts confirmed that there were no errors in the field.	The system experienced a crash in the field.
Type of Differences	Issue difference	Intensity and feature differences	Intensity difference
Our Approach			
Influential Events	4	17	4
Precision	75%	88%	100%
State-of-the-Practice Approach (Best Results)			
Influential Events	9	25	19
Precision	56%	60%	16%

4.1 Hadoop Case Study

4.1.1 The Hadoop Platform

Our first system is an application that is built on Hadoop. Hadoop is an open-source distributed data processing platform that implements MapReduce [3, 16].

MapReduce is a distributed data processing framework that allows large amounts of data to be processed in parallel by the nodes of a distributed cluster of machines [16]. The MapReduce framework consists of two steps: a Map step, where the input data is divided amongst the nodes of the cluster, and a Reduce step, where the results from each of the nodes is collected and combined.

Operationally, a Hadoop application may contain one or more MapReduce steps (each step is a "Job"). Jobs are further broken down into "tasks," where each task is either a Map task or a Reduce task. Finally, each task may be executed more than once to support fault tolerance within Hadoop (each execution is an "attempt").

4.1.2 The WordCount Application

The Hadoop application used in this case study is the WordCount application [4]. The WordCount application is a standard example of a Hadoop application that is used to demonstrate the Hadoop platform. The WordCount application reads one or more text files (a corpus) and counts the number of times each unique word occurs within the corpus.

Load test: We load test the Hadoop WordCount application on a cluster by attempting to count the number of times each unique word occurs in 3.69 gigabytes of text files. The cluster contains five machines, each with dual Intel Xeon E5540 (2.53GHz) quad-core CPUs, 12GB memory, a Gigabit network adaptor and SATA hard drives.

Field workload: We monitor the performance of the Hadoop WordCount application in the field. We find that the throughput (completed attempts/sec) is much lower than the throughput specified in the system's requirements. We also find that the average network IO (bytes/sec transfered between the nodes of the cluster) is considerably lower than the average historical network IO.

4.1.3 Applying Our Approach

We apply our approach to the execution logs collected from the WordCount application during the load test and from the field. We generate a system signature for each attempt because these attempts are the "users" of the Hadoop platform. Our approach identifies the following system signature differences (i.e., execution events that best describe the differences between the load test and the field):

```
INFO org.apache.hadoop.hdfs.DFSClient:  Abandoning
block blk_id
```

```
INFO org.apache.hadoop.hdfs.DFSClient:  Exception
in createBlockOutputStream java.io.IOException:
Bad connect ack with firstBadLink ip_address
```

```
WARN org.apache.hadoop.hdfs.DFSClient:  Error Recov-
ery for block blk_id bad datanode_id ip_address
```

```
INFO org.apache.hadoop.mapred.TaskTracker:
attempt_id progress
```

4.1.4 Results

Our approach flags only four execution events (out of 25,661 log lines that occur during the load test or in the field) for expert analysis. These execution events indicate that the WordCount application 1) cannot retrieve data from the Hadoop File System (HFS), 2) has a "bad" connection with the node at `ip_address` and 3) cannot reconnect to the datanode (data nodes store data in the HFS) at `ip_address`. Made aware of this issue, performance analysts could update their load tests to test how the system responds to machine failures and propose redundancy in the field.

The last execution event is a progress message. This execution event occurs less frequently than expected because some attempts in the field cannot retrieve data from the Hadoop File System (therefore these attempts make no progress). However, system experts do not believe that this is a meaningful difference between the system's behaviour during the load test and in the field. Hence, we have correctly identified 3 events out of the 4 flagged events. The precision of our approach (i.e., the percentage of correctly identified execution events) is 75%.

We also use the state-of-the-practice approach (outlined in Section 4) to identify the execution events with the largest occurrence frequency difference between the load test and the field. In order to produce the same results as our approach and identify the differences between the load test and the field, performance analysts must examine the top 6 events. Although examining the top 6 events will result in the same results as our approach, the precision is only 50% (compared to 75% for our approach).

4.2 Enterprise System Case Study

Although our Hadoop case study was promising, we perform two case studies on an enterprise system to examine the scalability of our approach. We note that these data sets are much larger than our Hadoop data set (see Table 9).

4.2.1 *The Enterprise System*

Our second system is a ULS enterprise software system in the telecommunications domain. For confidentiality reasons, we cannot disclose the specific details of the system's architecture, however the system is responsible for simultaneously processing millions of client requests and has very high performance requirements.

Performance analysts perform continuous load testing to ensure that the system continuously meets its performance requirements. Therefore, analysts must continuously ensure that the load test cases used during load testing accurately represent the current conditions in the field.

4.2.2 *Comparing Use-Case Load Tests to the Field*

Our first enterprise case study describes how our approach was used to validate a use-case load test (i.e., a load test driven by a load generator) by comparing the system behaviour during the load test and in the field. A load generator was configured to simulate the individual behaviour of thousands of users by concurrently sending requests to the system based on preset use-cases. The system had recently added several new clients. To ensure that the existing use-cases accurately represent the workloads driven by these new clients, we use our approach to compare a use-case load test to the field.

We use our approach to generate system signatures for each user within the use-case load test and in the field. We then compare the system signatures generated during the use-case load test to those generated in the field. Our approach identifies 17 executions events, that differ between the system signatures of the use-case load test and the field.

These results were then given to performance analysts and system experts who confirmed:

1. Nine events are under-stressed in the use-case load test relative to the field.

2. Six events are over-stressed in the use-case load test relative to the field.

3. Two events are artifacts of the load test (i.e., these events correspond to functionality used to setup the load test cases) and are not important differences between the load test and the field.

In summary, our approach correctly identifies 15 execution events (88% precision) that correspond to differences between the system's behaviour during the load test and in the field. Such results can be used to improve the use-case load tests in the future (i.e., by tuning the load generator to more accurately reflect the field conditions). In contrast, using the state-of-the-practice approach, performance analysts must examine the top 25 execution events in order to uncover the same 17 events that our approach has identified. However, the precision for the top 25 events is 60%, whereas the precision of our approach is 88%.

4.2.3 *Comparing Replay Load Tests to the Field*

Our second enterprise case study describes how our approach was used to validate a replay load test (i.e., a load test driven by a replay script) by comparing the system behaviour across a load test and the field.

Replay scripts record the behaviour of real users in the field then playback the recorded behaviour during a replay load test, where heavy instrumentation of the system is feasible. In theory, replay scripts can be used to perfectly replicate the conditions in the field during a replay load test [32]. However, replay scripts require complex software to concurrently simulate the millions of users and billions of requests captured in the field. Therefore, replay scripts do not scale well and use-case load tests that are driven by load generators are still the norm [35].

Performance analysts monitoring the system's behaviour in the field observed a spike in memory usage followed by a system crash. To understand the cause of this crash, and why it was not discovered during load testing, we use our approach to generate and compare system signatures for each user in the replay load test and the field. Our approach identifies 4 influential execution events that differ between the system signatures of the replay load test and the field.

These results were given to performance analysts who confirmed that four events are under-stressed in the replay load test relative to the field. Using this information, performance analysts update their load test cases. They then see the same behaviour during load testing as in the field.

In summary, our approach correctly identifies 4 influential execution events that correspond to differences between the system's behaviour during the load test and in the field. Such results provide performance analysts with a very concrete recommendation to help diagnose the cause of this crash.

We also compare our approach to the state-of-the-practice. In order to produce the same results as our approach and identify the differences between the load test and the field, performance analysts must examine the top 19 events. However, the precision of the state-of-the-practice approach is only 16% (compared to 100% for our approach).

> The state-of-the-practice approach has an average precision of 44%. However, performance analysts must examine an unknown number of events. Our approach flags events with an average precision of 88%.

5. SENSITIVITY ANALYSIS

The clustering phase of our approach relies on three statistical measures: 1) a distance measure (to determine the distance between each system signature), 2) a linkage criteria (to determine which clusters should be merged during the hierarchical clustering procedure) and 3) a stopping rule (to determine the number of clusters by cutting the hierarchical cluster dendrogram). To complement the existing literature, we verify that these measures perform well on our data and evaluate the sensitivity of our results to changes in these measures. We also determine the optimal distance measure, linkage criteria and stopping rule using our Hadoop case study data (similar results hold for our enterprise case study data).

5.1 The Optimal Distance Measure

The hierarchical clustering procedure begins with each system signature in its own cluster and proceeds to identify and merge clusters that are "close." The "closeness" of two clusters is measured by some distance measure. The optimal distance measure will result in a clustering that is closest to the true clustering.

We determine the optimal distance measure by comparing the results obtained by our approach (i.e., the execution events that we flag) when different distance measures are used. Table 10 presents how the number of flagged events, the precision (the percentage of correctly flagged events) and the recall (the percentage of true events that are flagged) is impacted by each of the distance measures in [2, 21]. We calculate recall using the best results in Table 10.

Table 10: Identifying the Optimal Distance Measure

Distance Measure	#Events	Precision	Recall
Pearson distance	4	75%	100%
Cosine distance	2	50%	33%
Euclidean distance	2	50%	33%
Jaccard distance	2	50%	33%
Kullback-Leibler Divergence	2	50%	33%

From Table 10, we find that the Pearson distance produces results with higher precision and recall than any other distance measure. In addition, all five distance measures identify the same two events (the Pearson distance correctly identifies two additional events).

5.2 The Optimal Linkage Criteria

The hierarchical clustering procedure takes a distance matrix and produces a dendrogram (i.e., a hierarchy of clusters). The abstraction from a distance matrix to a dendrogram results is some loss of information (i.e., the distance matrix contains the distance between each pair of system signatures, whereas the dendrogram presents the distance between each cluster). The optimal linkage criteria will enable the hierarchical clustering procedure to produce a dendrogram with minimal information loss.

We determine the optimal linkage criteria by using the cophenetic correlation. The cophenetic correlation measures how well a dendrogram preserves the information in the distance matrix [13]. The cophenetic correlation varies between 0 (the information in the distance matrix is completely lost) and 1 (the information is perfectly preserved). The optimal linkage criteria will have the highest cophenetic correlation. Table 11 presents the cophenetic correlation for a dendrogram built using each of the four main linkage criteria described in [20, 45].

Table 11: Identifying the Optimal Linkage Criteria

Linkage Criteria	Cophenetic Correlation
Average	0.782
Single	0.522
Ward	0.516
Complete	0.495

From Table 11, we find that the average linkage criteria produces a dendrogram that best represents the distance matrix. We also determine the optimal linkage criteria by applying the cluster analysis phase of our approach (see Subsection 3.4) on a dendrogram that has been built using each of these linkage criteria. Similar to our analysis of the optimal distance measure, Table 12 presents how the number of flagged events, the precision and the recall is impacted by using each of these linkage criteria.

Table 12: Identifying the Optimal Linkage Criteria

Distance Measure	#Events	Precision	Recall
Average	4	75%	100%
Single	6	50%	100%
Ward	2	50%	33%
Complete	3	33%	33%

From Table 12 we find that the average linkage criteria has the highest precision and recall. This is not surprising as the average linkage criteria also had the highest cophenetic correlation. Therefore, the cophenetic correlation may be used in the future to ensure that the average linkage criteria continues to perform well.

5.3 The Optimal Stopping Rule

To complete the clustering procedure, dendrograms must be cut at some height so that each system signature is assigned to only one cluster. Too few clusters will not allow outliers to emerge (i.e., they will remain nested in larger clusters) while too many clusters will lead to over-fitting and many false positives.

We determine the optimal stopping rule measure by applying the cluster analysis phase of our approach (see Subsection 3.4) on a dendrogram that has been cut using different stopping rules are used. Similar to our analysis of the optimal distance measure, Table 13 presents how the number of flagged events, the precision and the recall is impacted by the top 10 automated stopping rules in [36].

From Table 13, we find that the C-Index and Cubic Clustering Criterion stopping rules have 100% recall, but poor precision compared to the Calinski-Harabasz or Gamma stopping rules (75% precision and 75% recall). We select the Calinski-Harabasz stopping rule because the Gamma stopping rule is computationally intensive and does not scale well [12].

Table 13: Identifying the Optimal Stopping Rule

Distance Measure	#Events	Precision	Recall
Calinski-Harabasz	4	75%	75%
Duda and Hart	0	0%	0%
C-Index	7	57%	100%
Gamma	4	75%	75%
Beale	1	0%	0%
Cubic Clustering Criterion	7	57%	100%
Point-Biserial	1	0%	0%
G(+)	1	0%	0%
Davies and Bouldin	2	50%	25%
Stepsize	1	0%	0%

6. THREATS TO VALIDITY

6.1 Threats to Construct Validity

Evaluation

We have evaluated our approach by determining the precision with which our approach flags execution events that differ between the system signatures of a load test and the field. While performance analysts have verified these results, we do not have a gold standard data set. Further, complete system knowledge would be required to exhaustively enumerate every difference between a particular load test and the field. Therefore, we cannot calculate the recall of our approach. However, our approach is intended to help performance analysts identify differences between a load test and the field by flagging execution events for further analysis (i.e., to provide performance analysts with a starting point). Therefore, our goal is to maximize precision so that analysts have confidence in our approach. In our experience working with industry experts, performance analysts agree with this view [23, 24, 27, 43]. Additionally, we were able to identify at least one execution event that differed between the load test and the field in all of our case studies. Hence we were able to evaluate the precision of our approach in all three case studies.

6.2 Threats to Internal Validity

Execution Log Quality/Coverage

Our approach generates system signatures by characterizing a user's behaviour in terms of feature usage expressed by the execution events. However, it is possible that there are no execution logs to indicate when certain features are used. Therefore, our approach is incapable of identifying these features in the event that their usage differs between a load test and the field. However, this is true for all execution log based analysis, including manual analysis.

This threat may be mitigated by using automated instrumentation tools that would negate the need for developers to manually insert output statements into the source code. However, we leave this to future work as automated instrumentation imposes a heavy overhead on the system [34]. Further, Shang et al. report that execution logs are a rich source of information that are used by developers to convey important information about a system's behaviour [41]. Hence, automated instrumentation tools may not provide as deep an insight into the system's behaviour as execution logs.

Defining Users for Signature Generation

In our experience, ULS systems are typically driven by human agents. However, users may be difficult to define in ULS systems that are driven by software agents (e.g., web services [22]) or when users are allowed to have multiple IDs. Defining the users of a particular system is a task for the system experts. However, such a determination only needs to be made the first time our approach is used, afterwards this definition is reused.

6.3 Threats to External Validity

Generalizing Our Results

The studied software systems represent a small subset of the total number of Ultra-Large-Scale software systems. Therefore, it is unclear how our results will generalize to additional software systems, particularly systems from other domains (e.g., e-commerce). However, our approach does not assume any particular architectural details. Hence, there is no barrier to our approach being applied to other ULS systems. Further, we have evaluated our approach on two different systems: 1) an open-source distributed data processing system and 2) an enterprise telecommunications system that is widely used in practice.

Our approach may not perform well on small data sets (where we cannot generate many system signatures) or data sets where one set of execution logs (either the load test or field logs) is much larger than the other. However, the statistical measures that we have chosen are invariant to scale. Further, we have evaluated our approach on small data sets (10,145 load test log lines in our Hadoop study) and data sets where one set of execution logs is much larger than the other (the load test logs are twice as large as the field logs in our first Enterprise case study).

7. RELATED WORK

This paper presented an automated approach to validate load test cases by comparing execution logs from a load test and the field. Load test case design and log analysis are the most closely related areas of research to our work.

7.1 Load Test Case Design

Much of the work in load testing has focused on the automatic generation of load test cases [7, 8, 10, 18, 48]. A survey of load testing (and load test cases) may be found in [27]. Our approach may be used to validate load test cases by comparing the load tests performed with these test cases to the field. We intend to explore how the results of our approach may be used to automatically update load test cases.

7.2 Log Analysis

Shang et al. flag deviations in execution sequences mined from the execution logs of a test deployment and a field deployment of a ULS system [42]. Their approach reports deviations with a comparable precision to traditional keyword search approaches (23% precision), but reduces the number of false positives by 94%. Our approach does not rely on mining execution sequences. We also do not require any information regarding the timing of events within the system, which may be unreliable in distributed systems [33].

Jiang et al. flag performance issues in specific usage scenarios by comparing the distribution of response times for the scenario against a baseline derived from previous tests [30]. Their approach reports scenarios that have performance problems with few false positives (77% precision). To overcome the need for a baseline, Jiang et al. mine execution logs to determine the dominant (expected) behaviour of the system and flag anomalies from the dominant behaviour [29]. Their approach is able to flag <0.01% of the execution log lines for closer analysis by system experts. Our approach is interested in highlighting the differences between a load test and the field, as opposed to just anomalous behaviour. However, our approach can identify anomalous behaviour if such behaviour occurs primarily in the field (Subsection 4.2.3).

In our previous work, we proposed an approach to identify performance deviations in thread pools using performance counters [43, 44]. This approach is able to identify performance deviations (e.g., memory leaks) with high precision and recall. However, this approach did not make use of execution logs. Therefore, we could not identify the underlying cause of these performance deviations. The approach presented in this paper is concerned with highlighting the differences between a load test and the field, as opposed to just performance issues.

8. CONCLUSIONS AND FUTURE WORK

This paper presents an automated approach to validate load test cases by comparing system signatures from load tests and the field using execution logs. Our approach identifies differences between load tests and the field that performance analysts can use to update their load test cases to more accurately represent the field workloads.

We performed three case studies on two systems: one open-source system and one enterprise system. Our case studies explored how our approach can be used to identify feature differences, intensity differences and issue differences between load tests and the field. Performance analysts and system experts have confirmed that our approach provides valuable insights that help to validate their load tests and to support the continuous load testing process.

Although our approach performed well, we intend to explore how well our approach performs when comparing additional data sets as well as data sets from other ULS systems. We also intend to assess whether updating a load test case based on our approach results in the system's performance during the load test becoming more aligned with the system's performance in the field.

Acknowledgement

We would like to thank BlackBerry for providing access to the enterprise system used in our case study. The findings and opinions expressed in this paper are those of the authors and do not necessarily represent or reflect those of BlackBerry and/or its subsidiaries and affiliates. Moreover, our results do not reflect the quality of BlackBerry's products.

9. REFERENCES

[1] Cosine similarity, Pearson correlation, and OLS coefficients. http://brenocon.com/blog/2012/03/cosine-similarity-pearson-correlation-and-ols-coefficients.

[2] Gene Expression Data Analysis Suite: Distance measures. http://gedas.bizhat.com/dist.htm. Last Accessed: 17-May-2013.

[3] Hadoop. www.hadoop.apache.org/. Last Accessed: 17-Apr-2013.

[4] MapReduce Tutorial. http://hadoop.apache.org/docs/stable/mapred_tutorial.html. Last Accessed: 17-Apr-2013.

[5] PerfMon. http://technet.microsoft.com/en-us/library/bb490957.aspx. Last Accessed: 17-Apr-2013.

[6] The Sarbanes-Oxley Act 2002. http://soxlaw.com/. Last Accessed: 17-May-2013.

[7] A. Avritzer and E. J. Weyuker. Generating test suites for software load testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 44–57, Aug 1994.

[8] A. Avritzer and E. J. Weyuker. The automatic generation of load test suites and the assessment of the resulting software. *Transactions on Software Engineering*, 21(9):705–716, Sep 1995.

[9] L. Bertolotti and M. C. Calzarossa. Models of mail server workloads. *Performance Evaluation*, 46(2-3):65–76, Oct 2001.

[10] Y. Cai, J. Grundy, and J. Hosking. Synthesizing client load models for performance engineering via web crawling. In *Proceedings of the International Conference on Automated Software Engineering*, pages 353–362, Nov 2007.

[11] T. Calinski and J. Harabasz. A dendrite method for cluster analysis. *Communications in Statistics*, 3(1):1–27, Jan 1974.

[12] M. Charrad, N. Ghazzali, V. Boiteau, and A. Niknafs. *NbClust: An examination of indices for determining the number of clusters : NbClust Package*. Last Accessed: 13-Sep-2013.

[13] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni. Automated anomaly detection and performance modeling of enterprise applications. *Transactions on Computer Systems*, 27(3):6:1–6:32, Nov 2009.

[14] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. Routledge Academic, third edition, 2002.

[15] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, Feb 2013.

[16] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan 2008.

[17] T. V. der Meera, M. T. Grotenhuisb, and B. Pelzerb. Influential cases in multilevel modeling: A methodological comment. *American Sociological Review*, 75(1):173–178, 2010.

[18] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. Realistic load testing of web applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 57–68, Mar 2006.

[19] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons Inc, 1st edition, 1973.

[20] I. Frades and R. Matthiesen. Overview on techniques in cluster analysis. *Bioinformatics Methods In Clinical Research*, 593:81–107, Mar 2009.

[21] M. H. Fulekar. *Bioinformatics: Applications in Life and Environmental Sciences*. Springer, 1st edition, 2008.

[22] D. Greenwood, M. Lyell, A. Mallya, and H. Suguri. The IEEE FIPA approach to integrating software agents and web services. In *Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1412–1418, May 2007.

[23] A. E. Hassan and P. Flora. Performance engineering in industry: current practices and adoption challenges. In *Proceedings of the International Workshop on Software and Performance*, pages 209–209, Feb 2007.

[24] A. E. Hassan and R. C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3):335–367, Sep 2006.

[25] A. Huang. Similarity measures for text document clustering. In *Proceedings of the New Zealand Computer Science Research Student Conference*, pages 44–56, Apr 2008.

[26] S. E. Institute. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon University, 2006.

[27] Z. M. Jiang. *Automated Analysis of Load Testing Results*. PhD thesis, Queen's University, Jan 2013.

[28] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance and Evolution*, 20(4):249–267, Jul 2008.

[29] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automatic identification of load testing problems. In *Proceedings of the International Conference on Software Maintenance*, pages 307–316, Oct 2008.

[30] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated performance analysis of load tests. In *Proceedings of the International Conference on Software Maintenance*, pages 125–134, Sep 2009.

[31] T. Kremenek and D. Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the International Conference on Static Analysis*, pages 295–315, Jun 2003.

[32] D. Krishnamurthy, J. A. Rolia, and S. Majumdar. A synthetic workload generation technique for stress testing session-based systems. *Transactions on Software Engineering*, 32(11):868–882, Nov 2006.

[33] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, Jul 1978.

[34] H. Malik, Z. M. Jiang, B. Adams, A. E. Hassan, P. Flora, and G. Hamann. 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*, pages 222–231, Mar 2010.

[35] J. A. Meira, E. C. de Almeida, Y. L. Traon, and G. Sunye. Peer-to-peer load testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 642–647, Apr 2012.

[36] G. W. Milligan and M. C. Cooper. An examination of procedures for determining the number of clusters in a data set. *Psychometrika*, 50(2):159–179, Jun 1985.

[37] R. Mojena. Hierarchical grouping methods and stopping rules: An evaluation. *The Computer Journal*, 20(4):353–363, Nov 1977.

[38] D. L. Parnas. Software aging. In *Proceedings of the International Conference on Software Engineering*, pages 279–287, May 1994.

[39] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20(1):53–65, Nov 1987.

[40] N. Sandhya and A. Govardhan. Analysis of similarity measures with wordnet based text document clustering. In *Proceedings of the International Conference on Information Systems Design and Intelligent Applications*, pages 703–714, Jan 2012.

[41] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. W. Godfrey, M. Nasser, and P. Flora. An exploratory study of the evolution of communicated information about the execution of large software systems. In *Proceedings of the Working Conference on Reverse Engineering*, pages 335–344, Oct 2011.

[42] W. Shang, Z. M. Jiang, H. Hemmati, B. Adams, A. E. Hassan, and P. Martin. Assisting developers of big data analytics applications when deploying on hadoop clouds. In *Proceedings of the International Conference on Software Engineering*, pages 402–411, May 2013.

[43] M. D. Syer, B. Adams, and A. E. Hassan. Identifying performance deviations in thread pools. In *Proceedings of the International Conference on Software Maintenance*, pages 83–92, Sep 2011.

[44] M. D. Syer, B. Adams, and A. E. Hassan. Industrial case study on supporting the comprehension of system behaviour. In *Proceedings of the International Conference on Program Comprehension*, pages 215–216, Jun 2011.

[45] P.-N. Tan, M. Steinbach, and V. Kumar. *Cluster Analysis: Basic Concepts and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 2005.

[46] J. Voas. Will the real operational profile please stand up? *IEEE Software*, 17(2):87–89, Mar 2000.

[47] E. Weyuker and F. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Transactions on Software Engineering*, 26(12):1147–1156, Dec 2000.

[48] J. Zhang and S.-C. Cheung. Automated test case generation for the stress testing of multimedia systems. *Software: Practices and Experiences*, 32:1411–1435, Dec 2002.

[49] Z. Zhang, L. Cherkasova, and B. T. Loo. Benchmarking approach for designing a mapreduce performance model. In *Proceedings of the International Conference on Performance Engineering*, pages 253–258, Apr 2013.