A Survey on Load Testing of Large-Scale Software Systems

Zhen Ming Jiang, *Member, IEEE* and Ahmed E. Hassan, *Member, IEEE*

Abstract—Many large-scale software systems must service thousands or millions of concurrent requests. These systems must be load tested to ensure that they can function correctly under load (i.e., the rate of the incoming requests). In this paper, we survey the state of load testing research and practice. We compare and contrast current techniques that are used in the three phases of a load test: (1) designing a proper load, (2) executing a load test, and (3) analyzing the results of a load test. This survey will be useful for load testing practitioners and software engineering researchers with interest in the load testing of large-scale software systems.

Index Terms—Software testing, load testing, software quality, large-scale software systems, survey

1 Introduction

Many large-scale systems ranging from e-commerce websites to telecommunication infrastructures must support concurrent access from thousands or millions of users. Studies show that failures in these systems tend to be caused by their inability to scale to meet user demands, as opposed to feature bugs [1], [2]. The failure to scale often leads to catastrophic failures and unfavorable media coverage (e.g., the meltdown of the Firefox website [3], the botched launch of Apple's MobileMe [4] and US Government's Health Care Website [5]). To ensure the quality of these systems, load testing is a required testing procedure in addition to conventional functional testing procedures, like unit testing and integration testing.

This paper surveys the state of research and practices in the load testing of large-scale software systems. This paper will be useful for load testing practitioners and software engineering researchers with interests in testing and analyzing large-scale software systems. Unlike functional testing, where we have a clear objective (pass/fail criteria), load testing can have one or more functional and non-functional objectives as well as different pass/fail criteria. As illustrated in Fig. 1, we propose the following three research questions on load testing based on the three phases of traditional software testing (test design, test execution and test analysis [6]):

1) *How is a proper load designed?* The *Load Design* phase defines the load that will be placed on the system during testing based on the test objectives (e.g., detecting functional and performance problems under load). There are two main schools of load designs: (1) designing realistic loads, which simulate workload that may occur in the field; or (2) designing fault-inducing loads, which are likely to expose load related problems. Once the load is designed, some optimization and reduction techniques could be applied to further improve various aspects of the load (e.g., reducing the duration of a load test). In this research question, we will discuss various load design techniques and explore a number of load design optimization and reduction techniques.
2) *How is a load test executed?* In this research question, we explore the techniques and practices that are used in the *Load Test Execution* phase. There are three different test execution approaches: (1) using live-users to manually generate load, (2) using load drivers to automatically generate load, and (3) deploying and executing the load test on special platforms (e.g., a platform which enables deterministic test executions). These three load test execution approaches share some commonalities and differences in the following three aspects: (1) setup, which includes deploying the system and configuring the test infrastructure and the test environment, (2) load generation and termination, and (3) test monitoring and data collection.
3) *How is the result of a load test analyzed?* In this research question, we survey the techniques used in the *Load Test Analysis* phase. The system behavior data (e.g., execution logs and performance counters) recorded during the test execution phase needs to be analyzed to determine if there are any functional or non-functional load-related problems. There are three general load test analysis approaches: (1) verifying against known thresholds (e.g., detecting violations in reliability requirements), (2) checking for known problems (e.g., memory leak detection), and (3) inferring anomalous system behavior.

The structure of this paper is organized as follows: Section 2 provides some background about this survey.

- *Z.M. Jiang is with the Software Construction, AnaLytics and Evaluation (SCALE) Lab, Department of Electrical Engineering and Computer Science, York University, Toronto, ON, Canada.*
  E-mail: zmjiang@cse.yorku.ca.
- *A.E. Hassan is with the Software Analysis and Intelligence (SAIL) Lab, School of Computing, Queen's University, Kingston, ON, Canada.*
  E-mail: ahmed@cs.queensu.ca.

Manuscript received 9 June 2014; revised 13 May 2015; accepted 31 May 2015. Date of publication 14 June 2015; date of current version 13 Nov. 2015.
Recommended for acceptance by M. Woodside.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2015.2445340

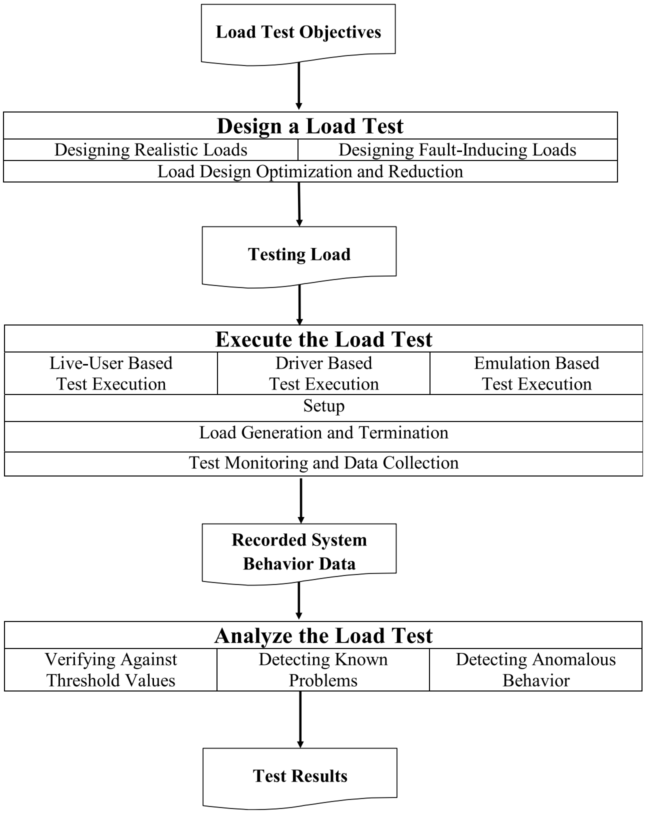
Load Test Objectives
Design a Load Test
Designing Realistic Loads
Designing Fault-Inducing Loads
Load Design Optimization and Reduction
Testing Load
Execute the Load Test
Live-User Based Test Execution
Driver Based Test Execution
Emulation Based Test Execution
Setup
Load Generation and Termination
Test Monitoring and Data Collection
Recorded System Behavior Data
Analyze the Load Test
Verifying Against Threshold Values
Detecting Known Problems
Detecting Anomalous Behavior
Test Results

Fig. 1. Load testing process.

Then, based on the flow of a load test, we discuss the techniques that are used in designing a load test (Section 3), in executing a load test (Section 4), and in analyzing the results of a load test (Section 5). Section 6 concludes our survey.

2 Background

Contrary to functional testing, which has clear testing objectives (pass/fail criteria), load testing objectives (e.g., performance requirements) are not clear in the early development stages [7], [8] and are often defined later on a case-by-case basis (e.g., during the initial observation period in a load test [9]). There are many different interpretations of load testing, both in the context of academic research and industrial practices (e.g., [10], [11], [12], [13]). In addition, the term load testing is often used interchangeably with two other terms: performance testing (e.g., [14], [15], [16]) and stress testing (e.g., [11], [17], [18]). In this section, we first provide our "own" working definition of load testing by contrasting among various interpretations of load, performance and stress testing. Then we briefly explain our selection process of the surveyed papers.

2.1 Definitions of Load Testing, Performance Testing and Stress Testing

We find that these three types of testing share some common aspects, yet each has its own focus. In the rest of this section, we first summarize the various definitions of the testing types. Then we illustrate their relationship with respect to each other. Finally, we present our definition of load testing. Our load testing definition unifies the existing load testing interpretations as well as performance and stress testing interpretations, which are also about load testing. There could be other aspects/objectives (e.g., additional non-functional requirements) of load testing that we may have missed due to our understanding of the objectives of software testing and our survey process.

Table 1 outlines the interpretations of load testing, performance testing and stress testing in the existing literature. The table breaks down various interpretations of load, performance and stress testing along the following dimensions:

- *Objectives* refer to the goals that a test is trying to achieve (e.g., detecting performance problems under load);
- *Stages* refer to the applicable software development stages (e.g., design, implementation, or testing), during which a test occurs;
- *Terms* refer to the terminology used in the relevant literature (e.g., load testing and performance testing);
- *Is It Load Testing?* indicates whether we consider such cases (performance or stress testing) to be load testing based on our working definition of load testing. The criteria for deciding load, performance and stress testing is presented later (Section 2.2).

2.1.1 Load Testing

Load testing is the process of assessing the behavior of a system under load in order to detect load-related problems. The rate at which different service requests are submitted to the system under test (SUT) is called the *load* [73]. The load-related problems can be either functional problems that appear only under load (e.g, such as deadlocks, racing, buffer overflows and memory leaks [23], [24], [25]) or non-functional problems which are violations in non-functional quality-related requirements under load (e.g., reliability [23], [37], stability [10], and robustness [30]).

Load testing is conducted on a system (either a prototype or a fully functional system) rather than on a design or an architectural model. In the case of missing non-functional requirements, the pass/fail criteria of a load test are usually derived based on the "no-worse-than-before" principle. The "no-worse-than-before" principle states that the non-functional requirements of the current version should be at least as good as the prior version [26]. Depending on the objectives, the load can vary from a normal load (the load expected in the field when the system is operational [23], [37]) or a stress load (higher than the expected normal load) to uncover functional or non-functional problems [29].

2.1.2 Performance Testing

Performance testing is the process of measuring and/or evaluating performance related aspects of a software system. Examples of performance related aspects include response time, throughput and resource utilizations [24], [25], [74].

Performance testing can focus on parts of the system (e.g., unit performance testing [61] or GUI performance testing [75]), or on the overall system [24], [25], [48]. Performance testing can also study the efficiency of various design/architectural decisions [63], [64], [65], different algorithms [58], [59] and various system configurations [48],

TABLE 1
Interpretations of Load Testing, Performance Testing and Stress Testing

Objectives	Stages	Terms	Is It Load Testing?
Detecting functional problems under load	Testing (After Conventional Functional Testing)	Load Testing [14], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], Stress Testing [18], [26], [30], [31], [32], [33], [34]	Yes
Detecting violations in performance requirements under load	Testing (After Conventional Functional Testing)	Load Testing [10], Performance Testing [2], [10], [35], Stress Testing [36], [37], [38]	Yes
Detecting violations in reliability requirements under load	Testing (After Conventional Functional Testing)	Load Testing [10], [21], [22], [23], [37], Reliability Testing [10]	Yes
Detecting violations in stability requirements under load	Testing (After Conventional Functional Testing)	Load Testing [10], Stability Testing [10]	Yes
Detecting violations in robustness requirements under load	Testing (After Conventional Functional Testing)	Load Testing [10], Stress Testing [10], [30]	Yes
Measuring and/or evaluating system performance under load	Implementation	Performance Testing [39], [40], [41], [42]	Depends
	Testing (After Conventional Functional Testing)	Performance Testing [12], [13], [15], [16], [43], [44], [45], [46], [47], [48], [49], Load Testing [15], [50], Stress Testing [37], [38], [51], [52], [53], [54]	Depends
	Maintenance (Regression Testing)	Performance Testing [55], Regression Benchmarking [56], [57]	Depends
Measuring and/or evaluating system performance without load	Testing (After Conventional Functional Testing)	Performance Testing [58], [59], [60]	No
Measuring and/or evaluating component/unit performance	Implementation	Performance Testing [61]	No
Measuring and/or evaluating various design alternatives	Design	Performance Testing [62], [63], [64], [65], Stress Testing [66], [67], [68], [69]	No
	Testing (After Conventional Functional Testing)	Performance Testing [70]	No
Measuring and/or evaluating system performance under different configurations	Testing (After Conventional Functional Testing)	Performance Testing [48], [71], [72]	No

[71], [72]. Depending on the types of systems, performance testing can be conducted with load (e.g., e-commerce systems [15], [16], [47], middle-ware systems [2], [35] or service-oriented distributed systems [39]), or single user request (e.g., mobile applications [60] or desktop applications [58], [59]).

Contrary to load testing, the objectives of performance testing are broader. Performance testing (1) can verify performance requirements [48] or in case of absent performance requirements, the pass/fail criteria are derived based on the "no-worse-than-previous" principle [26] (similar to load testing); or (2) can be exploratory (no clear pass/fail criteria). For example, one type of performance testing aims to answer the what-if questions like "what is system performance if we change this software configuration option or if we increase the number of users?" [47], [48], [76], [77].

2.1.3 Stress Testing

Stress testing is the process of putting a system under extreme conditions to verify the robustness of the system and/or to detect various load-related problems (e.g., memory leaks and deadlocks). Examples of such conditions can either be load-related (putting system under normal [36], [37], [38] or extreme heavy load [14], [26], [27], [37]), limited computing resources (e.g., high CPU [78]), or failures (e.g., database failure [20]). In other cases, stress testing is used to evaluate the efficiency of software designs [66], [67], [68], [69].

2.2 Relationships between Load Testing, Performance Testing and Stress Testing

As Dijkstra pointed out in [79], software testing can only show the presence of bugs but not their absence. Bugs are the behavior of systems which deviate from the specified requirements. Hence, we define our unified definition of load testing that is used in this paper is as follows:

> *Load testing is the process of assessing system behavior under load in order to detect problems due to one or both of the following reasons: (1) functional-related problems (i.e., functional bugs that appear only under load), and (2) non-functional problems (i.e., violations in non-functional quality-related requirements under load).*

Comparatively, *performance testing* is used to measure and/or evaluate performance related aspects (e.g., response time, throughput and resource utilizations) of algorithms, designs/architectures, modules, configurations, or the overall systems. *Stress testing* puts a system

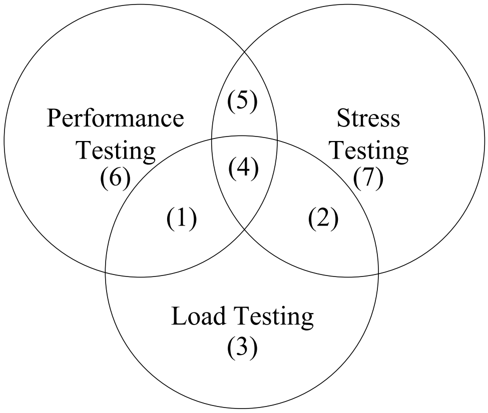

Fig. 2. Relationships among load, performance and stress testing.

under extreme conditions (e.g., higher than expected load or limited computing resources) to verify the robustness of the system and/or detect various functional bugs (e.g., memory leaks and deadlocks).

There are commonalities and differences among the three types of testing, as illustrated in the Venn Diagram shown in Fig. 2. We use an e-commerce system as a working example to demonstrate the relation across these three types of testing techniques.

1) *Scenarios considered as both load testing and performance testing*. The e-commerce system is required to provide fast response under load (e.g., millions of concurrent client requests). Therefore, testing is needed to validate the system's performance under the expected field workload. Such type of testing is not considered to be stress testing as the testing load does not exceed the expected field workload.
2) *Scenarios considered as both load testing and stress testing*. The e-commerce system must be be robust under extreme conditions. For example, this system is required to stay up even under bursts of heavy load (e.g., flash crowd [15]). In addition, the system should be free of resource allocation bugs, like deadlocks or memory leaks [34].

   This type of testing, which imposes a heavy load on the system to verify the system's robustness and to detect resource allocation bugs, is considered as both stress testing and load testing. Such testing is not performance testing, as software performance is not one of the testing objectives.
3) *Scenarios only considered as load testing*. Although this system is already tested manually using a small number of users to verify the functional correctness of a service request (e.g., the total cost of a shopping cart is calculated correctly when a customer checks out), the correctness of the same types of requests should be verified under hundreds or millions of concurrent users.

   The test, which aims to verify the functional correctness of a system under load is considered only as a load test. This scenario is not performance testing, as the objective is not performance related; nor is this scenario considered as stress testing, as the testing conditions are not extreme.
4) *Scenarios considered as load, performance and stress testing*. This e-commerce website can also be accessed using smartphones. One of the requirements is that the end-to-end service request response time should be reasonable even under poor cellular network conditions (e.g., packet drops and packet delays).

   The type of test used to validate the performance requirements of the SUT with limited computing resources (e.g., network conditions), can be considered as all three types of testing.
5) *Scenarios considered as performance testing and stress testing*. Rather than testing the system performance after the implementation is completed. The system architect may want to validate whether a compression algorithm can efficiently handle large image files (processing time and resulting compressed file size). Such testing is not considered to be load testing, as there is no load (concurrent access) applied to the SUT.
6) *Scenarios Considered as Performance Testing Only*. In addition to writing unit tests to check the functional correctness of their code, the developers are also required to unit test the code performance. The test to verify the performance of one unit/component of the system is considered only as performance testing.

   In addition, the operators of this e-commerce system need to know the system deployment configurations to achieve the maximal performance throughput using minimal hardware costs. Therefore, performance testing should be carried out to measure the system performance under various database or webserver configurations. The type of test to evaluate the performance of different architectures/algorithms/configurations is only considered as performance testing.
7) *Scenarios considered as stress testing only*. Developers have implemented a smartphone application for this e-commerce system to enable users to access and buy items from their smartphones. This smartphone application is required to work under sporadic network conditions. This type of test is considered as stress testing, since the application is tested under extreme network condition. This testing is not considered to be performance testing, since the objective is not performance related; nor is this scenario considered as load testing, as the test does not involve load.

2.3 Our Paper Selection Process

We first search the following three keywords on the General scholarly article search engines (DBLP searches [80] and Google Scholar [81]): "load test", "performance test" and "stress test". Second, we filter irrelevant papers based on the paper titles, publication venues and abstracts. For example, results like "Test front loading in early stages of automotive software development based on AUTOSAR" are filtered out. We also remove performance and stress testing papers that are not related to load testing (e.g., "Backdrive Stress-Testing of CMOS Gate Array Circuits"). Third, we add additional papers and tools based on the related work sections from relevant load testing papers, which do not contain the above three keywords. Finally, we include relevant papers that cite these papers, based on the "Cited by" feature from Microsoft Academic Search [82], Google Scholar [81], ACM Portal [83] and IEEE Explore [84]. For example, papers like [85], [86] are included, because they cite [23] and [87], respectively.

In the end, we have surveyed a total of 147 papers and tools between the year $1993 - 2013$. To verify the completeness of the surveyed papers, the final results include all the papers we knew beforehand to be related to load testing (e.g., [11], [15], [17], [21], [23], [87], [88]).

3 Research Question 1: How Is a Proper Load Designed?

The goal of the load design phase is to devise a load, which can uncover load-related problems under load. Based on the load test objectives, there are two general schools of thought for designing a proper load to achieve such objectives:

1) *Designing realistic loads*. As the main goal of load testing is to ensure that the SUT can function correctly once it is deployed in the field, one school of thought is to design loads, which resemble the expected usage once the system is operational in the field. If the SUT can handle such loads without functional and non-functional issues, the SUT would have passed the load test.

   Once the load is defined, the SUT executes the load and the system behavior is recorded. Load testing practitioners then analyze the recorded system behavior data to detect load-related problems. Test durations in such cases are usually not clearly defined and can vary from several hours to a few days depending on the testing objectives (e.g., to obtain steady state estimates of the system performance under load or to verify that the SUT is deadlock-free) and testing budget (e.g., limited testing time). There are two approaches proposed in the literature to design realistic testing loads as categorized in [53]:

a) *The aggregate-workload based load design approach*. The aggregate-workload based load design approach aims to generate the individual target request rates. For example, an e-commerce system is expected to handle three types of requests with different transaction rates: ten thousand purchasing requests per second, three million browsing requests per second, and five hundred registration requests per second. The resulting load, using the aggregate-workload based load design approach, should resemble these transaction rates.

b) *The use-case based load design approach*. The use-case (also called *user equivalent* in [53]) based approach is more focused on generating requests that are derived from realistic use cases. For example, in the aforementioned e-commerce system, an individual user would alternate between submitting page requests (browsing, searching and purchasing) and being idle (reading the webpage or thinking). In addition, a user cannot purchase an item before he/she logs into the system.

2) *Designing fault-inducing loads*. Another school of thought aims to design loads, which are likely to cause functional or non-functional problems under load. Compared to realistic loads, the test duration using faulting-inducing loads are usually deterministic and the test results are easier to analyze. The test durations in these cases are the time taken for the SUT to enumerate through the loads or the time until the SUT encounters a functional or non-functional problem.

There are two approaches proposed in the literature for designing fault-inducing loads:

a) *Deriving fault-inducing loads by analyzing the source code*. This approach uncovers various functional and non-functional problems under load by systematically analyzing the source code of the SUT. For example, by analyzing the source code of our motivating e-commerce system example, load testing practitioners can derive loads that exercise these potential functional (e.g., memory leaks) and non-functional (e.g., performance issues) weak spots under load.

b) *Deriving fault-inducing loads by building and analyzing system models*. Various system models abstract different aspects of system behavior (e.g., performance models for the performance aspects). By systematically analyzing these models, potential weak spots can be revealed. For example, load testing practitioners can build performance models in the aforementioned e-commerce system, and discover loads that could potentially lead to performance problems (higher than expected response time).

We introduce the following dimensions to compare among various load design techniques as shown in Table 2:

- *Techniques* refer to the names of the load design techniques (e.g., step-wise load);
- *Objectives* refer to the goals of the load (e.g., detecting performance problems);
- *Data sources* refer to the artifacts used in each load design technique. Examples of artifacts can be past field data or operational profiles. *Past field data* could include web access logs, which record the identities the visitors and their visited sites, and database auditing logs, which show the various database interactions. An *Operational Profile* describes the expected field usage once the system is operational in the field [23]. For example, an operational profile for an e-commerce system would describe the number of concurrent requests (e.g., browsing and purchasing) that the system would experience during a day. The process of extracting and representing the expected workload (operational profile) in the field is called *Workload Characterization* [89]. The goal of workload characterization is to extract the expected usage from hundreds or millions hours of past field data. Various workload characterization techniques have been surveyed in [89], [90], [91].

TABLE 2
Load Design Techniques

Techniques	Objectives	Data Sources	Output	References
Realistic Load Design - (1) Aggregate-Workload Based Load Design Approach (Section 3.1.1)				
Steady Load	Detecting Functional and Non-Functional Problems Under Load	Operational Profiles, Past Usage Data	One Configuration of Workload Mix and Workload Intensity	[45], [92]
Step-wise Load		Operational Profiles, Past Usage Data	Multiple Configurations of Workload Mixed and Workload Intensities	[30], [93], [36], [94], [95], [96], [97], [98]
Extrapolated Load		Beta-user Usage Data, Interviewing Domain Experts and Competitions Data	One or More Configurations of Workload Mix and Workload Intensities	[44], [99]
Realistic Load Design - (2) Use-Case Based Load Design Approach (Section 3.1.2)				
Testing Loads Derived using UML Models	Detecting Functional and Non-Functional Problems	UML Use Case Diagrams, UML Activity Diagrams, Operational Profile	UML Diagrams Tagged with Request Rates	[100], [101], [102], [103]
Testing Loads Derived using Markov Models		Past Usage Data	Markov Chain Models	[104], [105], [16]
Testing Loads Derived using Stochastic Form-oriented Models		Operational Profile, Business Requirements, User configurations	Stochastic Form-oriented Models	[106], [107]
Testing Loads Derived using Probabilistic Timed Automata		User Configurations	Probabilistic Timed Automata	[108], [109], [110]
Fault-Inducing Load Design - (1) Deriving Load from Analyzing the Source Code (Section 3.2.1)				
Testing Loads Derived using Data Flow Analysis	Detecting Functional Problems (memory leaks)	Source Code	Testing Loads Lead to Code Paths with Memory Leaks	[18]
Testing Loads Derived using Symbolic Execution	Detecting Functional Problems (high memory usage) and Performance Problems (high response time)	Source Code, Symbolic Execution Analysis Tools	Testing Loads Lead to Problematic Code Paths with Performance Problems	[111], [29]
Fault-Inducing Load Design - (2) Deriving Load from Building and Analyzing System Models (Section 3.2.2)				
Testing Loads Derived using Linear Programs	Detecting Performance Problems (audio and video not in sync)	Resource Usage Per Request	Testing Loads Lead to Performance Problems (high response time)	[38], [54]
Testing Loads Derived using Genetic Algorithms	Detecting Performance Problems (high response time)	Resource Usage and Response Time Per Task		[112], [113]

- *Output* refers to the types of output from each load design technique. Examples can be workload configurations or usage models. *Workload configuration* refers to one set of workload mix and workload intensity (covered in Section 3.1.1). *Models* refer to various abstracted system usage models (e.g., the Markov chain).
- *References* refer to the list of literatures, which propose each technique.

Both load design schools of thought (realistic versus fault-inducing load designs) have their advantages and disadvantages: In general, loads resulting from realistic-load based design techniques can be used to detect both functional and non-functional problems. However, the test durations are usually longer and the test analysis is more difficult, as the load testing practitioners have to search through large amounts of data to detect load-related problems. Conversely, although loads resulting from fault-inducing load design techniques take less time to uncover potential functional and non-functional problems, the resulting loads usually only cover a small portion of the testing objectives (e.g., only detecting the violations in the performance requirements). Thus, there are load optimization and reduction techniques proposed to mitigate the deficiencies of each load design technique.

This section is organized as follows: Section 3.1 covers the realistic load design techniques. Section 3.2 covers the fault-inducing load design techniques. Section 3.3 discusses the test optimization and reduction techniques used in the load design phase. Section 3.4 summarizes the load design techniques and proposes a few open problems.

3.1 Designing Realistic Loads

In this section, we discuss the techniques used to design loads, which resemble the realistic usage once the system is operational in the field. Sections 3.1.1 and 3.1.2 cover the techniques from the Aggregate-Workload and the Use-Case based load design approaches, respectively.

3.1.1 Aggregate-Workload Based Load Design Techniques

Aggregate-workload based load design techniques characterize loads along two dimensions: (1) *Workload Intensity*, and (2) *Workload Mix*:

- The *Workload Intensity* refers to the rate of the incoming requests (e.g., browsing, purchasing and searching), or the number of concurrent users;
- The *Workload Mix* refers to the ratios among different types of requests (e.g., 30 percent browsing, 10 percent purchasing and 60 percent searching).

Three load design techniques have been proposed to characterize loads with various workload intensity and workload mix:

1) *Steady load*. The most straightforward aggregate-workload based load design technique is to devise a steady load, which contains only one configuration of the workload intensity and workload mix throughout the entire load test [45]. A steady load can be inferred from past data or based on an existing operational profile. This steady load could be the normal expected usage or the peak time usage depending on the testing objectives. Running the SUT using a steady load can be used to verify the system resource requirements (e.g., memory, CPU and response time) [92] and to identify resource usage problems (e.g., memory leaks) [45].

2) *Step-wise load*. A system in the field normally undergoes varying load characteristics throughout a normal day. There are periods of light usage (e.g., early in the morning or late at night), normal usage (e.g., during the working hours), and peak usages (e.g., during lunch time). It might not be possible to load test a system using a single type of steady load. Step-wise load design techniques would devise loads consisting of multiple types of load, to model the light/normal/peak usage expected in the field.

   Step-wise load testing keeps the workload mix the same throughout the test, while increasing the workload intensity periodically [30], [36], [93], [94], [95], [96], [97], [98]. Step-wise load testing, in essence, consists of multiple levels of steady load. Similar to the steady load approach, the workload mix can be derived using the past field data or an operational profile. The workload intensity varies from system to system. For example, the workload intensity can be the number of users, the normal and peak load usages, or even the amount of results returned from web search engines.

3) *Load extrapolation based on partial or incomplete data*. The steady load and the step-wise load design techniques require an existing operational profile or past field data. However, such data might not be available in some cases: For example, newly developed systems or systems with new features have no existing operational profile or past usage data. Also, some past usage data may not be available due to privacy concerns. To cope with these limitations, loads are extrapolated from the following sources:

   - *Beta-usage data*. Savoia [99] proposes to analyze log files from a limited beta usage and to extrapolate the load based on the number of expected users in the actual field deployment.
   - *Interviews with domain experts*. Domain experts like system administrators, who monitor and manage deployed systems in the field, generally have a sense of system usage patterns. Barber [44] suggests to obtain a rough estimate of the expected field usage by interviewing such domain experts.
   - *Extrapolation from using competitors' data*. Barber [44] argues that in many cases, new systems likely do not have a beta program due to limited time and budgets and interviewing domain experts might be challenging. Therefore, he proposes an even less formal approach to characterize the load based on checking out published competitors' usage data, if such data exists.

3.1.2 Use-Case Based Load Design Techniques

The main problem associated with the aggregate-workload based load design approach is that the loads might not be realistic/feasible in practice, because the resulting requests might not reflect individual use cases. For example, although the load can generate one million purchasing requests per second, some of these requests would fail due to invalid user states (e.g., some users do not have items added to their shopping carts yet).

However, designing loads reflecting realistic use-cases could be challenging, as there may be too many use cases available for the SUT. Continuing with our motivating example of the e-commerce system, different users can follow different navigation patterns: some users may directly locate items and purchase them. Some users may prefer to browse through a few items before buying the items. Some other users may just browse the catalogs without buying. It would not be possible to cover all the combinations of these sequences. Therefore, various usage models are proposed to abstract the use cases from thousands and millions of user-system interactions. In the rest of this section, we discuss four load design techniques based on usage models.

1) *Testing loads derived using UML models*. UML diagrams, like Activity Diagrams and Use Case Diagrams, illustrate detailed user interactions in the system. One of the most straight forward use-case based load design techniques is to tag load information on the UML Activity Diagram [100], [101], [102] or Use Case Diagram [100], [103] with probabilities. For example, the probability beside each use case is the likelihood that a user triggers that action [103]. For example, a user is more likely to navigate around (40 percent probability) than to delete a file (10 percent probability).

2) *Testing loads derived using Markov Chain models*. The problem with the UML-based testing load is that the

UML Diagrams may not be available or such information may be too detailed (e.g., hundreds of use cases). Therefore, techniques are needed to abstract load information from other sources. A Markov Chain, which is also called the User Behavior Graph [16], consists of a finite number of states and a set of state transition probabilities between these states. Each state has a steady state probability associated with it. If two states are connected, there is a transition probability between these two states.

Markov Chains are widely used to generate load for web-based e-commerce applications [16], [104], [105], since Markov chains can be easily derived from the past field data (web access logs [16]). Each entry of the log is a URL, which consists of the requested webpages and "parameter name = parameter value" pairs. Therefore, sequences of user sessions can be recovered by grouping sequences of request types belonging to the same session. Each URL requested becomes one state in the generated Markov chain. Transition probabilities between states represent real user navigation patterns, which are derived using the probabilities of a user clicking page B when he/she is on page A.

During the course of a load test, user action sequences are generated based on the probabilities modeled in the Markov chain. The think time between successive actions is usually generated randomly based on a probabilistic distribution (e.g., a normal distribution or an exponential distribution) [16], [105]. As the probability in the Markov chain only reflects the average behavior of a certain period of time, Barros et al. [104] recommend the periodical updating of the Markov chain based on the field data in order to ensure that load testing reflects the actual field behavior.

3) *Testing loads derived using stochastic form-oriented models*. Stochastic Form-oriented Model is another technique used to model a sequence of actions performed by users. Compared to the testing loads represented by the Markov Chain models, a *Stochastic Form-oriented model* is richer in modeling user interactions in web-based applications [106]. For example, a user login action can either be a successful login and a redirect to the overview page, or a failure login and a redirect back to the login page. Such user behavior is difficult to model in a Markov chain [106], [107].

   Cai et al. [114], [115] propose a toolset that automatically generates a load for a web application using a three-step process: First, the website is crawled by a third party web crawler and the website's structural data is recovered. Then, their proposed toolset lays out the crawled web structure using a Stochastic Form-Oriented Model and prompts the performance engineer to manually specify the probabilities between the pages and actions based on an operational profile.

4) *Testing loads derived using probabilistic timed automata*. Compared to the Markov Chain and the Stochastic Form-oriented Models, Probability Timed Automata is an abstraction which provides support for user action modeling as well as timing delays [108], [109], [110]. Similar to the Markov chain model, a Probabilistic Timed Automata contains a set of states and transition probabilities between states. In addition, for each transition, a Probabilistic Timed Automata contains the time delays before firing the transition. The timing delays are useful for modeling realistic user behaviors. For example, a user could pause for a few seconds (e.g., reading the page contents) before triggering the next action (e.g., purchasing the items).

3.2 Designing Fault-Inducing Loads

In this section, we cover the load design technique from the school of fault-inducing load design. There are two approaches proposed to devise potential fault-inducing testing loads: (1) by analyzing the source code (Section 3.2.1), and (2) by building and analyzing various system models (Section 3.2.2).

3.2.1 Deriving Fault-Inducing Loads via Source Code Analysis

There are two techniques proposed to automatically analyze the source code for specific problems. The first technique is trying to locate specific code patterns, which lead to known load-related problems (e.g., memory allocation patterns for memory allocation problems). The second technique uses model checkers to systematically look for memory and performance problems. In this section, we only look at the techniques which analyze the source code statically. There are also load generation techniques which leverage dynamic analysis techniques. However, these techniques are tightly coupled with the load execution: the system behavior is monitored and analyzed, while new loads are generated. We have categorized such techniques as "dynamic-feedback-based load generation and termination techniques" in Section 4.2.3.

1) *Testing loads derived using data flow analysis*. Load sensitive regions are code segments, whose correctness depends on the amount of input data and the duration of testing [18]. Examples of load sensitive regions can be code dealing with various types of resource accesses (e.g., memory, thread pools and database accesses). Yang et al. [18] use data flow analysis of the system's source code to generate loads, which exercise the load sensitive regions. Their technique detects memory related faults (e.g., memory allocation, memory deallocation and pointers referencing).

2) *Testing loads derived using symbolic executions*. Rather than matching the code for specific patterns (e.g., the resource accesses patterns in [18]), Zhang et al. [29], [111] use symbolic test execution techniques to generate loads, which can cause memory or performance problems. Symbolic execution is a program analysis technique, which can automatically generates input values corresponding to different code paths.

   Zhang et al. use the symbolic execution to derive two types of loads:

a) *Testing loads causing large response time*. Zhang et al. assign a time value for each step along the code path (e.g., 10 for an invoking routing and 1 for other routines). Therefore, by summing up the costs for each code path, they can identify the paths that lead to the longest response time. The values that satisfy the path constraints form the loads.

b) *Testing loads causing large memory consumptions*. Rather than tracking the time, Zhang et al. track the memory usage at each step along the code path. The memory footprint information is available through a Symbolic Execution tool, (e.g., the Java Path Finder (JPF)). Zhang et al. use the JPF's built-in object life cycle listener mechanism to track the heap size of each path. Paths leading to large memory consumption are identified and values satisfying such code paths form the loads.

3.2.2 Deriving Fault-Inducing Loads by Building and Analyzing System Models

In the previous section, we have presented load design techniques which analyze the source code of a system to explore potential problematic regions/paths. However, some of those techniques only work for a particular type of systems. For example, [29], [111] only work for Java-based systems. In addition, in some cases, source code might not be directly accessible. Hence, there are also techniques that have been proposed to automatically search for potential problematic loads using various system models.

1) *Testing loads derived using petri nets and linear programs*. Online multimedia systems have various temporal requirements: (1) *Timing requirements:* audio and video data streams should be delivered in sequence and following strict timing deadlines; (2) *Synchronization requirements:* video and audio data should be in synch with each other; (3) *Functional requirements:* some videos can only be displayed after collecting fee.

   Zhang et al. [38], [54] propose a two-step technique that automatically generates loads, which can cause a system to violate the synchronization and responsive requirements while satisfying the business requirements. Their idea is based on the belief that timing and synchronization requirements usually fail when the SUT's resources are saturated. For example, if the memory is used up, the SUT would slow down due to paging.

a) *Identify data flows using a Petri Net*. The online multimedia system is modeled using a *Petri Net*, which is a technique that models the temporal constraints of a system. All possible user action sequences can be generated by conducting reachability analysis, which explores all the possible paths, on the Petri Net. For example, a new video action C cannot be fired until the previous video action A and audio action B are both completed.

b) *Formulate system behavior into a linear program in order to identify performance problems*.

   Linear programming is used to identify the sequences of user actions, which can trigger performance problems. Linear programming systematically searches for optimal solutions based on certain constraints. A linear program contains the following two types of artifacts: an objective function (the optimal criteria) and a set of constraints. The objective function is to maximize or minimize a linear equation. The constraints are a set of linear equations or inequalities. The sequence of arrivals of the user action sequences is formulated using linear constraints. There are two types of constraint functions: One constraint function ensures the total testing time is within a pre-specified value (the test will not run for too long). The rest of the constraint functions formulate the temporal requirements derived using the possible user action sequences, as the resource requirements (e.g., CPU, memory, network bandwidth) associated with each multimedia object (video or audio) are assumed to be known. The objective function is set to evaluate whether the arrival time sequence would cause the saturations of one or more system resources (CPU and network).

(2) *Testing loads derived using genetic algorithms*. An SLA, *Service Level Agreement*, is a contract with potential users on the non-functional properties like response time and reliability as well as other requirements like costs. Penta et al. [113] and Gu and Ge [112] uses Genetic Algorithms to derive loads causing SLA or quality of service (QoS) requirement violations (e.g., response time) in service-oriented systems. Like linear programming, *Genetic Algorithms*, is a search algorithm, which mimics the process of natural evolution for locating optimal solutions towards a specific goal.

The genetic algorithms are applied twice to derive potential performance sensitive loads:

a) Penta et al. [113] use the genetic algorithm technique that is proposed by Canfora et al. [116], in order to identify risky workflows within a service. The response time for the risky workflows should be as close to the SLA (high response time) as possible.

b) Penta et al. [113] apply the genetic algorithm to generate loads that cover the identified risky workflow and violate the SLA.

3.3 Load Design Optimization and Reduction Techniques

In this section, we discuss two classes of load design optimization and reduction techniques aimed at improving various aspects of load design techniques. Both classes of techniques are aimed at improving the realistic load design techniques.

- *Hybrid load optimization techniques*. The aggregate-workload based techniques focus on generating the desired workload, but fail to mimic realistic user behavior. The user-equivalent based techniques focus on mimicking the individual user behaviour, but fail to match the expected overall workload. The hybrid load optimization techniques (Section 3.3.1)

TABLE 3
Test Reduction and Optimization Techniques that Are Used in the Load Design Phase

Techniques	Target Load Design Techniques	Optimizing and Reducing Aspects	Data Sources	References
Hybrid Load Optimization	All Realistic Load Design Techniques	Combining the strength of aggregate-workload and use-case based load design techniques	Past usage data	[51], [53], [117]
Extrapolation	Step-wise Load Design	Reducing the number of workload intensity levels	Step-wise testing loads, past usage data	[15], [16], [118], [119]
Deterministic State	All Realistic Load Design Techniques	Reducing repeated execution of the same scenarios	Realistic testing loads	[21], [22], [23]

aim to combine the strength of the aggregate-workload and use-case based load design approaches. For example, for our example e-commerce system, the resulting load should resemble the targeted transaction rates and mimic real user behavior.
- *Optimizing and reducing the duration of a load test*. One major problem with loads derived from realistic load testing is that the test durations in these testing loads are usually not clearly defined (i.e., no clear stopping rules). The same scenarios are repeatedly-executed over several hours or days.

Table 3 compares the various load design optimization and reduction techniques along the following dimensions:

- *Techniques* refer to the used load design optimization and reduction techniques (e.g., the hybrid load optimization techniques).
- *Target load design techniques* refer to the load design techniques that the reduction or optimization techniques are intended to improve. For example, the hybrid load optimization techniques combine the strength of aggregate-workload and use-case based load design techniques.
- *Optimization and reducing aspects* refer to the aspects of the current load design that the optimization and reduction techniques attempt to improve. One example is to reduce the test duration.
- *References* refer to the list of literatures, which propose each technique.

3.3.1 Hybrid Load Optimization Techniques

Hybrid load optimization techniques aim to better model the realistic load for web-based e-commerce systems [53], [117]. These techniques consist of the following three steps:

- *Step 1—Extracting Realistic Individual User Behavior From Past Data*. Most of the e-commerce systems record past usage data in the form of web access logs. Each time a user hits a webpage, an entry is recorded in the web access logs. Each log entry is usually a URL (e.g., the browse page or the login page), combined with some user identification data (e.g., session IDs). Therefore, individual user action sequences, which describe the step-by-step user actions, can be recovered by grouping the log entries with user identification data.
- *Step 2—Deriving Targeted Aggregate Load By Carefully Arranging the User Action Sequence Data*. The aggregate load is achieved by carefully arranging and stacking up the user action sequences (e.g., two concurrent requests are generated from two individual user action sequences). There are two proposed techniques to calculate user action sequences:
  1) *Matching the peak load by compressing multiple hours worth of load*. Burstiness refers to short uneven spikes of requests. One type of burstiness is caused by the *flash crowd*. The phenomenon where a website suddenly experiences a heavier than expected request rate. An example of flash crowd includes when many users flocked to the news sites like CNN.com during the 9/11 incident, or during the World Cup period, the FIFA website was often more loaded when a goal was scored. During the flash crowd incident, the load could be several times higher than the expected normal load. Incorporating realistic burstiness into load testing is important to verify the capacity of a system [120].

     Maccabee and Ma [117] squeeze multiple one-hour user action sequences together into one-hour testing load to generate a realistic peak load, which is several times higher than the normal load.
  2) *Matching the specific request rates by linear programs*. Maccabee and Ma's [117] technique is simple and can generate higher than normal load to verify the system capacity and guard against problems like a flash crowd. However, their technique has problems like coarse-grained aggregate load, which cannot reflect the normal expected field usage. For example, the individual requests rates (e.g., browsing or purchasing rates) might not match with the targeting request rates. Krishnamurthy et al. [52] use linear programming to systematically arrange user action sequences, which match with the desired workload.
- *Step 3—Specifying the inter-arrival time between user actions*. There is a delay between each user action, when the user is either reading the page or thinking about what to do next. This delay is called the "think time" or the "inter-arrival time" between actions. The think time distribution among the user action sequences is specified manually in [52], [53]. Casale et al. [51] extend the technique in [52], [53] to create realistic burstiness. They use a burstiness level

metrics, called the *Index of Dispersion* [120], which can be calculated based on the inter-arrival time between requests. They use the same constraint functions as [52], [53], but a different non-linear objective function. The goal of the objective function is to find the optimal session mix, whose *index of dispersion* is as close to the real-time value as possible.

The hybrid technique outputs the exact individual user actions during the course of the actions. The advantage of such output is to avoid some of the unexpected system failures from other techniques like the Markov chains [53]. However, special load generators are required to take such input and generate the testing loads. In addition, the scalability of the approach would be limited by the machine's memory, as the load generators need to read in all the input data (testing user actions at each time instance) at once.

3.3.2 Optimizing and Reducing the Duration of a Load Test

Two techniques have been proposed to systematically optimize and reduce the load test duration for the realistic-load-based techniques. One technique aims at reducing a particular load design technique (step-wise load testing). The other technique aims at optimizing and reducing the realistic load design techniques by adding determinism.

1) *Load test reduction by extrapolation*. Load testing needs to be conducted at various load levels (e.g., number of user levels) for step-wise load testing. Rather than examining the system behavior under all load levels, Menasce et al. [15], [16] propose to only test a few load levels and extrapolate the system performance at other load levels. Weyuker et al. [119] propose a metric, called the Performance Nonscalability Likelihood (PNL). The PNL metric is derived from the past usage data and can be used to predict the workload, which will likely cause performance problems.

   Furthermore, Leganza [118] proposes to extrapolate the load testing data from the results conducted on a lower number of users onto the actual production workload (300 users in testing versus $1,500$ users in production) to verify whether the current SUT and hardware infrastructure can handle the desired workload.

2) *Load test optimization and reduction by deterministic states*. Rather than repeatedly executing a set of scenarios over and over, like many of the aggregate-workload based load design techniques (e.g., steady load and Markov-chain), Avritzer et al. [10], [21], [22], [23] propose a load optimization technique, called the *Deterministic State Testing*, which ensures each type of load is only executed once.

   Avritzer et al. characterize the testing load using states. Each state measures the number of different active processing jobs at the moment. Each number in the state represents the number of active requests of a particular request. Suppose our e-commerce system consists of four scenarios: registration, browsing, purchasing and searching. The state $(1,0,0,1)$ would indicate that currently only there is one registration request and one search request active and the state $(0,0,0,0)$ would indicate that the system is idle. The probability of these states, called "Probability Mass Coverage", measures the likelihood that the testing states is going to be covered in the field. These probabilities are calculated based on the production data. The higher the probability of one particular state, the more likely it is going to happen in the field.

   Load test optimization can also be achieved by making use of the probability associated with each state to prioritize tests. If time is limited, only a small set of states with a high probability of occurrence in the field can be selected.

   In addition to reducing the test durations, deterministic state testing is very good at detecting and reproducing resource allocation failures (e.g., memory leaks and deadlocks).

3.4 Summary and Open Problems

There are two schools of thought for load design: (1) Designing loads, which mimic realistic usage; and (2) Designing loads, which are likely to trigger functional and non-functional failures. Realistic Load Design techniques are more general, but the resulting loads can take a long time to execute. Results of a load test are harder to analyze (due to the large volume of data). On the contrary, Fault-Inducing Load Design techniques are more narrowly focused on a few objectives (i.e., you will not detect unexpected problems), but the test duration is usually deterministic and shorter. The test results are usually easier to analyze.

However, a few issues are still not explored thoroughly:

- *Optimal test duration for the realistic load design*. One unanswered question among all the realistic load design techniques is how to identify the optimal test duration, which is the shortest test duration while still covering all the test objectives. This problem is very similar as determining the optimal simulation duration for the discrete event simulation experiments. Recently, there have been works [121], [122] proposed to leverage statistical techniques to limit the duration of the simulation runs by determining the number of sample observations required to reach certain accuracy in the output metrics. Similar techniques may be used to help determine the optimal test duration of the realistic load design.
- *Benchmarking & empirical studies of the effectiveness of various techniques*. Among the load design techniques, the effectiveness of these techniques, in terms of scale and coverage, is not clear. In large-scale industrial systems, which are not web-based systems, can we still apply techniques like Stochastic Form-oriented Models? A benchmark suite (like the Siemens benchmark suite for functional bug detection [123]) is needed to systematical evaluate the scale and coverage of these techniques.
- *Test coverage metrics*. Unlike functional testing suites, which have various metrics (e.g., code coverage) to measure the test coverage. There are few load testing coverage metrics other than the "Probability Mass Coverage" metric, which is proposed by

TABLE 4
Load Execution Techniques

	Load Test Execution Approaches	**Live-user based Execution**	**Driver based Execution**	**Emulation based Execution**
Aspect 1. Setup				
Setup Activities	**System Deployment**	System installation and configuration in the field/field-like/lab environment	System installation and configurations in the field/field-like/lab environment	System deployment on the special platforms
	Test Execution Setup	Tester Recruitment and Training, Test Environment Configurations	Load Driver Installation and Configurations, Test Environment Configurations	Load Driver Installation and Configurations
Aspect 2. Load Generation and Terminations				
Options for Load Generation and Termination	**Static Configurations**	✓	✓	✓
	Dynamic	x	✓	x
	Deterministic	x	x	✓
Aspect 3. Test Monitoring and Analysis				
Types of System Behavior Data	**Functional Problems**	✓	✓	✓
	Execution Logs	✓	✓	✓
	Performance Metrics	✓	✓	x
	System Snapshots	x	✓	✓

Avritzer et al. [10], [21], [22], [23]. This metric only captures the workload coverage in terms of aggregate workload. We need more metrics to capture other aspects of the system behavior. For example, we need new metrics to capture the coverage of different types of users (a.k.a., use-case based loads).

- *Testing loads evolution and maintenance*. There is no existing work aimed at maintaining and evolving the resulting loads. Below we provide two examples where the evolution of the load is likely to play an important role:
  1) *Realistic loads:* As users get more familiar with the system, usage patterns are likely to change. How much change would merit an update to a realistic-based testing loads?
  2) *Fault-inducing loads:* As the system evolve over time, can we improve the model building of fault-inducing loads by incrementally analyzing the system internals (e.g., changed source code or changed features)?

4 Research Question 2: How Is a Load Test Executed?

Once a proper load is designed, a load test is executed. The load test execution phase consists of the following three main aspects: (1) *Setup*, which includes system deployment and test execution setup; (2) *Load Generation and Termination*, which consists of generating the load according to the configurations and terminating the load when the load test is completed; and (3) *Test Monitoring and Data Collection*, which includes recording the system behavior (e.g., execution logs and performance metrics) during execution. The recorded data is then used in the Test Analysis phase.

As shown in Table 4, there are three general approaches of load test executions:

1) *Live-user based executions*. A load test examines a SUT's behavior when the SUT is simultaneously used by many users. Therefore, one of the most intuitive load test execution approach is to execute a load test by employing a group of human testers [19], [118], [124]. Individual users (testers) are selected based on the testing requirements (e.g., locations and browsers).

   The live-user based execution approach reflects the most realistic user behaviors. In addition, this approach can obtain real user feedbacks on aspects like acceptable request performance (e.g., whether certain requests are taking too long) and functional correctness (e.g., a movie or a figure is not displaying properly). However, the live-user based execution approach cannot scale well, as the approach is limited by the number of recruited testers and the test duration [118]. Furthermore, the approach cannot explore various timing issues due to complexity of manual coordination of many testers. Finally, the load tests that are executed by the live users cannot be reproduced or repeated exactly as they occurred.

2) *Driver based executions*. To overcome the scalability issue of the live-user based approach, the driver based execution approach is introduced to automatically generate thousands or millions of concurrent requests for a long period of time. Compared to the live-user based executions, where individual testers are selected and trained, driver based executions require setup and configuration of the load drivers. Therefore, a new challenge in driver based execution is the configuration of load drivers to properly produce the load. In addition, some system behavior (e.g., the movie or image display) cannot be easily tracked, as it is hard for the load driver to judge the audio or video quality.

   Different from existing driver based surveys [125], [126], [127], which focus on comparing the capabilities of various load drivers, our survey of driver based execution focuses on the techniques used by the load drivers. Comparing the load driver techniques, as opposed to capabilities, has the following two advantages in terms of knowledge contributions: (1) *Avoid Repetitions:* Tools from different vendors can adopt similar techniques. For example, WebLoad [128] and HP LoadRunner [129] both support the store-and-replay test configuration technique. (2) *Tool Evolution:* The evolution of such load drivers is not tracked in the driver based surveys. Some tools get decommissioned over time. For example, tools like Microsoft's Web App Stress Tool surveyed in [127], no longer exist. New features (e.g., supported protocols) are constantly added into the load testing tools over time. For example, Apache JMeter [130] has recently added support for model-based testing (e.g., Markov-chain models).

There are three categories of load drivers:

a) *Benchmark Suite* is a specialized load driver, designed for one type of system. For example, LoadGen [131] is a load driver used to specifically load test the Microsoft Exchange MailServer. Benchmark suites are also used to measure and compare the performance of different versions of software and/or hardware setup (called *Benchmarking*). Practitioners specify the rate of requests as well as test duration. Such load drivers are usually customized and can only be used to load test one type of system [131], [132], [133], [134].

In comparison to benchmark suites, the following two categories of load drivers (centralized and peer-to-peer load drivers) are more generic (applicable for many systems).

b) *Centralized load drivers* refer to a single load driver, which generates the load [128], [129].

c) *Peer-to-peer load drivers* refer to a set of load drivers, which collectively generate the target testing load. Peer-to-peer load drivers usually have a controller component, which coordinates the load generation among the peer load drivers [135], [136], [137].

Centralized load drivers are better at generating targeted load, as there is only one single load driver to control the traffic. Peer-to-peer load drivers can generate larger scale load (more scalable), as centralized load drivers are limited by processing and storage capabilities of a single machine.

3) *Emulation based executions*. The previous two load test execution (live-user based and driver based execution) approaches require a fully functional system. Moreover, they conduct load testing in the field or in a field-like environment. The techniques that use the emulation based load test execution approach conduct the load testing on special platforms. In this survey, we focus on two types of special platforms:

a) *Special platforms enabling early and continuous examination of system behavior under load*. In the development of large distributed software systems (e.g., service-oriented systems), many components like the application-level entities and the infrastructure-level entities are developed and validated during different phases of the software lifecycle. This development process creates *serialized-phasing* problem, as the end-to-end functional and quality-of-service aspects cannot be evaluated until late in the software life cycle (e.g., at the system integration time) [39], [40], [41], [42], [138]. Emulation based execution can emulate parts of the system that are not readily available. Such execution techniques can be used to examine the system's functional and non-functional behavior under load throughout the software development lifecycle, even before the system is completely developed.

b) *Special platforms enabling deterministic execution*. Reporting and reproducing problems like deadlocks or high response time are much easier on these special platforms, as these platforms can provide fine-grained controls on method and thread inter-leavings. When problems occur, such platforms can provide more insights on the exact system state [34].

Live-user based and driver based executions require deploying the SUT and running the test in the field or field-like environment. Both approaches need to face the challenge of setting up realistic test environment (e.g., with proper network latency mimicking distributed locations). Running the SUT on special platforms avoids such complications. However, emulation based executions usually focus on a few test objectives (e.g., functional problems under load), which are not general purposes like the live-user based and driver based executions. In addition, like driver based executions, emulation based executions use load drivers to automatically generate the testing load.

Among the three main aspects of the load test execution phase, Table 4 outlines the similarities and differences among the aforementioned three load test execution approaches. For example, there are two distinct setup activities in the Setup aspect: System Deployment and Test Execution Setup. Some setup activities would contain different aspects for the three test execution approaches (e.g., during the test execution setup activity). Some other activities would be similar (e.g., the system deployment activity is the same for live-user and driver based executions).

In the next three sections, we compare and contrast the different techniques applied in the three aspects of the load execution phase: Section 4.1 explains the setup techniques, Section 4.2 discusses the load generation and termination techniques. Section 4.3 describes the test monitoring and data collection techniques. Section 4.4 summaries the load test execution techniques and lists some open problems.

4.1 Setup

As shown in Table 4, there are two setup activities in the Setup aspect:

- *System deployment* refers to deploying the SUT in the proper test environment and making the SUT operational. Examples can include installing the SUT and configuring the associated third party components (e.g., the mail server and the database server).

- *Test execution setup* refers to setting up and configuring the load testing tools (for driver based and emulation based executions), or recruiting and training testers (for live-user based executions) and configuring the test environment to reflect the field environment (e.g., increasing network latency for long distance communication).

4.1.1 System Deployment

The system deployment process is the same for the live-user based and driver based executions, but different from the emulation based executions.

- *System installation and configuration for the live-user based and driver based executions*. For live-user based and driver based executions, it is recommended to perform the load testing on the actual field environment, although the testing time can be limited and there could be high cost associated [99]. However, in many cases, load tests are conducted in a lab environment due to accessibility and cost concerns, as it is often difficult and costly to access the actual production environment) [44], [99], [118], [139]. This lab environment can be built from the dedicated computing hardware purchased in-house [118], [139] or by renting the readily available cloud infrastructures (Testing-as-a-Service) [140], [141]. The system deployed in the lab could behave differently compared to the field environment, due to issues like unexpected resource fluctuations in the cloud environment [142]. Hence, extra efforts are required to configure the lab environment to reflect the most relevant field characteristics.

  The SUT and its associated components (e.g., database and mail servers) are deployed in a field-like setting. One of the important aspects mentioned in the load testing literature is creating realistic databases, which have a size and structure similar to the field setting. It would be ideal to have a copy of the field database. However, sometimes no such data is available or the field database cannot be directly used due to security or privacy concerns. There are two proposed techniques to create field-like test databases: (1) importing raw data, which shares the same characteristics (e.g., size and structure) as the field data [31]; (2) sanitizating the field database so that certain sensitive information (e.g., customer information) is removed or anonymized [104], [143], [144].
- *System deployment for the emulation based executions*. For the emulation based executions, the SUT needs to be deployed on the special platforms, in which the load test is to be executed. The deployment techniques for the two types of special platforms mentioned above are different:
  - *Automated code generation for the incomplete system components*. The automated code generation for the incomplete system components is achieved using model-driven engineering platforms. Rather than implementing the actual system components via programming, developers can work at a higher level of abstraction in a model-driven engineering setup (e.g., using domain-specific modeling languages or visual representations). Concrete code artifacts and system configurations are generated based on the model interpreter [39], [40], [41], [42], [138] or the code factory [145]. The overall system is implemented using a model-based engineering framework in Domain-specific modeling languages. For the components, which are not available yet, the framework interpreter will automatically generate mock objects (method stubs) based on the model specifications. These mock objects, which conform to the interface of the actual components, emulate the actual component functionality. In order to support a new environment (e.g., middleware or operating system), the model interpreter needs to be adapted for various middleware or operating systems, but no change to the upper level model specifications is required.
  - *Special profiling and scheduling platform*. In order to provide more detailed information on the SUT's state when a problem occurs (e.g., deadlocks or racing), special platforms (e.g., the CHESS platform [34]), which control the inter-leaving of threads are used. The SUT needs to be run under a development IDE (Microsoft Visual Studio) with a specific scheduling in CHESS. In this way, the CHESS scheduler, rather than the operating system, can control the inter-leaving of threads.

4.1.2 Test Execution Setup

The test execution setup activity includes two parts: (1) setting up and configuring the test components: testers (for live-user based executions) or load drivers (for driver based and emulation based executions); and (2) configuring the test environment.

Setting up and configuring the test components. Depending on the execution approaches, the test components for setup and configuration are different:

- *Tester recruitment, setup and training (live-user based executions)*. For live-user based executions, the three main steps involved in the test execution setup and configuration aspects [19], [118] are:
  1) *Tester recruitment*. Testers are hired to perform load tests. There are specific criteria to select live users depending on the testing objectives and type of system. For example, for web-applications, individual users are picked based on factors like geographical locations, languages, operating systems and browsers;
  2) *Tester setup*. Necessary procedures are carried out to enable testers to access the SUTs (e.g., network permission, account permission, monitoring and data recording software installation);

3) *Tester training*. The selected testers are trained to be familiar with the SUT and their testing scenarios.

- *Load driver deployment (driver based and emulation based executions)*. Deploying the load drivers involves the installation and configuration of load drivers:

1) *Installation of load drivers*. The load drivers are usually installed on different machines from the SUT to avoid confounding of measurements and resource usage. The machines which have load drivers installed should have enough computing resources such that they do not saturate during a load test.

   The installation of load drivers is usually straight-forward [128], [129], except for peer-to-peer load drivers. Dumitrescu et al. [135] implement a framework to automatically push the peer load drivers to different machines for load testing Grid systems. The framework picks one machine in the Grid to act as a controller. The controller pushes the peer load driver to other machines, which are responsible for requesting web services under test.

2) *Configuration of load drivers*. The configuration of load drivers is the process of encoding the load as inputs, which the load drivers can understand. There are currently four general load driver configuration techniques:

   a) *Simple GUI configuration*. Some load drivers (especially the benchmark suites like [131]) provide a simple graphical user interface for load test practitioners to specify the rate of the requests as well as test durations.

   b) *Programable configuration*. Many of the general purpose load drivers let load test practitioners encode the testing load using programming languages. The choice of programming languages varies between load drivers. For example, the language could be generic programming languages like C++ [129], Javascript [128], Java [146] or XML [147]; or domain specific languages, which enable easy specifications of test environment components like the setup/configration of database, network and storage [148] and or for specialized systems (e.g., TTCN-3 for telecommunication systems [28]).

   c) *Store-and-replay configuration*. Rather than directly encoding the load via coding, many load drivers support store-and-replay to reduce the programming efforts. Store-and-replay load driver configuration techniques are used in web-based applications [102], [128], [129], [149] and distributed telecommunication applications [150], [151]. This configuration technique consists of the following three steps:

      i) *The storing phase*: During the storing phase, load test practitioners perform a sequence of actions for each scenario. For example, in a web-based system, a user would first login to the system, browse a few catalogs then logout. A probe, which is included in the load drivers, is used to capture all incoming and outgoing data. For example, all HTTP requests can be captured by either implementing a probe at the client browser side (e.g., browser proxy in WebLoad [128], [129]) or at the network packet level using a packet analyzer like Wireshark [152]. The recorded scenarios are encoded in load-driver specific programming languages (e.g., C+ + [129] and Javascript [128]).

         Rich Internet applications (RIA) dynamically update parts of the webpage based on the user actions. Therefore, the user action sequences cannot be easily used in record-and-replay via URL editing. Instead, The store-and-replay is achieved via using GUI automation tools like Selenium [153] to record user actions instead.

      ii) *The editing phase:* The recorded data needs to be edited and customized by load test practitioners in order to be properly executed by the load driver. The stored data is usually edited to remove runtime-specific values (e.g., session IDs and user IDs).

      iii) *The replaying phase*: Once load test practitioners finish editing, they need to identify the replay rates of these scenarios, the delay between individual requests and the test duration.

   d) *Model configuration*. Section 3.1.2 explains realistic load design techniques via usage models. There are two approaches to translate the usage models into load driver inputs: on one hand, many load drivers can directly take usage models as their inputs. On the other hand, research works have been proposed to automatically generate load driver configuration code based on usage models.

i) *Readily supported models*: Test cases formulated in Markov chain can be directly used in load test execution tools like LoadRunner [129] and Apache JMeter [130] (through plugin) or research tools like [154].

ii) *Automated generation of load driver configuration code*: Many techniques have been proposed to automatically generate load driver configuration code from usage models. LoadRunner scripts can be automatically generated from UML diagrams (e.g., activity diagrams and sequence diagrams) [100], [101]. The Stochastic Form Charts can be automatically encoded into JMeter scripts [114], [115].

Configuring the Test Environment. As mentioned above, live-user based and driver based executions usually take place in a lab environment. Extra care is needed to configure the test environment to be as realistic as possible.

First, it is important to understand the implication of the hardware platforms. Netto et al. [155] and White and Pilbeam [156] evaluate the stability of the generated load under virtualized environments (e.g., virtual machines). They find that the system throughput sometimes might not produce stable load on virtual machines. Second, additional operating system configurations might need to be tuned. For example, Kim [157] reports that extra settings need to be specified in Windows platforms in order to generate hundreds or millions of concurrent connections. Last, it is crucial to make network behavior as realistic as possible. The realism of the network is covered in two aspects:

1) *Network latency*. Many load-driver based test execution techniques are conducted within a local area network, where packets are delivered swiftly and reliably. The case of no/little packet latency is usually not applicable in the field, as packets may be delayed, dropped or corrupted. IP Network Emulator Tools like Shunra [158], [159] are used in load testing to create a realistic load testing network environment [27].
2) *Network spoofing*. Routers sometimes attempt to optimize the overall network throughput by caching the source and destination. If the requests come from the same IP address, the network latency measure won't be as realistic. In addition, some systems perform traffic controls based on requests from different network addresses (IP addresses) for purposes like guarding against denial of service (DoS) attacks or providing different quality of services. *IP Spoofing* in a load test refers to the practice of generating different IP addresses for workload requests coming from different simulated users. IP Spoofing is needed to properly load test some web-based systems using the driver based executions, as these systems usually deny large volume of requests from the same IP addresses to protect against the DoS attacks. IP spoofing is usually configured in supported load drivers (e.g., [129]).

4.2 Load Generation and Termination

This section covers three categories of load generation and termination techniques: manual load generation and termination techniques (Section 4.2.1), load generation and termination based on static configurations (Section 4.2.2), and load generation and termination techniques based on dynamic system feedback (Section 4.2.3).

4.2.1 Manual Load Generation and (Timer-Based) Termination Techniques

Each user repeatedly conducts a sequence of actions over a fixed period of time. Sometimes, actions among different live users need to be coordinated in order to reach the desired load.

4.2.2 Static-Configuration-Based Load Generation and Termination Techniques

Each load driver has a controller component to generate the specified load based on the configurations [104], [128], [129]. If the load drivers are installed on multiple machines, the controller needs to send messages among distributed components to coordinate among the load drivers to generate the desired load [135].

Each specific request is either generated based on a random number during runtime (e.g., 10 percent of the time user A is browsing) [104], [129] or based on a specific predefined schedule (e.g., during the first five minutes, user B is browsing) [52], [53].

There are four types of load termination techniques based on pre-defined static configurations. The first three techniques (continuous, timer-driven and counter-driven) exist in many existing load drivers [151]. The fourth technique (statistic-driven) was recently introduced [92], [159] to ensure the validity or accuracy of the data collected.

1) *Continuous*: A load test runs continuously until the load test practitioners manually stop it;
2) *Timer-driven*: A load test runs for a pre-specified test duration then stops;
3) *Counter-driven*: A load test runs continuously until a pre-specified number of requests have been processed or sent; and
4) *Statistic-driven:* A load test is terminated once the performance metrics of interest (e.g., response time, CPU and memory) are statistically stable. This means the metrics of interest yield high confidence interval to estimate such value or have small standard deviations among the collected data points [92], [159].

4.2.3 Dynamic-Feedback-Based Load Generation and Termination Techniques

Rather than generating and terminating a load test based on static configurations, techniques have been proposed to dynamically steer the load based on the system feedback [11], [17], [24], [25], [160].

Depending on the load testing objectives, the definition of important inputs can vary. For example, one goal is to detect memory leaks [17]. Thus, input parameters that significantly impact the system memory usage, are considered as important parameters. Other goals can be to find/verify the maximum number of users that the SUT can support before the response time degrades [17] or to locate software performance bottleneck [24], [25], [160]. Thus, important inputs are the ones that significantly impact the testing objectives (e.g., performance objectives like the response time or throughput). There are three proposed techniques to locate the important inputs.

1) *System identification technique*. Bayan and Cangussu calculate the important inputs using the System Identification Technique [161], [162]. The general idea is as follows: the metric mentioned in the objectives is considered as the output variable (e.g., memory usage or response time). Different combinations of input parameters lead to different values in

the output variable. A series of random testing runs, which measure the system performance using randomly generated inputs, would create a set of linear equations with the output variable on one side and various combinations of input variables on the other side. Thus, locating the resource impacting inputs is equivalent to solving these linear equations and identifying the inputs, which are large (i.e., sensitive to the resources of interest).

2) *Analytical queuing modeling*. Compared with the System Identification Technique, which calculates the important inputs before load test execution starts, Branal et al. dynamically model the SUT using a two-layer queuing model and use analytical techniques to find the workload mixes that change the bottlenecks in the SUT. Branal et al. iteratively tune the analytical queuing model based on the system performance metrics (e.g., CPU, disk and memory). Through iteratively driving load, their model gradually narrows down the bottleneck/important inputs.

3) *Machine learning technique*. Similar to [161], [162], Grechanik et al. first apply random testing to monitor the system behavior with respect to a set of randomly selected inputs. Then they apply machine learning techniques to derive performance rules, which describe the characteristics of user inputs causing bad performance (e.g., long response time). The load testing is conducted adaptively, so that only new inputs are passed into the SUT. During the adaptive load testing process, execution traces (method entry/exit), software performance metrics (e.g., response time) are recorded and the performance rules are re-learned. The adaptive load testing is stopped when there are no new performance rules discovered.

Once these important inputs are identified, the load driver automatically generates the target load to detect memory leaks [17], to verify system performance requirements [11], or to identify software bottlenecks [24], [25], [160].

4.2.4 Deterministic Load Generation and Termination Techniques

Even though all of these load test execution techniques manage to inject many concurrent requests into the SUT, none of those techniques can guarantee to explore all the possible inter-leavings of threads and timing of asynchronous events. Such system state information is important, as some thread inter-leaving events could lead to hard to catch and reproduce problems like deadlocks or racing conditions.

As we mentioned in the beginning of this section, the CHESS platform [34] can be used to deterministically execute a test based on all the possible event inter-leavings. The deterministic inter-leaving execution is achieved by the scheduling component, as the actual scheduling during the test execution is controlled by the tool scheduler rather than the OS scheduler. The CHESS scheduler understands the semantics of all non-deterministic APIs and provides an alternative implementation of these APIs. By picking different threads to block at different execution points, the CHESS scheduler is able to deterministically explore all the possible inter-leavings of task executions. The test stops when the scheduler explores all the task inter-leavings. During this process, the CHESS platform automatically reports when there is a deadlock or race conditions, along with the exact execution context (e.g., thread interleaving and events).

4.3 Test Monitoring and Data Collection

The system behavior under load is monitored and recorded during the course of the load test execution. There is a tradeoff between the level of monitoring details and monitoring overhead. Detailed monitoring has a huge performance overhead, which may slow down the system execution and may even alter the system behavior [163]. Therefore, probing techniques for load testing are usually light weight and are intended to impose minimal overhead to the overall system.

In general, there are four categories of collected data in the research literature: Metrics, Execution Logs, Functional Failures, and System Snapshots.

4.3.1 Monitoring and Collecting Metrics

Metrics are tracked by recruited testers in the live-user based executions [19], [164], by load drivers in the driver based and emulation based executions [13], [15], [49], [50], [128], [129], [165], or by light weight system monitoring tools like PerfMon [166], pidstats [164], Munin [167], and SNMP MIBs [168].

In general, there are two types of metrics that are monitored and collected during the course of the load test execution phase: Throughput Metrics (“Number of Pass/Fail Requests”) and Performance Metrics (“End-to-End Response Time” and “Resource Usage Metrics”).

1) *Number of passed and failed requests*. Once the load is terminated, the number of passed and failed requests are collected from live users. This metric can either be recorded periodically (the number of pass and fail requests at this interval) or recorded once at the end of the load test (the total number of pass and failed requests).

2) *End-to-end response time*. The end-to-end response time (or just response time) is the time that it takes to complete one individual request.

3) *Resource usage metrics*. System resource usage metrics like CPU, memory, disk and network usage, are collected for the system under load. These resource usage metrics are usually collected and recorded at a fixed time interval. Similar to the end-to-end metrics, depending on the specifications, the recorded data can either be aggregated values or a sampled value at that particular time instance. System resource usage metrics can either be collected through system monitoring tools like PerfMon in Windows or pidstats in Unix/Linux. Such resource usage metrics are usually collected both for the SUT and its associated components (e.g., databases and mail servers).

Emulation-based test executions typically do not track these metrics, as the systems are deployed on specialized platforms which are not reflective of the actual field behavior.

4.3.2 Instrumenting and Collecting Execution Logs

Execution logs are generated by the instrumentation of code that developers insert into the source code. Execution logs record the runtime behavior of the system under test. However, excessive instrumentation is not recommended, as contention for outputting the logs could slow down the application under test [169]. There are three types of instrumentation mechanisms: (1) ad-hoc debug statements, like printf or System.out, (2) general instrumentation frameworks, like Log4j [170], and (3) through specialized instrumentation frameworks like Application Response Measurement (ARM) [171]:

1) *Ad-hoc logging*: The ad-hoc logging mechanism is the most commonly used, as developers insert output statements (e.g., printf or System.out) into the source code for debugging purposes [39]. However, extra care is required to (1) minimize the amount of information generated, and to (2) to make sure that the statements are not garbled as multiple logging threads attempt to write to the same file concurrently.
2) *General instrumentation framework*: General instrumentation frameworks, like Log4j [170], address the two limitations in the ad-hoc mechanism. The instrumentation framework provides a platform to support thread-safe logging and fine-grained control of information. *Thread-safe logging* makes sure that each logging thread serially accesses the single log file for multi-threaded systems. *Fine-grained logging control* enables developers to specify logging at various levels. For example, there can be many levels of logging suited for various purposes, like information level logs for monitoring and legal compliances [172], and debug level logs for debugging purposes. During load tests and actual field deployments, only higher level logging (e.g., at the information level) is generated to minimize overhead.
3) *Specialized instrumentation framework*: Specialized instrumentation frameworks like ARM [171] can facilitate the process of gathering performance information from running programs.

4.3.3 Monitoring and Collecting Functional Failures

Live-user based and emulation based executions record functional problems, whenever the failure occurs. For each request that a live user executes, he/she records whether the request has completed successfully. If not, he/she will note the problem areas (e.g., flash content is not displayed properly [19]).

4.3.4 Monitoring System Behavior and Collecting System Snapshots

Rather than capturing information throughout the course of the load test, Bertolino et al. [165] propose a technique that captures a snapshot of the entire test environment as well as the system state when a problem arises. Whenever the SUT's overall QoS is below some threshold, all network requests as well as snapshot of the system state are saved. This snapshot can be replayed later for debugging purposes. For the deterministic emulation based execution (e.g., in the case of the CHESS platform), the detailed system state is recorded when the deadlock or race conditions occur.

4.4 Summary and Open Problems

There are three general load test execution approaches: (1) the live-user based executions, where recruited testers manually generate the testing load; (2) the driver based executions, where the testing load is automatically generated; and (3) the emulation based executions, where the SUT is executed on top of special platforms. Live-user based executions provide the most realistic feedback on the system behavior, but suffer from scalability issues. Driver based executions can scale to large testing load and test durations, but require substantial efforts to deploy and configure the load drives for the targeted testing load. Emulation based executions provide special capacities over the other two execution approaches: (1) early examination of system behavior before SUT is fully implemented, (2) easy detection and reporting of load problems. However, emulation based execution techniques can only focus on a small subset of the load testing objectives.

Here, we list two open problems, which are still not explored thoroughly:

- *Encoding testing loads into testing tools*. It is not straight-forward to translate the designed load into inputs used by load drivers. For example, the load resulted from hybrid load optimization techniques [53] is in the form of traces. Therefore, load drivers need to be modified to take these traces as inputs and replay the exact order of these sequences. However, if the size of traces becomes large, the load driver might not be able to handle traces. Similarity, testing load derived from deterministic state testing [10], [85] is not easily realized in existing load drivers, either.
- *System monitoring details and load testing analysis*. On one hand, it is important to minimize the system monitoring overhead during the execution of a load test. On the other hand, the recorded data might not be sufficient (or straight-forward) for load testing analysis. For example, recorded data (e.g., metrics and logs) can be too large to be examined manually for problems. Additional work is needed to find proper system monitoring data suited for load testing.

5 Research Question 3: How Is the Result of a Load Test Analyzed?

During the load test execution phase, the system behavior (e.g., logs and metrics) is recorded. Such data must be analyzed to decide whether the SUT has met the test objectives. Different types of data and analysis techniques are needed to validate different test objectives.

As discussed in Section 4.3, there are four categories of system behavior data: metrics, execution logs, functional failures and system snapshots. All of the research literature focuses on the analysis and reporting techniques that are used for working with metrics and execution logs. (It is relatively straight-forward to handle the functional failure data by reporting them to the development team, and there is no further discussion on how to analyze system snapshots [165]).

There are three categories of load testing analysis approaches:

1) *Verifying against threshold values*. Some system requirements under load (especially non-functional requirements) are defined using threshold values. One example is the system resource requirements. The CPU and memory usage cannot be too high during the course of a load test, otherwise the request processing can hang and system performance can be unstable [173]. Another example is the reliability requirement for safety critical and telecommunication systems [10], [85]. The reliability requirements are usually specified as "three-nines" or "five-nines", which means the system reliability cannot be lower than 99.9 percent (for "three-nines") and 99.999 percent (for "five-nines"). The most intuitive load test analysis technique is to summarize the system behavior into one number and verify this number against a threshold. The usual output for such analysis is simply pass/fail.
2) *Detecting known types of problems*. Another general category of load test analysis is examining the system behavior to locate patterns of known problems; as some problems are buried in the data and cannot be found based on threshold values, but can be spotted by known patterns. One example of such analysis approach is to check the memory growth trend over time for memory leaks. The usual output for such analysis is a list of detected problems.
3) *Detecting anomalous behavior*. Unfortunately, not all problems can be specified using patterns and certainly not all problems have been detected previously. In addition, the volume of recorded system behavior is too large for manual examination. Therefore, automated techniques have been proposed to systematically analyze the system behavior to uncover anomalous behavior. These techniques automatically derive "normal/expected behavior" and flag "anomalous behavior" from the data. However, the accuracy of such techniques might not be as high as the above two approaches, as the "anomalous behavior" are merely hints of potential problems under load. The output for such analysis is usually the anomalous behavior and some reasoning/diagnosis on the potential problematic behavior.

All three aforementioned techniques can analyze different categories of data to verify a range of objectives (detecting functional problems and non-functional problems). These load test analysis techniques can be used individually or together based on the types of data available and the available time. For example, if time permits, load testing practitioners can verify against known requirements based on thresholds, locate problems based on specific patterns and run the automated anomaly detection techniques just to check if there are any more problems. We categorize the various load test analysis techniques into the following six dimensions as shown in Table 5.

- *Approaches* refer to one of the above three load test analysis approaches.
- *Techniques* refer to the load test analysis technique like memory leak detection.
- *Data* refers to the types of system behavior data that the test analysis technique can analyze. Examples are execution logs and performance metrics like response time.
- *Test objectives* refer to the goal or goals of load test objectives (e.g., detecting performance problems), which the test analysis technique achieves.
- *Reported results* refer to the types of reported outcomes, which can simply be pass/fail or detailed problem diagnoses.
- *References* refer to the list of literatures, which propose each technique.

This section is organized as follows: The next three sections describe the three categories of load testing analysis techniques respectively: Section 5.1 explains the techniques of verifying load test results against threshold values, Section 5.2 describes the techniques of detecting known types of problems, and Section 5.3 explains the techniques of automated anomaly detection and diagnosis. Section 5.4 summarizes the load test analysis techniques and highlights some open problems.

5.1 Verifying against Threshold Values

The threshold-based test analysis approach can be further broken down into three techniques based on the availability of the data and threshold values.

5.1.1 Straight-Forward Comparison

When the data is available and the threshold requirement is clearly defined, load testing practitioners can perform a straight-forward comparison between the data and the threshold values. One example is throughput analysis. Throughput, which is the rate of successful requests completed, can be used to compare against the load to validate whether the SUT's functionality scales under load [174], [175], [176].

5.1.2 Comparing against Processed Data

If the system resources, like CPU and memory utilization are too high, the system performance may not be stable [173] and user experience could degrade (e.g., slow response time) [27], [43], [93], [95], [191].

There can be many formats of system behavior. One example is resource usage data, which is sampled at a fixed interval. Another example is the end-to-end response time, which is recorded as response time for each individual request. These types of data need to be processed before comparing against threshold values. On one hand, as Bondi points out [45], system resources may fluctuate during the startup time for warmup and cooldown periods. Hence, it is important to only focus on the system behavior once the system reaches a stabilized state. On the other hand, a proper data summarization technique is needed to describe these many data instances into one number. There are three types of data summarization techniques proposed in the

TABLE 5
Load Test Analysis Techniques

Approaches	Techniques	Data	Test Objectives	Reported Results	References
Verifying Against Threshold Values	Straight-forward Comparison	Performance metrics	Detecting violations in performance and scalability requirements	Pass/Fail	[174], [175], [176]
	Comparing Against Processed Data (Max, median or 90-percentile values)	Periodic sampling metrics	Detecting violations in performance requirements		[15], [38], [39], [40], [41], [42], [135], [173], [174], [177], [178]
	Comparing Against Derived (Threshold and/or target) Data	Number of pass/fail requests, past performance metrics	Detecting violations in performance and reliability requirements		[15], [16], [26], [179]
Detecting Known Types of Problems	Detecting Memory Leaks	Memory usage metrics	Detecting load-related functional problems	Pass/Fail	[9], [17], [45]
	Locating Error Keywords	Execution logs	Detecting functional problems	Error log lines and error types	[180]
	Detecting Deadlocks	CPU	Detecting load-related functional problems and violations in scalability requirements	Pass/Fail	[9]
	Detecting Unhealthy System States	CPU, Response Time and Workload	Detecting load-related functional problems and violations in performance requirements	Pass/Fail	[9]
	Detecting Throughput Problems	Throughput, response time metrics	Detecting load-related functional problems and violations in scalability requirements	Pass/Fail	[159]
Detecting Anomalous Behavior	Detecting Anomalous Behavior using Performance Metrics	Performance metrics	Detecting performance problems	Anomalous performance metrics	[86], [181], [182], [183], [184], [185], [186], [187], [188], [189]
	Detecting Anomalous Behavior using Execution Logs	Execution logs	Detecting functional and performance problems	Log sequences with anomalous functional or performance behaviors	[87], [88]
	Detecting Anomalous Behavior using Execution Logs and Performance Metrics	Execution logs and performance metrics	Detecting memory-related problems	Potential problematic log lines causing memory-related problems	[190]

literature. We use response time analysis as an example to describe the proposed data summarization techniques:

1) *Maximum values*. For online distributed multi-media systems, if any video and audio packets are out of sync or not delivered in time, it is considered a failure [38]. Therefore, the inability of the end-to-end response time to meet a specific threshold (e.g., video buffering period) is considered as a failure.
2) *Average or median values*. The average or median response time summarizes the majority of the response times during the load test and is used to evaluate the overall system performance under load [15], [39], [40], [41], [42], [135], [174].
3) *90-percentile values*. Some researchers advocate that the 90-percentile response time is a better measurement than the average/median response time [173], [177], [178], as 90-percentile response time accounts for most of the peaks, while eliminating the outliers.

5.1.3 Comparing against Derived Data

In some cases, either the data (e.g., the reliability) to compare or the threshold value is not directly available. Extra steps need to be taken to derive this data before analysis.

- *Deriving thresholds*. Some other threshold values for non-functional requirements are informally defined. One example is the "no-worse-than-before" principle when verifying the overall system performance. The "no-worse-than-before" principle states that the average response time (system performance requirements) for the current version should be at least as good as prior versions [26].
- *Deriving target data*. There are two methods for deriving the target data to be analyzed:
  - *Through extrapolation.* As mentioned in Section 3.3.2, due to time or cost limitations, sometimes it is not possible to run the targeted load, but we might run tests with lower load

levels (same workload mix but different intensity). Based on the performance of these lower workload intensity level tests, load test practitioners can extrapolate the performance metrics at the targeted load [15], [16], [179]. If certain resource metrics are higher than the hardware limits (e.g., requires more memory than provided or CPU utilization is greater than 100 percent) based on the extrapolation, scalability problems are noted.

- *Through Bayesian network.* Software reliability is defined as the probability of failure-free operation for a period of time, under certain conditions. Mission critical systems usually have very strict reliability requirements. Avritzer et al. [23], [85] use the Bayesian Network to estimate the system reliability from the load test data. Avritzer et al. use the failure probability of each type of load (workload mix and workload intensity) and the likelihood of these types of load occurring in the field. Load test practitioners can then use such reliability estimates to track the quality of the SUT across various builds and decide whether the SUT is ready for release.

5.2 Detecting Known Types of Problems

There are five known load related problems, which can be analyzed using patterns: detection of memory leaks (Section 5.2.1), locating error keywords (Section 5.2.2), detecting deadlocks (Section 5.2.3), detecting unhealthy system states (Section 5.2.4), and detecting throughput problems using queuing theory (Section 5.2.5).

5.2.1 Detecting Memory Leaks

Memory leaks can cause long running systems to crash. Memory leak problems can be detected if there is an upward trend of the memory footprint throughout the course of load testing [9], [17], [45].

5.2.2 Locating Error Keywords

Execution logs, generated by code instrumentations, provide textual descriptions of the system behavior during runtime. Compared to system resource usage data, which are structural and easy to analyze, execution logs are more difficult to analyze, but provide more in-depth knowledge.

One of the challenges of analyzing execution logs is the size of the data. At the end of a load test, the size of execution logs can be several hundred megabytes or gigabytes. Therefore, automatic log analysis techniques are needed to scan through logs to detect problems.

Load testing practitioners can search for specific keywords like “errors”, “failures”, “crash” or “restart” in the execution logs [88]. Once these log lines are found, load test practitioners need to analyze the context of the matched log lines to determine whether they indicate problems or not. One of the challenges of performing a simple keyword search is that the data is not categorized. There can be hundreds of “error” log lines belonging to several different types of errors. Jiang et al. [180] extend this approach to further categorize these log lines into various types of errors or failures. They accomplish this by first abstracting each execution log line into an execution event where the runtime data is parameterized. Then, they group these execution events by their associated keywords like “failures” or “errors”. A log summary report is then produced with a clear breakdown of the types of “failures” and “errors”, their frequencies and examples of their occurrence in the logs.

5.2.3 Detecting Deadlocks

Deadlocks can cause CPU resource to deviate from normal levels [9]. A typical pattern would be CPU resource repeatedly drops below normal levels (indicating deadlock) and returns to normal levels (indicating lock releases).

5.2.4 Detecting Unhealthy System States

Avritzer and Bondi [9] observe that under normal conditions the CPU resource has a linear relation with the workload and that the response time should be stable over time. However, when such observations (a.k.a., the linear relationship among the CPU resource, the workload and the stable response time) no longer hold, then the SUT might have performance problems (e.g., software bottlenecks or concurrency problems).

5.2.5 Detecting Throughput Problems

Mansharamani et al. [159] use Little’s Law from Queuing Theory to validate the load test results:

$$Throughput = \frac{Number\ of\ users}{Response\ Time\ +\ Average\ Think\ Time}.$$

If there is a big difference between the calculated and measured throughput, there could be failure in the transactions or load variations (e.g., during warm up or cool down) or load generation errors (e.g., load generation machines cannot keep up with the specified loads).

5.3 Detecting Anomalous Behavior

Depending on the types of data available, the automated anomaly detection approach can be further broken down into two groups of techniques: (1) techniques based on metrics; and (2) techniques based on execution logs.

5.3.1 Anomaly Detection Using Performance Metrics

The proposed techniques based on metric-based anomaly detection are focused on analyzing the resource usage data. There are six techniques proposed to derive the “expected/normal” behavior and flag “anomalous” behavior based on resource usage data:

1) **Deriving and Comparing Clusters**

   As noted by Georges et al. [163], [192], it is important to execute the same tests multiple times to gain a better view of the system performance due to issues like system warmup and memory layouts. Bulej et al. [56] propose the use of statistical techniques to detect performance regressions (performance degradations in the context of regression testing). Bulej et al. repeatedly execute the same tests multiple times. Then, they group the response time for each request into clusters and compare the response time distributions cluster-by-cluster. They have used various statistical tests (Student-t test, Kolmogorov-Smirnov Test, Wilcoxon test, Kruskal-Wallis test) to compare the response time distributions between the current release and prior releases. The results of their case studies show that these statistical tests yield similar results.

2) *Deriving clusters and finding outliers*. Rather than comparing the resulting clusters as in [56], Syer et al. [86], [189] use a hierarchical clustering technique to identify outliers, which represent threads with deviating behavior in a thread pool. A thread pool, which is a popular design pattern for large-scale software systems, contains a collection of threads available to perform the same type of computational tasks. Each thread in the thread pool performs similar tasks and should exhibit similar behavior with respect to resource usage metric, such as CPU and memory usage. Threads with performance deviations likely indicate problems, such as deadlock or memory leaks.

3) *Deriving performance ranges*. A control chart consists of three parts: a control line (center line), a lower control limit (LCL) and an upper control limit. If a point lies outside the controlled regions (between the upper and lower limits), the point is counted as a violation. Control charts are used widely in the manufacturing process to detect anomalies. Nguyen et al. [188] use control charts to flag anomalous resource usage metrics. There are several assumptions associated with applying control chart: (1) the collected data is normally distributed; (2) the testing loads should be constant or linearly correlated with the system performance metrics; and (3) the performance metrics should be independent of each other.

For each recorded resource usage metrics, Nguyen et al. derive the "expected behavior" in the form of control chart limits based on prior good tests. For tests whose loads are not constant, Nguyen et al. use linear extrapolation to transform the performance metrics data. Then current test data is overlayed on the control chart. If the examined performance metric (e.g., subsystem CPU) has a high number of violations, this metric is flagged as an anomaly and is reported to the development team for further analysis.

4) *Deriving performance rules*. Nguyen et al. [188] treat each metric separately and derive range boundary values for each of these metrics. However, in many cases the assumptions of control chart may not hold by the performance metrics data. For example, when the SUT is processing a large number of requests, the CPU usage and memory usage could be high.

Foo et al. [181] build performance rules, and flag metrics, which violate these rules. A performance rule groups a set of correlating metrics. For example, a large number of requests imply high CPU and memory usage. For all the past tests, Foo et al. first categorize each metrics into one of high/median/low categories, then derive performance rules by applying an artificial intelligence technique, called Association Rule mining. The performance rules (association rules) are derived by finding frequent co-occurred metrics. For example, if high browsing requests, high Database CPU and high web server memory footprint always appear together, *Browsing/DB CPU/Web Server Memory* form a set (called "frequent-item-set"). Based on the frequent-item-set, association rules can be formed (e.g., high browsing requests and high web server memory implies high database CPU). Metrics from the current test are matched against these rules. Metrics (e.g., low database CPU), which violate these rules, are flagged as "anomalous behavior".

5) *Deriving performance signatures*. Rather than deriving performance rules [181], Malik et al. [184], [185], [186], [187] select the most important metrics among hundreds or thousands of metrics and group these metrics into relevant groups, called "Performance Signatures". Malik et al. propose two main types of performance signature generation techniques: an unsupervised learning approach and a supervised learning approach.

If the past performance tests are not clearly labeled with pass/fail information, Malik et al. use an unsupervised learning technique, called Principal Component Analysis (PCA) [184], [185], [186], [187]. First, Malik et al. normalize all metrics into values between 0 and 1. Then PCA is applied to show the relationship between metrics. PCA groups metrics into groups, called Principle Components (PC). Each group has a value called variance, which explains the importance/relevance of the group to explain the overall data. The higher the variance values of the groups, the more relevant these groups are. Furthermore, each metric is a member of all the PCs, but the importance of the metrics within one group varies. The higher the eigenvalue of a metric within one group, the more important the metric is to the group. Malik et al. select first N Principle Components with then largest variance. Then within each Principle Component, Malik et al. select important counters by calculating pair-wise correlations between counters. These important counters forms the "Performance Signatures". The performance signatures are calculated on the past good tests and the current test, respectively. The discrepancies between the performance signatures are flagged as "Anomalous Behavior".

If the past performance tests are labeled with as pass/fail, Malik et al. recommend to use a supervised learning approach to pin-point performance problems over the aforementioned unsupervised learning approach, as the supervised learning approach yields better results. In [184], they first use a machine learning technique called the Wrapped-based attribute selection technique to pick the top N performance counters, which best characterizes the performance behavior of the SUT under load. Then they build a logistic regression model with these signature performance counters. The performance counter data from the new test is passed into a logistic regression model to identify whether they are anomalous or not.

6) *Deriving transaction profiles (TPs)*. The aforementioned five techniques uses data mining techniques to derive the expected behavior and flag the anomalously behavior, Ghaith et al. use queuing network model to derive the expected behavior, called the Transaction Profiles [182], [183]. A TP represents the service demands on all resources when processing a particular transaction. For example, in a web-based application, the TP for a single "Browsing" transaction would be 0.1 seconds of server CPU, 0.2 seconds of server disk and 0.05 seconds of client CPU. Ideally, the performance of each transaction would be identical regardless of the system load, if the SUT does not experience any performance bottleneck. Hence, the TP would be identical for a particular transaction type regardless of the load. If the TP deviates from one release to another, the SUT might have a performance regression problem. Ghaith et al. derive TPs from the performance data on previous releases and compares again the current release. If the TPs differ, the new release might have performance regression problems.

5.3.2 Anomaly Detection Using Execution Logs

There are two proposed log-based anomaly detection techniques. One technique is focused on detecting anomalous functional behavior, the other one is focused on detecting anomalous performance (i.e., non-functional) behavior.

1) *Detecting anomalous functional behavior*. There are limitations associated with keyword-based analysis approaches described in Section 5.2.2: First, not all log lines with the keywords correspond to failures. For example, the log line, "Failure to locate item in the cache", contains the "failure" keyword, it is not an anomalous log line worthy of investigation. Second, not all log lines without such keywords are failure free. For example, the log line, "Internal message queue is full", does not contain the word failure, though it is an indication of anomalous situation that should be investigated.

   Jiang et al. [88] propose a technique that detects the anomalous execution sequences in the execution logs, instead of relying on the log keywords. The main intuition behind this work is that a load test repeatedly executes a set of scenarios over a period of time. The applications should follow the same behavior (e.g., generating the same logs) when the scenario is executed each time. As load testing is conducted after the functional tests are completed, the dominant behavior is usually the normal (i.e., correct) behavior and the minority (i.e., deviated) behaviors are likely troublesome and worth investigating. For example, the database disconnects and reconnects with the SUT intermittently throughout the test. These types of anomalous behavior should be raised for further investigation.

   Similar as in Section 5.2.2, Jiang et al. first abstract each log line into an execution event, then group these log lines into pairs (based on runtime information like session IDs or thread IDs). Then, Jiang et al. group these event pairs and flag small deviations. For example, if 99 percent of the time a lock-open event is followed by a lock-close event and 1 percent of the time lock open is followed by something else; such deviated behavior should be flagged as an "anomalous behavior".

2) *Detecting anomalous performance behavior (response time)*. As a regular load test simulates periods of peak usage and periods of off-hour usage, the same workload is usually applied across load tests, so that the results of prior load tests are used as an informal baseline and compared against the current run. If the current run has scenarios, which follow a different response time distribution than the baseline, this run is probably troublesome and worth investigating. Jiang et al. proposed an approach, which analyzes the response time extracted from the execution logs [87]. Jiang et al. recover the scenario sequences by linking the corresponding identifiers (e.g., session IDs). In this way, both the end-to-end and step-wise response times are extracted for each scenario. By comparing the distribution of end-to-end and step-wise response times, this approach reports scenarios with performance problems and pin-points performance bottlenecks within these scenarios.

5.3.3 Anomaly Detection Using Execution Logs and Performance Metrics

All the aforementioned anomaly detection techniques only examine one type of system behavior data (execution logs or performance metrics), Syer et al. [190] analyze both execution logs and performance metrics for memory-related problems. Ideally, same set of log lines (a.k.a., same workload) would lead to similar system resource usage levels (e.g., similar CPU and memory usages). Otherwise, scenarios corresponding to these log lines might lead to potential performance problems. Syer et al. first divide the logs and memory usage data into equal time intervals and combine these two types of system behavior data into profiles. Then these profiles are clustered based on the similarity of logs. Finally, outliers within these clusters are identified by the deviation of their memory footprints. Scenarios corresponding to the outlier clusters could lead to potential memory-related problems (e.g., memory leaks).

5.4 Summary and Open Problems

Depending on the types of data and test objectives, there are different load test analysis techniques that have been proposed. There are three general test analysis approaches: verifying the test data against fixed threshold values, searching through the test data for known problem patterns and automated detection of anomalous behaviors.

Below are a few open problems:

- *Can we use system monitoring techniques to analyze load test data?* Many research ideas in production system monitoring may be applicable for load testing analysis. For example, approaches (e.g., [20], [193], [194], [195]) have been proposed to build performance signatures based on the past failures, so that whenever such symptoms occur in the field, the problems can be detected and notified right away. Analogously, we can formulate our performance signature based on mining the past load testing history and use these performance signatures to detect recurrent problems in load tests. A promising research area is to explore the applicability and ease of adapting system monitoring techniques for the analysis of load tests.

- *Scalable and efficient analysis of the results of load tests*. As load tests generate large volumes of data, the load test analysis techniques need to be scalable and efficient. However, as data grows larger (e.g., bigger than one machine's hard-drive to store), many of the test analysis techniques may not scale well. It is very important to explore scalable test analysis techniques, which can automatically examine gigabytes or terabyte of system behavior data efficiently.

6 Survey Conclusion

To ensure the quality of large scale systems, load testing is required in addition to conventional functional testing procedures. Furthermore, load testing is becoming more important, as an increasing number of services are being offered in the cloud to millions of users. However, as observed by Visser [196], load testing is a difficult task requiring a great understanding of the SUT. In this paper, we have surveyed techniques that are used in the three phases of load testing: the load design phase, the load execution phase, and the load test analysis phase. We compared and contrasted these techniques and provided a few open research problems for each phase of the load testing problem.

References

[1] "Applied performance management survey," Oct. 2007.

[2] 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.

[3] Firefox download stunt sets record for quickest meltdown [Online]. Available: http://www.siliconbeat.com/2008/06/17/firefox-download-stunt-sets-record-for-quickest-meltdown/, 2015.

[4] Steve Jobs on MobileMe [Online]. Available: http://arstechnica.com/journals/apple.ars/2008/08/05/steve-jobs-on-mobi leme-the-full-e-mail

[5] S. G. Stolberg and M. D. Shear. (2013). Inside the race to rescue a health care site, and Obama [Online]. Available: http://www.nytimes.com/2013/12/01/us/politics/inside-the-race-to-rescue -a-health-site-and-obama.html

[6] M. J. Harrold, "Testing: A roadmap," in *Proc. Conf. Future Softw. Eng.*, 2000, pp. 61–72.

[7] C.-W. Ho, L. Williams, and A. I. Anton, "Improving performance requirements specification from field failure reports," in *Proc. IEEE 15th Int. Requirements Eng. Conf.*, 2007, pp. 79–88.

[8] C.-W. Ho, L. Williams, and B. Robinson, "Examining the relationships between performance requirements and 'not a problem' defect reports," in *Proc. 16th IEEE Int. Requirements Eng. Conf.*, 2008, pp. 135–144.

[9] A. Avritzer and A. B. Bondi, "Resilience assessment based on performance testing," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, Eds. Berlin, Germany: Springer, 2012, pp. 305–322.

[10] 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.

[11] M. S. Bayan and J. W. Cangussu, "Automatic feedback, control-based, stress and load testing," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 661–666.

[12] G. Gheorghiu. (2005). Performance vs. load vs. stress testing [Online]. Available: http://agiletesting.blogspot.com/2005/02/performance-vs-load-vs-stress- testing.html

[13] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. (1997, Sep.). Performance testing guidance for web applications - patterns & practices [Online]. Available: http://msdn.microsoft.com/en-us/library/bb924375.aspx

[14] B. Dillenseger, "Clif, a framework based on fractal for flexible, distributed load testing," *Ann. Telecommun.*, vol. 64, pp. 101–120, 2009.

[15] D. A. Menasce, "Load testing, benchmarking, and application performance management for the web," in *Proc. Comput. Manag. Group Conf.*, 2002, pp. 271–281.

[16] D. A. Menasce, "Load testing of web sites," *IEEE Internet Comput.*, vol. 6, no. 4, pp. 70–74, Jul./Aug. 2002.

[17] M. S. Bayan and J. W. Cangussu, "Automatic stress and load testing for embedded systems," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf.*, 2006, pp. 229–233.

[18] C.-S. D. Yang and L. L. Pollock, "Towards a structural load testing tool," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1996, pp. 201–208.

[19] uTest - Load Testing Services [Online]. Available: http://www.utest.com/load-testing, 2013.

[20] M. Acharya and V. Kommineni, "Mining health models for performance monitoring of services," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 409–420.

[21] A. Avritzer and B. B. Larson, "Load testing software using deterministic state testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1993, pp. 82–88.

[22] A. Avritzer and E. J. Weyuker, "Generating test suites for software load testing," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 1994, pp. 44–57.

[23] A. Avritzer and E. J. Weyuker, "The automatic generation of load test suites and the assessment of the resulting software," *IEEE Trans. Softw. Eng.*, vol. 21, no. 9, pp. 705–716, Sep. 1995.

[24] C. Barna, M. Litoiu, and H. Ghanbari, "Autonomic load-testing framework," in *Proc. 8th ACM Int. Conf. Autonomic Comput.*, 2011, pp. 91–100.

[25] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 872–875.

[26] F. Huebner, K. S. Meier-Hellstern, and P. Reeser, "Performance testing for IP services and systems," in *Performance Eng., State of the Art and Current Trends*. New York, NY, USA: Springer, 2001, pp. 283–299.

[27] B. Lim, J. Kim, and K. Shim, "Hierarchical load testing architecture using large scale virtual clients," in *Proc. IEEE Int. Cof. Multimedia Expo*, 2006, pp. 581–584.

[28] I. Schieferdecker, G. Din, and D. Apostolidis, "Distributed functional and load tests for web services," *Int. J. Softw. Tools for Technol. Transfer*, vol. 7, pp. 351–360, 2005.

[29] P. Zhang, S. G. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Nov. 2011, pp. 43–52.

[30] S. Abu-Nimeh, S. Nair, and M. Marchetti, "Avoiding denial of service via stress testing," in *Proc. IEEE Int. Conf. Comput. Syst. Appl.*, 2006, pp. 300–307.

[31] D. Bainbridge, I. H. Witten, S. Boddie, and J. Thompson, "Stress-testing general purpose digital library software," in *Research and Advanced Technology for Digital Libraries*. New York, NY, USA: Springer, 2009, pp. 203–214.

[32] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proc. Conf. Genetic Evolutionary Comput.*, 2005, pp. 1021–1028.

[33] C. D. Grosso, G. Antoniol, M. D. Penta, P. Galinier, and E. Merlo, "Improving network applications security: A new heuristic to generate stress testing data," in *Proc. Conf. Genetic Evolutionary Comput.*, 2005, pp. 1037–1043.

[34] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 267–280.

[35] F. I. Vokolos and E. J. Weyuker, "Performance testing of software systems," in *Proc. 1st Int. Workshop Softw. Perform.*, 1998, pp. 80–87.

[36] A. Chakravarty, "Stress testing an AI based web service: A case study," in *Proc. 7th Int. Conf. Inf. Technol.: New Generations*, Apr. 2010, pp. 1004–1008.

[37] M. Kalita and T. Bezboruah, "Investigation on performance testing and evaluation of prewebd: A .net technique for implementing web application," *IET Softw.*, vol. 5, no. 4, pp. 357–365, Aug. 2011.

[38] J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Softw. - Practice Experience*, vol. 32, no. 15, pp. 1411–1435, 2002.

[39] J. Hill, D. Schmidt, J. Edmondson, and A. Gokhale, "Tools for continuously evaluating distributed system qualities," *IEEE Softw.*, vol. 27, no. 4, pp. 65–71, Jul./Aug. 2010.

[40] J. H. Hill, "An architecture independent approach to emulating computation intensive workload for early integration testing of enterprise DRE systems," in *Proc. Confederated Int. Conf., CoopIS, DOA, IS, and ODBASE 2009 On the Move to Meaningful Internet Syst.*, 2009, pp. 744–759.

[41] J. H. Hill, D. C. Schmidt, A. A. Porter, and J. M. Slaby, "Cicuts: Combining system execution modeling tools with continuous integration environments," in *Proc. 15th Annu. IEEE Int. Conf. Workshop Eng. Comput. Based Syst.*, 2008, pp. 66–75.

[42] J. H. Hill, S. Tambe, and A. Gokhale, "Model-driven engineering for development-time qos validation of component-based software systems," in *Proc. 14th Annu. IEEE Int. Conf. Workshops Eng. Comput.-Based Syst.*, 2007, pp. 307–316.

[43] A. Avritzer, J. Kondek, D. Liu, and E. J. Weyuker, "Software performance testing based on workload characterization," in *Proc. 3rd Int. Workshop Softw. Perform.*, 2002, pp. 17–24.

[44] S. Barber, "Creating effective load models for performance testing with incomplete empirical data," in *Proc. 6th IEEE Int. Workshop Web Site Evolution*, 2004, pp. 51–59.

[45] A. B. Bondi, "Automating the analysis of load test results to assess the scalability and stability of a component," in *Proc. Comput. Meas. Group Conf.*, 2007, pp. 133–146.

[46] G. Gheorghiu. (2005). More on performance vs. load testing [Online]. Available: http://agiletesting.blogspot.com/2005/04/more-on-performance-vs-load-te sting.html

[47] D. A. Menasce and V. A. F. Almeida, *Scaling for E Business: Technologies, Models, Performance, and Capacity Planning*. Upper Saddle River, NJ, USA: Prentice-Hall, 2000.

[48] B. A. Pozin and I. V. Galakhov, "Models in performance testing," *Program. Comput. Softw.*, vol. 37, no. 1, pp. 15–25, Jan. 2011.

[49] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Proc. Int. Conf. Softw. Eng. Future Softw. Eng. Track*, 2007, pp. 171–187.

[50] D. A. Menasce, V. A. F. Almeida, R. Fonseca, and M. A. Mendes, "A methodology for workload characterization of e-commerce sites," in *Proc. 1st ACM Conf. Electron. Commerce*, 1999, pp. 119–128.

[51] G. Casale, A. Kalbasi, D. Krishnamurthy, and J. Rolia, "Automatic stress testing of multi-tier systems by dynamic bottleneck switch generation," in *Proc. 10th ACM/IFIP/USENIX Int. Conf. Middleware*, 2009, pp. 1–20.

[52] D. Krishnamurthy, J. Rolia, and S. Majumdar, "Swat: A tool for stress testing session-based web applications," in *Proc. Comput. Meas. Group Conf.*, 2003, pp. 639–649.

[53] 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.

[54] J. Zhang, S.-C. Cheung, and S. T. Chanson, "Stress testing of distributed multimedia software systems," in *Proc. IFIP TC6 WG6.1 Joint Int. Conf. Formal Description Techn. Distrib. Syst. Commun. Protocols Protocol Specification, Testing Verification*, 1999, pp. 119–133.

[55] A. F. Karr and A. A. Porter, "Distributed performance testing using statistical modeling," in *Proc. 1st Int. Workshop Adv. Model-Based Testing*, 2005, pp. 1–7.

[56] L. Bulej, T. Kalibera, and P. Tma, "Repeated results analysis for middleware regression benchmarking," *Perform. Evaluation*, vol. 60, no. 1-4, pp. 345–358, 2005.

[57] T. Kalibera, L. Bulej, and P. Tuma, "Automated detection of performance regressions: The mono experience," in *Proc. 13th IEEE Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, Sep. 27–29, 2005, pp. 183–190.

[58] J. W. Cangussu, K. Cooper, and W. E. Wong, "Reducing the number of test cases for performance evaluation of components," in *Proc. 19th Int. Conf. Softw. Eng. Knowl. Eng.*, 2007, pp. 145–150.

[59] J. W. Cangussu, K. Cooper, and W. E. Wong, "A segment based approach for the reduction of the number of test cases for performance evaluation of components," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 481–505, 2009.

[60] M. G. Stochel, M. R. Wawrowski, and J. J. Waskiel, "Adaptive agile performance modeling and testing," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf. Workshops*, 2012, pp. 446–451.

[61] M. J. Johnson, C.-W. Ho, M. E. Maximilien, and L. Williams, "Incorporating performance testing in test-driven development," *IEEE Softw.*, vol. 24, no. 3, pp. 67–73, May/Jun. 2007.

[62] A. Avritzer and E. J. Weyuker, "Deriving workloads for performance testing," *Softw. - Practice Experience*, vol. 26, no. 6, pp. 613–633, 1996.

[63] P. Csurgay and M. Malek, "Performance testing at early design phases," in *Proc. IFIP TC6 12th Int. Workshop Testing Commun. Syst.*, 1999, pp. 317–330.

[64] G. Denaro, A. Polini, and W. Emmerich, "Early performance testing of distributed software applications," in *Proc. 4th Int. Workshop Softw. Perform.*, 2004, pp. 94–103.

[65] G. Denaro, A. Polini, and W. Emmerich, "Performance testing of distributed component architectures," in *Performance Testing of Distributed Component Architectures. In Building Quality Into COTS Components: Testing and Debugging*. New York, NY, USA: Springer-Verlag, 2005.

[66] V. Garousi, "A genetic algorithm-based stress test requirements generator tool and its empirical evaluation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 778–797, Nov./Dec. 2010.

[67] V. Garousi, "Empirical analysis of a genetic algorithm-based stress test technique," in *Proc. 10th Annu. Conf. Genetic Evolutionary Comput.*, 2008, pp. 1743–1750.

[68] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware stress testing of distributed systems based on UML models," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 391–400.

[69] V. Garousi, L. C. Briand, and Y. Labiche, "Traffic-aware stress testing of distributed real-time systems based on UML models using genetic algorithms," *J. Syst. Softw.*, vol. 81, no. 2, pp. 161–185, 2008.

[70] E. Bozdag, A. Mesbah, and A. van Deursen, "Performance testing of data delivery techniques for AJAX applications," *J. Web Eng.*, vol. 8, no. 4, pp. 287–315, 2009.

[71] D. S. Hoskins, C. J. Colbourn, and D. C. Montgomery, "Software performance testing using covering arrays: Efficient screening designs with categorical factors," in *Proc. 5th Int. Workshop Softw. Perform.*, 2005, pp. 131–136.

[72] M. Sopitkamol and D. A. Menascé, "A method for evaluating the impact of software configuration parameters on e-commerce sites," in *Proc. 5th Int. Workshop Softw. Perform.*, 2005, pp. 53–64.

[73] B. Beizer, *Software System Testing and Quality Assurance*. New York, NY, USA: Van Nostrand, Mar. 1984.

[74] I. Gorton, *Essential Software Architecture*. Springer, 2000.

[75] A. Adamoli, D. Zaparanuks, M. Jovic, and M. Hauswirth, "Automated GUI performance testing," *Softw. Quality Control*, vol. 19, no. 4, pp. 801–839, Dec. 2011.

[76] D. A. Menasce and V. A. F. Almeida, *Capacity Planning for Web Services: Metrics, Models, and Methods*. Upper Saddle River, NJ, USA: Prentice-Hall, 2001.

[77] D. A. Menasce, V. A. Almeida, and L. W. Dowd, *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, 1997.

[78] S. Nejati, S. D. Alesio, M. Sabetzadeh, and L. Briand, "Modeling and analysis of cpu usage in safety-critical embedded systems to support stress testing," in *Proc. 15th Int. Conf. Model Driven Eng. Languages Syst.*, 2012, pp. 759–775.

[79] E. W. Dijkstra, "Notes on structured programming, " in, *Structured Programming*. London, U.K.: Academic, Apr. 1970.

[80] CompleteSearch DBLP [Online]. Available: http://dblp.uni-trier.de/search/publ, 2015.

[81] Google Scholar [Online]. Available: https://scholar.google.ca/, 2015.

[82] Microsoft Academic Search [Online]. Available: http://academic.research.microsoft.com/, 2015.

[83] ACM Portal [Online]. Available: http://dl.acm.org/, 2015.

[84] IEEE Explore [Online]. Available: http://ieeexplore.ieee.org/Xplore/home.jsp, 2015.

[85] A. Avritzer, F. P. Duarte, a. Rosa Maria Meri Le, E. de Souza e Silva, M. Cohen, and D. Costello, "Reliability estimation for large distributed software systems," in *Proc. Conf. Center Adv. Stud. Collaborative Res.*, 2008, pp. 157–165.

[86] M. D. Syer, B. Adams, and A. E. Hassan, "Identifying performance deviations in thread pools," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 83–92.

[87] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automated performance analysis of load tests," in *Proc. 25th IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 125–134.

[88] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora, "Automatic identification of load testing problems," in *Proc. 24th IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 307–316.

[89] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. New York, NY, USA: Wiley, Apr. 1991.

[90] M. Calzarossa and G. Serazzi, "Workload characterization: A survey," *Proc. IEEE*, vol. 81, no. 8, pp. 1136 –1150, Aug. 1993.

[91] S. Elnaffar and P. Martin, "Characterizing computer systems' workloads," Queen's Univ., Kingston, ON, Canada, Tech. Rep. 2002-461, 2002.

[92] N. Snellman, A. Ashraf, and I. Porres, "Towards automatic performance and scalability testing of rich internet applications in the cloud," in *Proc. 37th EUROMICRO Conf. Softw. Eng. Adv. Appl.*, Sep. 2011, pp. 161–169.

[93] M. Andreolini, M. Colajanni, and P. Valente, "Design and testing of scalable web-based systems with performance constraints," in *Proc. Workshop Techn., Methodologies Tools Perform. Evaluation Complex Syst.*, 2005, pp. 15–25.

[94] S. Dawar, S. Meer, J. Keeney, E. Fallon, and T. Bennet, "Cloudifying mobile network management: Performance tests of event distribution and rule processing," in *Proc. 5th Int. Conf. Mobile Netw. Manag.*, 2013, pp. 94–107.

[95] B. L. Farrell, R. Menninger, and S. G. Strickland, "Performance testing & analysis of distributed client/server database systems," in *Proc. Comput. Manag. Group Conf.*, 1998, pp. 910–921.

[96] R. Hayes, "How to load test e-commerce applications," in *Proc. Comput. Manag. Group Conf.*, 2000, pp. 275–282.

[97] J. A. Meira, E. C. de Almeida, G. Suny, Y. L. Traon, and P. Valduriez, "Stress testing of transactional database systems." *J. Inf. Data Manag.*, vol. 4, no. 3, pp. 279–294, 2013.

[98] J. K. Merton, "Evolution of performance testing in a distributed client server environment," in *Proc. Comput. Manag. Group Conf.*, 1999, pp. 118–124.

[99] A. Savoia, "Web load test planning: Predicting how your web site will respond to stress," *STQE Mag.*, vol. March/April 2001, pp. 32–37, 2001.

[100] L. T. Costa, R. M. Czekster, F. M. de Oliveira, E. de M. Rodrigues, M. B. da Silveira, and A. F. Zorzo, "Generating performance test scripts and scenarios based on abstract intermediate models," in *Proc. 24th Int. Conf. Softw. Eng. Knowl. Eng.*, 2012, pp. 112–117.

[101] M. B. da Silveira, E. de M. Rodrigues, A. F. Zorzo, L. T. Costa, H. V. Vieira, and F. M. de Oliveira, "Reusing functional testing in order to decrease performance and stress testing costs," in *Proc. 23rd Int. Conf. Softw. Eng. Knowl. Eng.*, 2011, pp. 470–474.

[102] I. de Sousa Santos, A. R. Santos, and P. de Alcantara dos S. Neto, "Generation of scripts for performance testing based on UML models," in *Proc. 23rd Int. Conf. Softw. Eng. Knowl. Eng.*, 2011, pp. 258–263.

[103] X. Wang, B. Zhou, and W. Li, "Model based load testing of web applications," in *Proc. Int. Symp. Parallel Distrib. Process. Appl.*, Sep. 2010, pp. 483–490.

[104] 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 *Proc. 37th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2007, pp. 612 –617.

[105] K. Kant, V. Tewary, and R. Iyer, "Geist: A web traffic generation tool," *Computer Performance Evaluation: Modelling Techniques and Tools*, Lecture Notes in Computer Science, vol. 2324, pp. 227–232, 2002.

[106] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber, "Realistic load testing of web applications," in *Proc. Conf. Softw. Maintenance Reeng.*, 2006, pp. 57–70.

[107] C. Lutteroth and G. Weber, "Modeling a realistic workload for performance testing," in *Proc. 12th Int. IEEE Enterprise Distrib. Object Comput. Conf.*, 2008, pp. 149–158.

[108] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, "Model-based performance testing in the cloud using the mbpet tool," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 423–424.

[109] A. J. Maalej, M. Hamza, M. Krichen, and M. Jmaiel, "Automated significant load testing for WS-BPEL compositions," in *Proc. IEEE 6th Int. Conf. Softw. Testing, Verification Validation Workshops*, Mar. 2013, pp. 144–153.

[110] A. J. Maalej, M. Krichen, and M. Jmaiel, "Conformance testing of WS-BPEL compositions under various load conditions," in *Proc. IEEE 36th Annu. Comput. Softw. Appl. Conf.*, Jul. 2012, pp. 371–371.

[111] P. Zhang, S. Elbaum, and M. B. Dwyer, "Compositional load test generation for software pipelines," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 89–99.

[112] Y. Gu and Y. Ge, "Search-based performance testing of applications with composite services," in *Proc. Int. Conf. Web Inf. Syst. Mining*, 2009, pp. 320–324.

[113] M. D. Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based testing of service level agreements," in *Proc. 9th Annu. Conf. Genetic Evolutionary Comput.*, 2007, pp. 1090–1097.

[114] Y. Cai, J. Grundy, and J. Hosking, "Experiences integrating and scaling a performance test bed generator with an open source case tool," in *Proc. 19th IEEE Int. Conf. Automated Softw. Eng.*, 2004, pp. 36–45.

[115] Y. Cai, J. Grundy, and J. Hosking, "Synthesizing client load models for performance engineering via web crawling," in *Proc. 22nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2007, pp. 353–362.

[116] G. Canfora, M. D. Penta, R. Esposito, and M. L. Villani, "An approach for QoS-aware service composition based on genetic algorithms," in *Proc. Conf. Genetic Evolutionary Comput.*, 2005, pp. 1069–1075.

[117] M. M. Maccabee and S. Ma, "Web application performance: Realistic work load for stress test," in *Proc. Comput. Manag. Group Conf.*, 2002, pp. 353–362.

[118] G. M. Leganza, "The stress test tutorial," in *Proc. Comput. Manag. Group Conf.*, 1991, pp. 994–1004.

[119] 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.

[120] N. Mi, G. Casale, L. Cherkasova, and E. Smirni, "Burstiness in multi-tier applications: Symptoms, causes, and new models," in *Proc. 9th ACM/IFIP/USENIX Int. Conf. Middleware*, 2008, pp. 265–286.

[121] F. Borges, A. Gutierrez-Milla, R. Suppi, and E. Luque, "Optimal run length for discrete-event distributed cluster-based simulations," in *Proc. Int. Conf. Comput. Sci.*, 2014, pp. 73–83.

[122] D. Meisner, J. Wu, and T. F. Wenisch, "Bighouse: A simulation infrastructure for data center systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2012, pp. 35–45.

[123] M. J. Harrold and G. Rothermel, "Siemens Programs, HR Variants," http://www.cc.gatech.edu/aristotle/Tools/subjects/, Oct. 2013.

[124] G.-H. Kim, Y.-G. Kim, and S.-K. Shin, "Software performance test automation by using the virtualization," in *IT Convergence and Security 2012*, K. J. Kim and K.-Y. Chung, Eds. Doetinchem, Netherlands: Springer Science & Business Medias, 2013, pp. 1191–1199.

[125] T. Bear. (2006). Shootout: Load Runner vs The Grinder vs Apache JMeter [Online]. Available: http://blackanvil.blogspot.com/2006/06/shootout-load-runner-vs-grinder- vs.html

[126] A. Podelko. Load Testing Tools [Online]. Available: http://alexanderpodelko.com/PerfTesting.html#LoadTestingTools, 2015.

[127] C. Vail. (2005). Stress, load, volume, performance, benchmark and base line testing tool evaluation and comparison [Online]. Available: http://www.vcaa.com/tools/loadtesttoolevaluation-chart-023.pdf, visited 2014-11-24, 2005.

[128] WebLOAD product overview [Online]. Available: http://www.radview.com/webload-download/, 2015.

[129] HP LoadRunner software [Online]. Available: http://www8.hp.com/ca/en/software-solutions/loadrunner-load-testing/, 2015.

[130] Apache JMeter [Online]. Available: http://jakarta.apache.org/jmeter/, 2015.

[131] Microsoft Exchange Load Generator (LoadGen) [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=14060, 2015.

[132] X. Che and S. Maag, "Passive testing on performance requirements of network protocols," in *Proc. 27th Int. Conf. Adv. Inf. Netw. Appl. Workshops*, 2013, pp. 1439–1444.

[133] S. Dimitrov and T. Stoilov, “Loading test of apache HTTP server by video file and usage measurements of the hardware components,” in *Proc. 14th Int. Conf. Comput. Syst. Technol.*, 2013, pp. 59–66.

[134] M. Murth, D. Winkler, S. Biffl, E. Kuhn, and T. Moser, “Performance testing of semantic publish/subscribe systems,” in *Proc. Int. Conf. On Move Meaningful Internet Syst.*, 2010, pp. 45–46.

[135] C. Dumitrescu, I. Raicu, M. Ripeanu, and I. Foster, “Diperf: An automated distributed performance testing framework,” in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, 2004, pp. 289–296.

[136] J. A. Meira, E. C. de Almeida, Y. L. Traon, and G. Sunye, “Peer-to-peer load testing,” in *Proc. IEEE 5th Int. Conf. Softw. Testing, Verification Validation*, 2012, pp. 642–647.

[137] J. Xie, X. Ye, B. Li, and F. Xie, “A configurable web service performance testing framework,” in *Proc. 10th IEEE Int. Conf. High Perform. Comput. Commun.*, 2008, pp. 312–319.

[138] N. Baltas and T. Field, “Continuous performance testing in virtual time,” in *Proc. 9th Int. Conf. Quantitative Evaluation Syst.*, 2012, pp. 13–22.

[139] D. A. Menasce, “Workload characterization,” *IEEE Internet Comput.*, vol. 7, no. 5, pp. 89–92, Sep./Oct. 2003.

[140] Q. Gao, W. Wang, G. Wu, X. Li, J. Wei, and H. Zhong, “Migrating load testing to the cloud: A case study,” in *Proc. IEEE 7th Int. Symp. Service Oriented Syst. Eng.*, Mar. 2013, pp. 429–434.

[141] M. Yan, H. Sun, X. Wang, and X. Liu, “Building a taas platform for web service load testing,” in *Proc. IEEE Int. Conf. Cluster Comput.*, 2012, pp. 576–579.

[142] J. Zhou, S. Li, Z. Zhang, and Z. Ye, “Position paper: Cloud-based performance testing: Issues and challenges,” in *Proc. Int. Workshop Hot Topics Cloud Services*, 2013, pp. 55–62.

[143] M. Grechanik, C. Csallner, C. Fu, and Q. Xie, “Is data privacy always good for software testing?” in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 368 –377.

[144] Y. Wang, X. Wu, and Y. Zheng, “Efficient evaluation of multifactor dependent system performance using fractional factorial design,” in *Trust and Privacy in Digital Business*. New York, NY, USA: Springer, 2004, pp. 142–151.

[145] A. Bertolino, G. Angelis, A. Marco, P. Inverardi, A. Sabetta, and M. Tivoli, “A framework for analyzing and testing the performance of software services,” in *Proc. 3rd Int. Symp. Leveraging Appl. Formal Methods, Verification Validation*, 2009, pp. 206–220.

[146] X. Meng, “Designing approach analysis on small-scale software performance testing tools,” in *Proc. Int. Conf. Electron. Mech. Eng. Inf. Technol.*, Aug. 2011, pp. 4254–4257.

[147] C. H. Kao, C. C. Lin, and J.-N. Chen, “Performance testing framework for rest-based web applications,” in *Proc. 13th Int. Conf. Quality Softw.*, Jul. 2013, pp. 349–354.

[148] S. Dunning and D. Sawyer, “A little language for rapidly constructing automated performance tests,” in *Proc. 2nd Joint WOSP/SIPEW Int. Conf. Perform. Eng.*, 2011, pp. 371–380.

[149] M. Dhote and G. Sarate, “Performance testing complexity analysis on Ajax-based web applications,” *IEEE Softw.*, vol. 30, no. 6, pp. 70–74, Nov./Dec. 2013.

[150] N. Stankovic, “Distributed tool for performance testing,” in *Softw. Eng. Research and Practice*. New York, NY, USA: 2006, pp. 38–44.

[151] N. Stankovic, “Patterns and tools for performance testing,” in *Proc. IEEE Int. Conf. Electro/Inf. Technol.*, 2006, pp. 152 –157.

[152] Wireshark - Go Deep [Online]. Available: http://www.wireshark.org/, 2015.

[153] Selenium - Web Browser Automation [Online]. Available: http://seleniumhq.org/, 2015.

[154] S. Shirodkar and V. Apte, “Autoperf: An automated load generator and performance measurement tool for multi-tier software systems,” in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 1291–1292.

[155] M. A. S. Netto, S. Menon, H. V. Vieira, L. T. Costa, F. M. de Oliveira, R. Saad, and A. F. Zorzo, “Evaluating load generation in virtualized environments for software performance testing,” in *Proc. IEEE Int. Symp. Parallel Distrib. Processing Workshops PhD Forum*, May 2011, pp. 993–1000.

[156] J. White and A. Pilbeam, “A survey of virtualization technologies with performance testing,” *CoRR*, vol. abs/1010.3233, 2010.

[157] G.-B. Kim, “A method of generating massive virtual clients and model-based performance test,” in *Proc. 5th Int. Conf. Quality Softw.*, 2005, pp. 250–254.

[158] Shunra [Online] Available: http://www8.hp.com/us/en/software-solutions/network-virtualization/, 2015.

[159] R. K. Mansharamani, A. Khanapurkar, B. Mathew, and R. Subramanyan, “Performance testing: Far from steady state,” in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf. Workshops*, Jul. 2010, pp. 341–346.

[160] M. Grechanik, C. Fu, and Q. Xie, “Automatically finding performance problems with feedback-directed learning software testing,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 156–166.

[161] J.-N. Juang, *Applied System Identification*, 1st ed. Englewood Cliffs, NJ, USA: Prentice-Hall, 1993.

[162] L. Ljung, *System Identification: Theory for the User*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1987.

[163] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, “Evaluating the accuracy of Java profilers,” in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, 2010, pp. 187–197.

[164] G. M. Leganza, “Coping with stress tests: Managing the application benchmark,” in *Proc. Comput. Manag. Group Conf.*, 1990, pp. 1018–1026.

[165] A. Bertolino, G. D. Angelis, and A. Sabetta, “VCR: Virtual capture and replay for performance testing,” in *Proc. 23rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2008, pp. 399–402.

[166] PerfMon [Online]. Available: http://technet.microsoft.com/en-us/library/bb490957.aspx, 2015.

[167] Munin [Online]. Available: http://munin-monitoring.org/, 2015.

[168] Net SNMP [Online]. Available: http://www.net-snmp.org/, 2015.

[169] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, “A systematic survey of program comprehension through dynamic analysis,” *IEEE Trans. Softw. Eng.*, vol. 35, no. 5, pp. 684–702, Sep./Oct. 2009.

[170] T. A. S. Foundation, Log4j [Online]. Available: http://logging.apache.org/log4j/2.x/

[171] T. O. Group, Application Response Measurement - ARM [Online]. Available: http://regions.cmg.org/regions/cmgarmw/, 2015.

[172] Sarbanes-Oxley Act of 2002 [Online]. Available: http://www.soxlaw.com/, 2015.

[173] E. M. Friedman and J. L. Rosenberg, “Web load testing made easy: Testing with WCAT and WAST for windows applications,” in *Proc. Comput. Manag. Group Conf.*, 2003, pp. 57–82.

[174] G. Din, I. Schieferdecker, and R. Petre, “Performance test design process and its implementation patterns for multi-services systems,” in *Proc. 20th IFIP TC 6/WG 6.1 Int. Conf. Testing Softw. Commun. Syst.*, 2008, pp. 135–152.

[175] P. Tran, J. Gosper, and I. Gorton, “Evaluating the sustained performance of cots-based messaging systems,” *Softw. Testing, Verification Rel.*, vol. 13, no. 4, pp. 229–240, 2003.

[176] X. Yang, X. Li, Y. Ji, and M. Sha, “Crownbench: A grid performance testing system using customizable synthetic workload,” in *Proc. 10th Asia-Pacific Web Conf. Progress WWW Res. Develop.*, 2008, pp. 190–201.

[177] A. L. Glaser, “Load testing in an ir organization: Getting by ’with a little help from my friends’,” in *Proc. Comput. Manag. Group Conf.*, 1999, pp. 686–698.

[178] D. Grossman, M. C. McCabe, C. Staton, B. Bailey, O. Frieder, and D. C. Roberts, “Performance testing a large finance application,” *IEEE Softw.*, vol. 13, no. 5, pp. 50–54, Sep. 1996.

[179] S. Duttagupta and M. Nambiar, “Performance extrapolation for load testing results of mixture of applications,” in *Proc. 5th UKSim Eur. Symp. Comput. Model. Simul.*, Nov. 2011, pp. 424–429.

[180] Z. M. Jiang, A. E. Hassa, G. Hamann, and P. Flora, “An automated approach for abstracting execution logs to execution events,” *J. Softw. Maintenance Evolution*, vol. 20, pp. 249–267, July 2008.

[181] K. C. Foo, Z. M. Jiang, B. Adams, Y. Z. Ahmed E. Hassan, and P. Flora, “Mining performance regression testing repositories for automated performance analysis,” in *Proc. 10th Int. Conf. Quality Softw.*, Jul. 2010, pp. 32–41.

[182] S. Ghaith, M. Wang, P. Perry, and J. Murphy, “Profile-based, load-independent anomaly detection and analysis in performance regression testing of software systems,” in *Proc. 17th Eur. Conf. Softw. Maintenance Reeng.*, 2013, pp. 379–383.

[183] S. Ghaith, “Analysis of performance regression testing data by transaction profiles,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 370–373.

[184] H. Malik, H. Hemmati, and A. E. Hassan, “Automatic detection of performance deviations in the load testing of large scale systems,” in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1012–1021.

[185] H. Malik, B. Adams, and A. E. Hassan, "Pinpointing the subsystems responsible for the performance deviations in a load test," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, Nov. 2010, pp. 201–210.

[186] H. Malik, Bram, Adams, A. E. Hassan, P. Flora, and G. Hamann, "Using load tests to automatically compare the subsystems of a large enterprise system," in *Proc. IEEE 34th Annu. Comput. Softw. Appl. Conf.*, Jul. 2010, pp. 117–126.

[187] H. Malik, Z. M. Jiang, B. Adams, P. Flora, and G. Hamann, "Automatic comparison of load tests to support the performance analysis of large enterprise systems," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, Mar. 2010, pp. 222–231.

[188] T. H. D. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. N. Nasser, and P. Flora, "Automated verification of load tests using control charts," in *Proc. 18th Asia Pacific Softw. Eng. Conf.*, Dec. 2011, pp. 282–289.

[189] M. D. Syer, B. Adams, and A. E. Hassan, "Industrial case study on supporting the comprehension of system behaviour under load," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, 2011, pp. 215–216.

[190] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "Leveraging performance counters and execution logs to diagnose memory-related performance issues," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 110–119.

[191] J. K. Merton, "Performance testing in a client-server environment," in *Proc. Comput. Manag. Group Conf.*, 1997, pp. 594–601.

[192] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous Java performance evaluation," in *Proc. 22nd Annual ACM SIGPLAN Conf. Object-oriented Programming Syst. Applications*, OOPSLA '07, Montreal, Quebec, Canada, 2007, pp. 57–76.

[193] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, indexing, clustering, and retrieving system history," in *Proc. 20th ACM Symp. Operating Syst. Principles*, 2005, pp. 105–118.

[194] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance debugging in the large via mining millions of stack traces," in *Proc. 34th Int. Conf. Softw. Eng.*, Jun. 2012, pp. 145–155.

[195] M. Jiang, M. A. Munawar, T. Reidemeister, and P. A. S. Ward, "Detection and diagnosis of recurrent faults in software systems by invariant analysis," in *Proc. 11th IEEE High Assurance Syst. Eng. Symp.*, 2008, pp. 323–332.

[196] W. Visser, "Who really cares if the program crashes?" in *Proc. 16th Int. SPIN Workshop Model Checking Softw.*, 2009, p. 5.

Zhen Ming Jiang received the BMath and MMath degrees in computer science from the University of Waterloo, and the PhD degree from the School of Computing at the Queen's University. He is an assistant professor in the Department of Electrical Engineering and Computer Science, York University. Prior to joining York, he was at BlackBerry Performance Engineering Team. His research interests lie within software engineering and computer systems, with special interests in software performance engineering, mining software repositories, source code analysis, software architectural recovery, software visualizations and debugging and monitoring of distributed systems. Some of his research results are already adopted and used in practice on a daily basis. He is the cofounder and co-organizer of the annually held International Workshop on Large-Scale Testing (LT). He also received several Best Paper Awards including ICSE 2015 (SEIP track), ICSE 2013, WCRE 2011, and MSR 2009 (challenge track). He is a member of the IEEE.

Ahmed E. Hassan received the PhD degree in computer science from the University of Waterloo. He is the NSERC/BlackBerry Software engineering chair at the School of Computing at Queens University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of *IEEE Transactions on Software Engineering*, *Springer Journal of Empirical Software Engineering*, and *Springer Journal of Computing*. He is a member of the IEEE.