Analytics-Driven Load Testing: An Industrial Experience Report on Load Testing of Large-Scale Systems

Tse-Hsun Chen*, Mark D. Syer*, Weiyi Shang†, Zhen Ming Jiang‡,
Ahmed E. Hassan*, Mohamed Nasser§, and Parminder Flora§
*Queen's University, †Concordia University, York University, §BlackBerry, Canada
*{tsehsun, mdsyer, ahmed}@cs.queensu.ca, †shang@encs.concordia.ca, ‡zmjiang@cse.yorku.ca

***Abstract*—Assessing how large-scale software systems behave under load is essential because many problems cannot be uncovered without executing tests of large volumes of concurrent requests. Load-related problems can directly affect the customer-perceived quality of systems and often cost companies millions of dollars. Load testing is the standard approach for assessing how a system behaves under load. However, designing, executing and analyzing a load test can be very difficult due to the scale of the test (e.g., simulating millions of users and analyzing terabytes of data). Over the past decade, we have tackled many load testing challenges in an industrial setting. In this paper, we document the challenges that we encountered and the lessons that we learned as we addressed these challenges. We provide general guidelines for conducting load tests using an analytics-driven approach. We also discuss open research challenges that require attention from the research community. We believe that our experience can be beneficial to practitioners and researchers who are interested in the area of load testing.**

***Index Terms*—load testing; test analysis; performance testing; mining software repositories;**

I. Introduction

Modern large-scale software systems such as e-commerce websites (e.g., eBay and Amazon.com) and telecommunications systems (e.g., Gmail, WhatsApp, and Skype) must support millions of concurrent user requests. Hence, the behaviour of these systems under load is one of the most important aspects of the systems' quality because load-related problems negatively impact the customers' experience. Load-related problems may be performance (e.g., memory leak) or functional problems (e.g., buffer overflow) that only occur when a system is under load. Further, load-related problems are often very costly. For example, increasing the time to load one of Amazon.com's web pages by merely one second may cost the company as much as 1.6 billion dollars in lost sales per year [15]. Therefore, it is essential to thoroughly load test such systems to uncover and fix problems before releasing the systems into production.

Load testing is the standard approach for assessing how large-scale software systems behave under load. The objective of load testing is to uncover load-related problems [21]. Performance analysts design load tests to ensure that the test loads are similar to the load seen in production (i.e., that the test loads are "realistic"). When conducting a load test, the systems are continuously monitored by collecting performance counters (e.g., CPU and memory), and the system's execution logs and event traces (e.g., "User *Peter* logged in"). However, conducting a load test and analyzing the performance counters and execution logs that are collected during testing can be extremely challenging and time-consuming [38].

Load testing produces and consumes a large amount of data. For example, load test cases need to be derived from historical usage data to ensure that the tests are realistic. However, deriving accurate test cases requires performance analysts to analyze terabytes or even petabytes of historical usage data [4]. Further, load tests usually last for hours (e.g., eight hours to simulate a working day) or even days (e.g., one work week) and the amount of collected performance counters and execution logs during testing can be overwhelming. Finally, performance analysts need to carefully review the collected data during testing to uncover abnormal behaviour and to determine if the test passes or fails.

We worked closely with our industrial partner on addressing the challenges that they face when designing, executing and analyzing load tests of large-scale software systems for over a decade. The authors of this paper have spent several years working at BlackBerry either as embedded researchers or as performance analysts. During this time, we faced many challenges that are associated with effectively and efficiently designing, executing and analyzing load tests.

In this paper, we summarize and consolidate our previously published work [3], [12], [13], [16], [22], [23], [25], [29], [30], [33], [36], [37], and provide general guidelines for conducting load testing using an *analytics-driven approach*. We discuss how we leverage data mining techniques to improve the efficiency and effectiveness of load testing. We also share the lessons that we learned when developing and deploying load testing tools in practice. Finally, we discuss open research challenges that require attention from the research community. Although we document our experience of conducting load testing and related research at BlackBerry, the experience is general and is applicable to other industrial settings. We believe that our experiences can help 1) practitioners in designing, executing and analyzing load tests; and 2) researchers who are interested in conducting load testing research.

The main contributions of this paper are:

- We provide an experience report that discusses the challenges that we encountered when designing, executing and analyzing load tests in an industrial setting.
- We also present the lessons that we learned while developing and deploying load testing tools in practice.
- We provide a general guideline on how to use an analytics-driven approach for designing, executing and analyzing load tests.
- We discuss several open research challenges which we hope will inspire future research efforts in the software engineering community.

Paper Organization. Section II provides a background on the different stages of load testing, and an overview of our analytics-driven approaches. Section III, IV, V, and VI discuss challenges that we encountered and open research challenges in test design, execution, and analysis. Section VII discusses related work. Finally, Section VIII concludes the paper.

II. Background and Overview

In this section, we introduce the three phases of load testing: test design, test execution, and test analysis, and describe the existing practices in the industry. We also briefly discuss the challenges that we faced in each of the three load testing phases. Our discussions focus on blackbox testing and are valid for a variety of large-scale enterprise systems, including cloud and on-prem deployments.

A. A Brief Introduction to Designing, Executing and Analyzing Load Tests

Load testing assesses systems' behaviour under load to detect load-related problems [21]. A workload is composed of a mix of requests that are sent at specific rates (e.g., 99 login requests per second or 1,337 tweets posted per second). Load testing can help uncover both functional (e.g., deadlocks) and non-functional problems (e.g., response time that violates service-level agreements). Figure 1 shows an overview of the different phases in load testing: test design, test execution, and test analysis. Below, we briefly discuss our experiences within each of these three load testing phases.

Test Design. The quality and effectiveness of a load test are highly dependent on the workload that is derived during the test design phase. Hence, performance analysts often attempt to design realistic workloads that model how users interact with the systems [24], [26]. However, in our experience, we find that designing workloads that are reflective of the field is a difficult because 1) the complexity of collecting and analyzing terabytes or even petabytes of field data; and 2) the constantly evolving field load [7]. Therefore, performance analysts must regularly analyze the field data and update the load test in response to the evolving field workloads. Hence, the current industry practice is to use an existing benchmark or to analyze the field data. However, benchmarks tend to capture the aggregated users' behavior (i.e., the overall request rates) and ignore detailed usage information (e.g., how an individual user interacts with the system). Further, once these workloads are derived they are rarely updated despite the constantly evolving nature of field workloads.

Test Execution. Load tests are executed in a testing environment ("lab") using dedicated hardware [27], [39]. The lab may consist of several servers and external services. The lab is configured to simulate a real-world environment (e.g., adding network latency [32]). In current industry practice, load tests are executed for a pre-specified time (e.g., several hours or even days) or until specific thresholds are met (e.g., number of received requests) [35]. Performance counters (e.g., CPU and memory) and system execution logs are recorded during tests. Both types of data provide insight into the system's behaviour under load. In our experience, the effectiveness of the load tests is not clear because the duration of the load tests may not be sufficient to uncover load-related problems. However, load tests need to be conducted on an almost daily basis to ensure the quality of each build of the system, and it not clear whether it is cost-effective to repeatedly execute the same scenarios over an extended period of time.

Test Analysis. Performance analysts analyze the performance counters and execution logs that are collected during a load test to determine whether any load-related problems were uncovered during testing. Performance counters are system performance data (e.g., CPU usage) that are collected at fixed time interval (e.g., every 30 seconds). Performance analysts may collect hundreds or thousands of performance counters [25]. Therefore, terabytes of performance counters may be collected during testing. Execution logs, which are generated by the systems, record events that occur during system execution (e.g., "User *Peter* logged in"). Terabytes or petabytes of logs may be collected during testing. As a result, performance analysts usually use heuristics to detect potential problems (e.g., searching for the keywords "Warning," "Error" or "Failure" in the execution logs) because too much data is collected for manual analysis. However, in our experience [23], we find that such heuristics are often insufficient to reliably determine whether a system "passes" a load test. Hence, the main challenge during test analysis is to effectively determine whether a system "passes" a load test by analyzing the large volume of collected log and counter data. In addition, performance analysts must also diagnose the underlying root cause of any problems.

B. Overview of the Challenges and Lessons Learned

In this section, we present some of the challenges that we encountered when designing, executing and analyzing load tests. We use analytics-driven approaches to improve the current practices of load testing. We believe that documenting these challenges and the lessons that we learned may help practitioners develop their own automated system performance assurance and analysis pipeline, and may inspire novel research opportunities in load testing.

Figure 1 shows the challenges associated with each phase of load testing. In the following four sections (Section III, IV, V, and VI), we discuss our experience in addressing

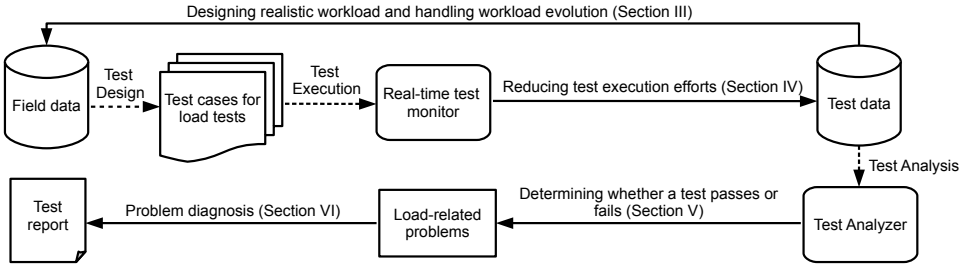

Fig. 1. An overview of the three phases in load testing (test design, test execution, and test analysis), and the challenges that we encountered.

these challenges. In each of these sections, we describe 1) the challenge, 2) our approach to address the challenge and 3) the lessons that we learned. Finally, we present some open research challenges.

III. Test Design Challenges: Designing Realistic Workload and Handling Workload Evolution

A. Challenge Description

Performance analysts need to ensure the performance and quality of the system when it is deployed in production and used by real users. Hence, load test workloads are usually derived from the field (i.e., alpha/beta testing data or actual field data) [6]. Workloads are based on the behaviour of thousands or millions of users interacting with the system. (We use the word "user" to represent any kind of endpoint that is accessing the system; for example, humans, mobile devices, or SQL queries.) Therefore, simulating the "realistic" usage of a system during testing should lead to load tests that are more representative of the field. However, field workloads continuously evolve as the user base changes, as features are added or removed, and as feature usage changes. Such evolving field workloads often lead to "outdated" tests that are not reflective of the field. Thus, it is important to have an automated approach that helps performance analysts with test design and validation by leveraging real-world field data.

B. Our Solutions to Test Design Challenges

Users' behaviour affects the systems in two ways. First, users' aggregated behaviour (e.g., number of requests per second) has an impact on the systems' performance (this is the "traditional" notion of a workload). Second, users' individual behaviour also has an impact on the systems' performance. We found cases when a feature only causes errors when over-stressed by an individual user (i.e., one user executing the feature 1,000 times has a different impact on the system's behaviour in contrast to 100 users each executing the feature 10 times) [36]. Therefore, we need to assess both the aggregate and user-level behaviour of the system's users.

We developed an approach for comparing the behaviour of users during a load test with the behaviour of users from a historical execution [36]. Our approach consists of the following three phases:

1) *Data preparation* – we abstract the execution logs to execution events using a technique similar to token-based clone detection [22], [23] (see Section V-B2 for more details). We then generate workload signatures that represent the behaviour of the systems' users (a "signature" is a mathematical construct that we us to quantify a workload). We generate two types of workload signatures. *An user-level workload signature* represents the collection of features that are used by an individual user. They are generated by 1) identifying all of the unique user IDs (e.g., names, email addresses, or device IDs) that appear in the execution logs; and 2) counting the number of times that each type of execution event is attributable to each user ID. *An aggregate user workload signature* represents the traditional notion of a "workload" (i.e., the total number and mix of features used in the systems). The information can be used to determine the aggregate workload when designing load tests. Workload signatures are generated by 1) grouping the execution logs into time intervals (i.e., grouping the execution logs that occur between two moments in time); and 2) counting the number of times that each type of execution event occurs within that time interval.

2) *Clustering* – we cluster the workload signatures into groups where a similar set of events have occurred. Clustering consists of three steps: 1) we calculate the dissimilarity (i.e., distance) between every pair of workload signatures; 2) we use a hierarchical clustering procedure to cluster the workload signatures into groups where a similar set of events have occurred; and 3) we convert the hierarchical clustering into k partitional clusters (i.e., where each workload signature is a member in only one cluster).

3) *Cluster analysis* – we analyze the clusters to identify the execution events that correspond to the differences between the workload signatures from the load test and the historical execution. Cluster analysis consists of two steps: 1) we detect outlying clusters (i.e., clusters that contain workload signatures that occur during the load test significantly more than in the historical execution) using *one-sample upper-tailed z-test for a population proportion* and 2) we identify key execution events that best explain the differences between the systems' workload during the load test in comparison to the systems' workload during the historical execution using *unpaired two-sample two-tailed Welch's unequal variances t-test* and *Cohen's d* [36].

We suggest comparing the systems' workload during testing to the systems' workload in the field to determine whether tests are comprehensive and realistic [36].

C. Lessons Learned

Load tests are often designed to meet throughput targets (e.g., 100 requests serviced per second). However, focusing on the aggregate user behaviour (i.e., the request rates) ignores the variability and the importance of the individual user behaviour (i.e., how a particular user interacts with the system).

Prior industry practice only focuses on analyzing request rates, rather than analyzing individual user behaviour. Our approach can uncover unique user behaviour and code paths that were not previously tested. We were also able to confirm the underlying cause of load-related problems and help highlight the importance of persona-based load testing (i.e., designing workloads using a combination of personas that model the behaviour of real-world users). A persona models the interaction of a particular type of real-world user with the system. For example, the personas for an e-commerce system could include "shopaholics" (i.e., users who make many purchases) and "window shoppers" (users who view many items without making any purchases). These personas correspond to different feature preferences (e.g., "shopaholics" will use payment features whereas "window shoppers" will make more use of search features). Therefore, persona-based load testing helps performance analysts ensure that their system performs well under a realistic workload. Our approach also helps performance analysts uncover previously unknown personas that have impacts on the system performance. Using our approach, we were able to show that load tests using workloads that were only designed to meet throughput targets are insufficient to confidently claim that the systems will perform well in production. Our approach is used regularly (e.g., every several months) to analyze the most recent user usage data from the field in order to verify whether the load tests are still up-to-date.

D. Open Research Challenges

How to Automatically Generate Test Data? Generating load test data that satisfies several system constraints (e.g., database schema or business logic) is extremely difficult. For example, performance analysts need to understand the database schema to avoid violating foreign key constraints when inserting test data into the database. In addition, the data insertion speed must be fast, because the test data many include hundreds of tables and millions of records. Hence, it may be necessary to skip certain data validation steps (e.g., user authentication on external servers). Approaches that can help performance analysts quickly generate a large amount of test data that satisfies all of the system's constraints may be very beneficial [18].

IV. Test Execution Challenges: Reducing Test Execution Efforts

A. Challenge Description

Load tests usually last several hours or even several days. In modern development practice, such as DevOps, load tests are executed on a daily basis to ensure the quality of the evolving systems [2], [8]. Hence, it is important to reduce the load test execution efforts while ensuring the tests are successfully executed and that the collected testing data is sufficient for assessing the system's behaviour. To address this challenge, we first discuss one of our approaches to determine whether a test is valid or not (e.g., environment issues such as disk failures) in Section IV-B1. We then discuss an approach that we use to determine when to stop a test in Section IV-B2.

B. Our Solutions to Test Execution Challenges

1) Determining the validity of a test: Performance analysts need to quickly determine that a test is invalid and should be stopped early. An invalid test happens when the system encounters some environmental or external failures that are not directly related to the system's quality. For example, if some unexpected problems, such as network failure or running out of disk space (e.g., forgetting to clean the old logs before running a new test), occur when conducting a load test, performance analysts may only find out that the test is invalid after the test is completed. In such cases, the time and computing resources are wasted, and the results of load testing are delayed.

We learned that to quickly determine that a test is invalid and should be stopped, we need real-time monitoring of the system under test. We leverage logs that are generated during load tests to monitor the system in real-time. In practice, logs are collected through central logging infrastructure (e.g., *syslog*) and are continuously sent to a remote logging server. Thus, we implement a framework to automatically monitor and analyze the logs on the logging server in real-time. If an unrecoverable issue occurs (e.g., disk failure), the framework will notify the performance analysts and they can decide whether the test should be stopped. Hence, performance analysts do not need to wait until a test is completed to determine whether the test results are invalid.

2) Finding test stopping criteria: There are no criteria to determine how long a load test should be executed. Although load tests often execute for a sufficiently long enough time to simulate real-world usage, much of the generated data is repetitive and non-informative. Therefore, if we can know that continuing the load test would not provide any new information about the system's performance, we can then simply stop the test. Reducing the test time can help minimize the costs of load testing resources. We measure repetitiveness in performance counters that are already generated from the load test, to determine whether to stop the test [3]. To quantify the repetitiveness of the generated performance data, we randomly pick a short time period from the data and exhaustively search for another time period that contains similar data. We leverage statistical analysis to determine whether the data in two time periods are similar or not. If we could find another time period that contains similar data, we consider the randomly picked time period as repeated. We conduct such process of picking a random time period and searching for a similar time period for a large number (e.g., 1,000) of times to determine the probability of having a time period that contains repeated data. We use this probability as a measure of repetitiveness in the

load test results. Performance analysts can use this measure to determine whether to stop the test based on their own experience. In addition, we developed an automated tool that suggests the stopping of a test if the repetitiveness stabilizes during a load test in real-time by continuously monitoring and analyzing the counters. We find that by using our approach, we can reduce the testing time by over 50%.

Modern large-scale systems usually rely on different external services [28]. For example, a system may depend on external services for authentication, notification, or billing. As the number of external services increases, the performance variations that we observe during a test may be due to instabilities from external services (e.g., when external services receive irregular heavy load). Performance analysts usually do not have access or control over such external services. The variation in the external services' performance may impact measuring the repetitiveness of the load test results. Therefore, it is important to eliminate uncontrolled performance variation from external services when conducting load tests. To reduce the uncontrolled variation during the test, we often implement mock services as a replacement for external services.

To reduce test execution, we continuously monitor if a test is valid (i.e., no unexpected error such as network failure). Then, we suggest stopping the load tests early if the results are highly repetitive.

C. Lessons Learned

The amount of resources that is required for load testing may be significant. However, our approaches have been used in practice to reduce the required resources to execute load tests on large-scale enterprise systems. Performance analysts use our approaches to reduce the costs of executing load tests in a continuous delivery pipeline (i.e., executing load tests to verify daily builds). We also find that deploying mock services reduces the required resources to execute load tests because mock services have the bare minimum functionalities of a real service. Performance analysts also have full control of the behaviour of the mock services when conducting load tests (e.g., adding network delays). External services may not be equipped with the necessary hardware for executing load tests without paying extra money. Performance analysts may even need to notify the operators of the external services and specifically schedule time for executing load tests. In some cases, we have even seen external services crashing under the load, which significantly delays the test execution plan. Prior to adopting our approaches, the number of executed load tests was much lower than the number of new system builds, which delays the development of the system. Increasing the number of executed load tests also increases the probability of uncovering load-related problems.

D. Open Research Challenges

How Can We Reduce the Costs of Running Load Tests? Executing load tests is expensive due to the long execution time and the required hardware. In some cases, performance analysts need to use the same production hardware settings (e.g., hundreds or thousands of servers) to ensure the performance of the system when it is deployed. Hence, it is difficult to run different types of tests (e.g., reproduce different performance problems that are reported by several customers) at the same time, which significantly slows down the entire load testing process. We believe that reducing the size of the load tests can further help improve the system performance (e.g., executing a smaller test on fewer machines will reach similar conclusions). However, there exists no proposed solution in the literature yet [21].

How to Reduce the Costs of Creating and Maintaining Realistic Mock Services? We find that mock services are essential for improving the repeatability of the identified load-related problems and for reducing dependencies on external service providers. One challenge that we find with creating mock services is that it is a manual-intensive process. Performance analysts need to first understand the protocols and format of the payloads that the external services use, and implement a mock service based on the result. In the case where the communication is encrypted between the systems and external services, performance analysts need to spend time decrypting the communicated messages. When the external services evolve, the mock services also need to be updated to include the new feature/behaviour. Hence, providing automated approaches for creating and maintaining mock services can significantly reduce the costs of load testing.

V. Test Analysis Challenge: Determining Whether a Test Passes or Fails

A. Challenge Description.

There are no clear criteria for determining whether a system "passes" or "fails" a load test. Even if a system complies with its service level agreement, there can still be performance problems that need to be uncovered and resolved (e.g., memory leaks). Due to the amount of data that is collected during load testing, it is important to provide automated tools that can help performance analysts uncover anomalies that happen during a test. We use anomaly detection based approaches to decide whether a test passes or fails [23], [33]. Such anomalies often indicate either performance or functional problems that require further attention from performance analysts. We first proposed approaches to detect performance anomalies (i.e., performance regressions) using performance counters [33]. Then, we proposed approaches to detect functional anomalies (e.g., potential functional problems) using logs [23].

B. Our Approaches for Detecting Anomalies

1) Automatically detecting performance anomalies: Performance regressions occur when problems are introduced after recent code changes (e.g., the new version is slower). However, since there can be hundreds of performance counters that are collected during a load test, it is difficult and time-consuming to manually identify the counters that suffer from performance regressions. We use a model-based performance regression

detection approach to automatically detect performance regressions [33]. Since there exists no clear passing and failing criteria for load tests, our approach provides an indicator of the test outcome (e.g., pass/fail) by mining historical tests.

We first group performance counters into clusters using a clustering algorithm (i.e., hierarchical clustering). Each data point in the cluster analysis can be viewed as a vector that represents the performance counters (e.g., CPU, memory, and disk I/O) sampled at a particular interval. For example, if a load test runs for an hour and performance counters are recorded every minute, there would be 60 data points for this load test, and each point is a vector that stores sampled values for CPU, memory, and disk I/O. After the cluster analysis, we can allocate the data points into several clusters (i.e., some points have similar characteristics). We then build a regression model using the data points in each cluster (e.g., if we have five clusters, we would have five regression models).

If new test data arrives, we first find the cluster to which it belongs. Then, we use the corresponding regression model that is trained using previous tests to determine whether the model has a low prediction error when applied on the new test data. If the predicted values are lower than the actual observed values (e.g., the system is slower than expected), then a performance regression might have happened, and a warning is sent to performance analysts for further verification. We also find that by clustering the counters together, performance analysts can quickly determine the type of performance regressions (e.g., whether it is CPU or memory related), which helps in the performance regression verification process.

2) Automatically detecting functional anomalies: It is challenging to verify the functional correctness in a fine-grained manner in a load test due to its size. Hence, performance analysts usually use keywords to search for errors in the log [23]. However, in many cases the problematic log lines may not contain such keywords, or the log lines are only indication of abnormal system behaviours (e.g., a log line that indicates the queue size is smaller than 0, which is likely caused by a buffer overflow, but no "error" keyword is logged). We develop an approach to automatically identify anomalies in log lines that may represent incorrect system behaviour. Our approach uncovers the dominant behavior (i.e., execution sequences) for the application and flags anomalies (i.e., deviations) from the dominant behavior. We identify execution sequences by first abstracting logs into events (i.e., remove parametric variables from log lines), then linking related events together using information such as thread IDs (i.e., the log lines are generated by the same thread, which usually corresponds to the same user activity). We then use a statistical approach called z-stat to identify anomalous event sequences [23]. The z-stat gives a score to the data distribution to indicate how far away the data is from the normal distribution. Thus, the larger the z-stat score is, the more likely an anomaly exists. For example, if *Event A* $\rightarrow$ *Event B* happens 10,000 times and *Event A* $\rightarrow$ *Event C* happens only 1 time, the z-stat score will be very large, and *Event A* $\rightarrow$ *Event C* will be identified as anomalous and will require attention from performance analysts.

TABLE I
AN EXAMPLE OF ABSTRACTING LOG LINES INTO EVENTS.

Before abstraction	**After Abstraction**
Add to cart, item=computer, price=800	Add to cart, item=$v, price=$v
Check out, total amount is 800	Checkout, total amount is $v
Add to cart, item=book, price=18	Add to cart, item=$v, price=$v
Check out, total amount is 18	Checkout, total amount is $v

In order to use an advanced analysis approach (e.g., data mining or machine learning algorithms) to analyze logs, we need to make the structure of log lines consistent. However, one challenge when implementing log abstraction is that logs follow a semi-structured format. Most systems use ad-hoc and non-standardized logging formats, which makes automated analysis very complex. For example, it may not always be easy to automatically identify which words correspond to parametric variables in a log line. Moreover, since the format of logs may change over time [34], a pre-defined log abstraction approach (e.g., using only regex to abstract the parametrize variables) may not always work and will require continuous maintenance. Table I shows an example of abstracted logs. One can see that, only after the abstraction may performance analysts apply different machine learning or data mining algorithms to analyze logs. Otherwise, machine learning algorithms cannot distinguish the difference between the first two log lines, and will treat them as separate events. Hence, we use an automated approach to process the log lines and abstract them into events (i.e., become structure data) [22].

Our approach tries to find log lines that have a very similar structure with slight differences due to parameterization (e.g., computer vs book in Table I). We first use regular expressions to abstract the parameters (e.g., *word=value*), then we tokenize the log lines using spaces. We group the log lines that have the same number of words and parameters together. For example, the first and third log line in Table I both have five words and two parameters, so they will be grouped together. Abstracted log lines in the same groups are merged to the same event. To improve the precision of our approach, we also merge the log lines in the same group if they differ from each other by one token at the same position. As an example, if we only use "=" to identify parameters in the log, we will not be able to abstract the second and the fourth log lines in Table I. However, since these two log lines have the same number of tokens and the token similarity differs by only one token (i.e., 800 and 18 in our example), we group these two log lines together as the same event. Note that, as mentioned in Section III-B, log abstraction is needed for comparing workloads in the test design step.

To help performance analysts detect anomalies, we provide two anomaly detection approaches. We leverage performance counters to detect performance anomalies (i.e., regressions); we leverage logs to detect functional anomalies.

C. Lessons Learned

Performance analysts have been using the presented approaches to detect anomalies for many years. Our approaches

are integrated into the continuous delivery process to verify the result of each load test. Our log abstraction and anomaly detection techniques are generic and are used to analyze logs from many systems within the organization with minimal changes. The approaches are well-received by the analysts. We were told that our approach of finding functional anomalies can also help uncover how customers are using the systems by highlighting the dominant behaviour. Our approach also helps improve the logging practices of developers. Developers are aware of the benefits of aforementioned approaches, so they try to unify the structure of the logs to ease analysis and avoid significant changes on logging formats. We find that log abstraction is one of the most important steps for in-depth log analysis. Log abstraction helps transform logs into events that can then be analyzed using advanced statistical or machine learning models. Nevertheless, based on our experience, many companies do not perform any log abstraction. Hence, standard logging infrastructure that can systematically separate the static and dynamic content of logs can help increase the adoption of log abstraction for more advanced analysis.

D. Open Research Challenges

How to Verify a Fix Without Re-executing the Entire Load Test Again? Conducting load tests can be very time and resource consuming. However, when the root cause of a problem is located and a fix is provided, performance analysts need to re-run the same test again to verify if the problem is resolved. Hence, approaches that can help performance analysts quickly verify a fix would be of great value. For example, perhaps the load test can be simplified to only trigger the modified code in order to reduce the execution time.

Is Having no Performance Regressions Good Enough? A major part of load test analysis is to identify performance regressions (e.g., performance anomalies in new builds). However, having no performance regression does not necessarily mean that the performance of the system is problem-free. In our experience, we have seen people increase the recommended requirement of the hardware when deploying their system since their system does not scale well. If we can successfully quantify the performance of a system along different aspects, we may help developers know where they should focus their efforts when improving the system performance. For example, developers may use bug detection approaches to uncover existing performance problems that are not related to performance regressions [11]–[13]. Also, approaches that automatically find optimal performance configurations can be very beneficial [10]. We believe that finding such problems can further help improve system performance and scalability. We encourage further research efforts on integrating other performance assurance approaches into the continuous delivery process to compliment load testing.

VI. Test Analysis Challenge: Problem Diagnosis

A. Challenge Description

In Section V, we discuss the approaches that we proposed and were adopted by our industrial partner to detect anomalies in load tests. After deploying the tools, we find that it is also important to help performance analysts further diagnose the root cause of the detected problems. Such problem diagnosis can significantly reduce the required manual effort for analyzing load test results. However, there are a few challenges that are associated with problem diagnosis, such as uncovering detail information in the workload, finding the reasons for performance deviation, and identifying the performance differences (e.g., anomalies). In the following subsection, we discuss our proposed solutions to the aforementioned challenges.

B. Solutions to Problem Diagnosis Challenges

1) Generating Important Metrics for Uncovering Load-related Problems: Identifying and generating important metrics that can be used for problem diagnosis is a major challenge in load testing. For example, if all we know is the time when a system is slow, it is challenging for us to uncover the root cause of a performance problem for that system. We need to gather additional information and from which we can then extract meaningful metrics (e.g., a particular event that is extremely slow when the entire system is slow). The more important metrics that we have, the higher the chance that we can uncover and locate load-related problems.

Software logs contain valuable information about the system execution. Such information is necessary for uncovering load-related problems [31], [41]. However, parsing and analyzing logs can be difficult due to the fact that logs are unstructured. Hence, based on our experience, we find that it is beneficial to provide performance analysts a general framework for extracting important metrics from logs. The framework requires some domain knowledge from the performance analysts to define some patterns in the logs. The pattern does not need to be related to only one printed line in logs (e.g., a log line that indicates a user is logged in), since many metrics that are extracted from logs require analyzing multiple log lines at the same time. These patterns usually reflect the business logic of the system, which helps performance analysts understand the root cause of the problem and ease the debugging process. After specifying the patterns, the framework extracts the metrics based on a pre-defined time interval (e.g, what is the maximum, mean, and minimum service subscription rate per each user at a ten-second interval), and the extracted metrics can be mapped to hardware counters such as CPU and memory usage. If a problem is located, performance analysts can quickly see how the extracted metrics correlate with the performance counters, and thus can uncover the root cause of the problem in a timely manner [25].

2) Quickly Identify Performance Deviation Using a User-Friendly Dashboard: Performance counters provide performance analysts with valuable insights into the performance of their systems during testing. However, hundreds or thousands of different performance counters are collected during testing [25]. Performance analysts are easily overwhelmed by the task of determining which performance counters indicate performance anomalies and when such anomalies occur during tests (tests may last for hours or days, yet an anomaly may only

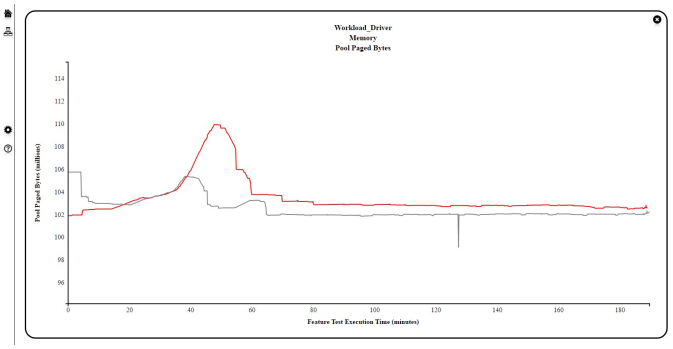

Fig. 3. When performance analysts click on a counter in Figure 2, they are shown the counter plotted over the duration of the test. This figure shows the Pool Paged Bytes (the portion of shared system memory that can be paged to the disk paging file [1]) on the Workload_Driver machine during the current load test (in red) and the baseline test in grey). Performance analysts can quickly see that the Pool Paged Byes on the Workload_Driver machine is much higher during the current load test than in the baseline test (especially between the 40th-60th minute of the load test).

last minutes or even seconds). Therefore, in addition to the performance anomaly detection approach that we mention in Section V-B1, we also implement a dashboard to help performance analysts quickly identify the most troubling anomalies. Figure 2 shows an example of our dashboard. Performance analysts can click on any counter in the dashboard to further investigate the difference between two tests (Figure 3). Tests can be automatically certified as "passing" if there are no deviations, "conditionally passing" if there are a few deviations or "failing" if there are many deviations.

We guide performance analysts to the point in time when the performance deviation (e.g., anomaly) occurred using our dashboard. Performance analysts can quickly determine the nature of the deviation (e.g., the CPU usage exceeds its historical values for a few seconds or memory usage exceeds its historical values for the entire test) and can use this knowledge to start a manual investigation. For example, performance analysts can identify the execution logs leading up to an anomaly for manual investigation. Our dashboard can also be supplemented with domain knowledge to combine our automatically generated statistical analysis with manually generated domain knowledge. For example. performance analysts can specify test phases (e.g., warm-up or cool-down) or periods of time when specific features are tested (e.g., feature A is tested for one hour, then feature B for one hour).

3) Quickly Comparing Results Across Tests: The system is always evolving, and the underlying software stack and configurations of the hardware may also change from time to time. There are two challenges associated with such evolution. First, as the performance data ages, the baseline test (tests for determining if a performance regression occurs) may change, so many of the discussed performance regression and anomaly detection approaches will not work using the old baseline test. Thus, it is important to keep track of the system versions and the detailed test environment for each load test. Second, it is important to compare the load testing result with prior tests to study how the system performance changes over time as the system evolves. By doing so, we can make sure that the system performance is not getting worse very slowly over time (i.e., performance regresses very slowly so it cannot be captured using statistical tests).

To overcome the above-mentioned challenges, we develop a load test repository. Similar to a version-control system that stores and maintains source code history, the load test repository stores and maintains all the detailed information of load tests that were conducted during the development history; such as the test environment, the version of the system under test, test description, summary of the test results, and the data that is generated during the test (e.g., performance counters and logs). One advantage of having a centralized load test repository is that the data can be easily shared among performance analysts. In addition, our load test repository allows performance analysts to compare the results of different load tests for problem diagnosis. For example, a performance analyst can compare two similar tests that are conducted in version 1 and version 35. Such comparison allows performance analyst to spot any possible performance regressions that might have been missed by the automated performance regression detection approaches. Our load test repository also allows performance analysts to search the conducted tests using different attributes, such as the version of the system under test and the version of the software library that is used. As a result, performance analysts can quickly compare the current test with tests that were conducted in an old environment, and manually examine any performance differences [16].

To help performance analysts with problem diagnosis, we implement a log analysis framework that can extract important metrics that reflect system operation. We implement a dashboard that allows performance analysts to quickly identify when and where anomalies happen. Finally, we develop a load test repository that allows performance analysts to quickly compare tests to identify problems.

C. Lessons Learned

Based on the feedback from the performance analysts, we find that automated tools are extremely important for analyzing load test results. Before the tools were developed, performance analysts spent much time parsing logs and analyzing counters in an ad-hoc fashion. After deploying the tools, we were told that performance analysts can significantly reduce the test analysis time. Moreover, performance analysts prefer tools that can directly flag problems in a test and prioritize the uncovered problems. We also find that it is important to leverage historical information, since the information preserves the knowledge of repeated problems and allows performance analysts to retrospectively improve their load testing process. The developed tools are used by performance analysts on a daily basis to analyze the results of load tests. Overall, we find that data visualization is key for the adoption and success of the tools, since data visualization allows performance analysts to quickly understand the root causes of any uncovered problems.

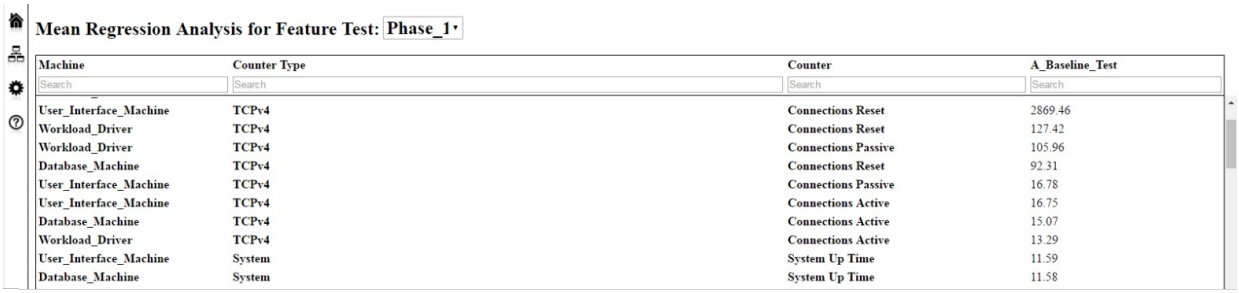

Fig. 2. Performance analysts are presented with a table of performance counters that is sorted by the magnitude of the difference in the counter values between the current load test and one or more baseline tests (i.e., A_Baseline_Test). For example, TCPv4 Connections Reset (i.e., the number of times that the TCP connections have made a direct transition to the CLOSED state from either the ESTABLISHED state or the CLOSE-WAIT state [1]) on the User_Interface_Machine machine (first row) shows the largest difference in the counter values between the current and baseline load test. Performance analysts can also quickly navigate the table using the search boxes (e.g., search for counters collected from only one machine).

D. Open Research Challenges

Can We Provide Auto-diagnosis for Previously Uncovered Problems? Our visualization approaches significantly reduce the required manual efforts on problem diagnosis. However, in our experience, we find that similar problems may occur again in the future as the systems evolve (e.g., developers make similar mistakes). Hence, automated approaches that can diagnose previously-uncovered problems (e.g., identifying recurring problematic patterns [13]) can further help performance analysts reduce problem diagnosis efforts.

VII. Related Work

In this section, we discuss prior work that is relevant to our paper. We categorize the related work into two categories: 1) experience reports on conducting load tests; and 2) using data analysis approaches for uncovering load-related problems.

Experience Reports on Conducting Load Tests. Weyuker *et al*. [40] discuss their experience on conducting load tests. They discuss the goal of load testing and the role of requirements when doing load testing. They also discuss how one can conduct load tests using the proposed approaches by walking the readers through an example. Avritzer *et al*. [5] propose a way to design system-independent workloads for evaluating the performance of the systems. The authors discuss how they use the approach to help a company. Garousi [17] propose a performance engineering process for stress testing real-time systems. The author applies the process on a case study system and documents the challenges that he encounters and discusses open research questions. Menasce *et al*. [26] describe the aspects where load testing helps improve system quality, and provide approaches for conducting load tests. Grechanik *et al*. [19] propose a feedback-directed approach that automatically selects appropriate inputs for performance tests to uncover problems. Grechanik *et al*. also discuss how their approach helps practitioners with problem diagnosis.

Many prior studies discuss load testing challenges that are related to this paper. However, these studies usually only report the experience on a particular phase of load testing and do not leverage analytics (i.e., data mining) approaches. On the other hand, we discuss our experience and the data analytics approaches that we use in assessing the various stages of load test (i.e., test design, execution, and analysis).

Using Data Analytics Approaches for Uncovering Load-Related Problems.

Similar to this paper, many prior studies use data analytics approaches when analyzing system data, since the amount of system-generated data can be tremendous. Prior studies apply statistical tests to detect performance anomalies and regressions [20], [25], [29], [30]. Some studies have used more advanced machine learning models such as Tree-Augmented Bayesian Networks [14] and linear regression [9], [33] to detect load-related problems. However, prior papers focus mostly on detecting problems in the load test results. In this paper, we focus on discussing our experience on using an analytics-driven approach to assist in not only test analysis but also test design and execution.

VIII. Conclusion

How a system behaves under load is one of the most important aspects of a system's quality because any load-related problem can negatively impact the users' experience and potentially cost a company billions of dollars each year. Load testing is the standard approach for assessing the behaviour of a system under load and for uncovering load-related problems. However, there are many challenges that are associated with load testing. Performance analysts need to execute tests that simulate millions of concurrent users and they need to analyze the terabytes of data that are collected during testing. We have been working closely with our industrial partner on conducting load testing research in an industrial setting. In this paper, we document and discuss the challenges that we encountered and the learned lessons over the past decade. We provide a general guideline on conducting load testing using an analytics-driven approach. We discuss how we leverage the data that is collected during load testing and from the field to address challenges in designing, executing and analyzing load tests. We also discuss open research challenges that require future research efforts. We believe that our experiences can help practitioners who wish to adopt load testing in their software development process, and can help researchers who are interested in conducting research in this area.

Acknowledgment

We are grateful to BlackBerry for providing access to many of the enterprise systems that we used in our case studies over

the years. The finding 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 affiliation. Our results do not in any way reflect the quality of BlackBerry's products.

REFERENCES

[1] "Common counters," https://technet.microsoft.com/en-us/library/ff367896(v=exchg.141).aspx.

[2] "Software performance engineering in the devops world," http://www.dagstuhl.de/de/programm/kalender/evhp/?semnr=16394, last accessed Oct 25 2016.

[3] H. M. Alghamdi, M. D. Syer, W. Shang, and A. E. Hassan, "An automated approach for recommending when to stop performance tests," in *Proceedings of the 32nd International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[4] A. Avritzer, J. P. Ros, and E. J. Weyuker, "Reliability testing of rule-based systems," *IEEE Softw.*, vol. 13, no. 5, pp. 76–82, Sep. 1996.

[5] A. Avritzer and E. J. Weyuker, "Deriving workloads for performance testing," *Softw. Pract. Exper.*, vol. 26, no. 6, pp. 613–633, Jun. 1996.

[6] S. Barber, "Creating effective load models for performance testing with incomplete empirical data," in *Proceedings of the Web Site Evolution, Sixth IEEE International Workshop (WSE)*, 2004, pp. 51–59.

[7] M. D. Barros, J. Shiau, C. Shang, K. Gidewall, H. Shi, and J. Forsmann, "Web services wind tunnel: On performance testing large-scale stateful web services," in *Proceedings of the 37th International Conference on Dependable Systems and Networks (DSN)*, 2007, pp. 612–617.

[8] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison Wesley, 2015.

[9] P. Bodík, M. Goldszmidt, and A. Fox, "Hilighter: Automatically building robust signatures of performance behavior for small- and large-scale systems," in *Proceedings of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2008, pp. 3–3.

[10] T.-H. Chen, S. Weiyi, A. E. Hassan, M. Nasser, and P. Flora, "CacheOptimizer: Helping developers configure caching frameworks for Hibernate-based database-centric web applications," in *Proceedings of the 24th International Symposium on the Foundations of Software Engineering (FSE)*, 2016, pp. 666–677.

[11] ——, "Detecting problems in the database access code of large scale systems - an industrial experience report," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 71–80.

[12] T.-H. Chen, S. Weiyi, h. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Trans. on Soft. Eng.*, 2016.

[13] T.-H. Chen, S. Weiyi, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Detecting performance anti-patterns for applications developed using object-relational mapping," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, 2014, pp. 1001–1012.

[14] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. S. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation (OSDI)*, 2004, pp. 3–16.

[15] F. Company, "How one second could cost Amazon 16 billion sales," http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales, last accessed March 3 2016.

[16] K. C. Foo, Z. M. J. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "An industrial case study on the automated detection of performance regressions in heterogeneous environments," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, 2015, pp. 159–168.

[17] V. Garousi, "Experience and challenges with uml-driven performance engineering of a distributed real-time system," *Inf. Softw. Technol.*, vol. 52, no. 6, pp. 625–640, Jun. 2010.

[18] M. Grechanik and G. Devanla, "Mutation integration testing," in *International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 353–364.

[19] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 156–166.

[20] C. Heger, J. Happe, and R. Farahbod, "Automated root cause isolation of performance regressions during software development," in *Proceedings of the 4th International Conference on Performance Engineering (ICPE)*, 2013, pp. 27–38.

[21] Z. M. Jiang and A. E. Hassan, "A survey on load testing of large-scale software systems," *IEEE Transactions on Software Engineering*, vol. 41, no. 11, pp. 1091–1118, 2015.

[22] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "An automated approach for abstracting execution logs to execution events," *J. Softw. Maint. Evol.*, vol. 20, no. 4, pp. 249–267, 2008.

[23] ——, "Automatic identification of load testing problems," in *Proceedings of 24th International Conference on Software Maintenance (ICSM)*, 2008, pp. 307–316.

[24] D. Krishnamurthy, J. A. Rolia, and S. Majumdar, "A synthetic workload generation technique for stress testing session-based systems," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 868–882, Nov. 2006.

[25] H. Malik, B. Adams, and A. E. Hassan, "Pinpointing the subsystems responsible for the performance deviations in a load test," in *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE)*, 2010, pp. 201–210.

[26] D. A. Menascé, "Load testing of web sites," *IEEE Internet Computing*, vol. 6, no. 4, pp. 70–74, Jul. 2002.

[27] ——, "Workload characterization," *IEEE Internet Computing*, vol. 7, no. 5, pp. 89–92, 2003.

[28] S. Newman, *Building Microservices*. O'Reilly, 2015.

[29] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An industrial case study of automatically identifying performance regression-causes," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR)*, 2014, pp. 232–241.

[30] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd International Conference on Performance Engineering (ICPE)*, 2012, pp. 299–310.

[31] A. Oliner, A. Ganapathi, and W. Xu, "Advances and challenges in log analysis," *Commun. ACM*, vol. 55, no. 2, pp. 55–61, Feb. 2012.

[32] H. Packard, "Shunra," http://www8.hp.com/ca/en/software-solutions/network-virtualization/, last accessed September 21 2016.

[33] W. Shang, A. E. Hassan, M. Nasser, and P. Flora, "Automated detection of performance regressions using regression models on clustered performance counters," in *Proceedings of the 6th International Conference on Performance Engineering (ICPE)*, 2015, pp. 15–26.

[34] W. Shang, Z. M. Jiang, B. Adams, A. E. Hassan, M. 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 18th Working Conference on Reverse Engineering (WCRE)*, 2011.

[35] N. Stankovic, "Patterns and tools for performance testing," in *In Processings of IEEE International Conference on Electro/Information Technology*, 2006, pp. 152–157.

[36] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Continuous validation of load test suites," in *Proceedings of the 5th International Conference on Performance Engineering (ICPE)*, 2014, pp. 259–270.

[37] M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, "Continuous validation of performance test workloads," *Automated Software Engineering*, pp. 1–43, 2016.

[38] W. Visser, "Who really cares if the program crashes?" in *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, 2009, pp. 5–5.

[39] E. J. Weyuker and A. Avritzer, "A metric for predicting the performance of an application under a growing workload," *IBM Syst. J.*, vol. 41, no. 1, pp. 45–54, Jan. 2002.

[40] E. J. Weyuker and F. I. Vokolos, "Experience with performance testing of software systems: Issues, an approach, and case study," *IEEE Trans. Softw. Eng.*, vol. 26, no. 12, pp. 1147–1156, Dec. 2000.

[41] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "Sherlog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010, pp. 143–154.